Duke$$$

An NUS Software Development Project

1. Table of Contents

2. Introduction

`Duke$$$` is a one-of-a-kind desktop application that aims to provide financial tracking for international university students who are on semester exchange to NUS and prefer typing over clicking. Built using Java, `Duke$$$` is developed to have a Graphic User Interface (GUI), but uses a Command Line Interface (CLI) for all user interactions to cater to this preference.

2.1. Purpose of Document

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$$$`, and any general audience who would like to read up more on the design and implementations made in `Duke$$$`. 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
Appendix.. Documentation	To describe how the documentation of the developer guide was carried out

2.4. User Guide

The User Guide for `Duke$$$` can be found here.

2.5. Markup

This document uses a few, simple markups to convey information with greater clarity.

These markups include:

- `Classes` - Refers to Java Objects implemented
- **Method** - Refers to Methods implemented in these Java Objects
- *<code>* - Refers to code excerpts
- `Entities` - Refers to entities that do not exist in code but are useful when trying to understand the current implementation
- ***Variable*** - Refers to a variable that can be modified

3. Setting Up

This section describes the procedures for setting up `Duke$$$` in a development environment.

3.1. Prerequisites

Please ensure that your Desktop has

1. JDK 11 or later
2. IntelliJ IDE

3.2. Setting up the project on your own Device

The following steps detail how to set up the most recent `Duke$$$` codebase into your own Device:

1. Fork the `Duke$$$` repository and clone it onto your desktop
    - ➢ `Duke$$$` Repository: https://github.com/AY1920S1-CS2113T-F09-1/main
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

`Duke$$$` has now been successfully set up on your Device.

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 (6 weeks), 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:

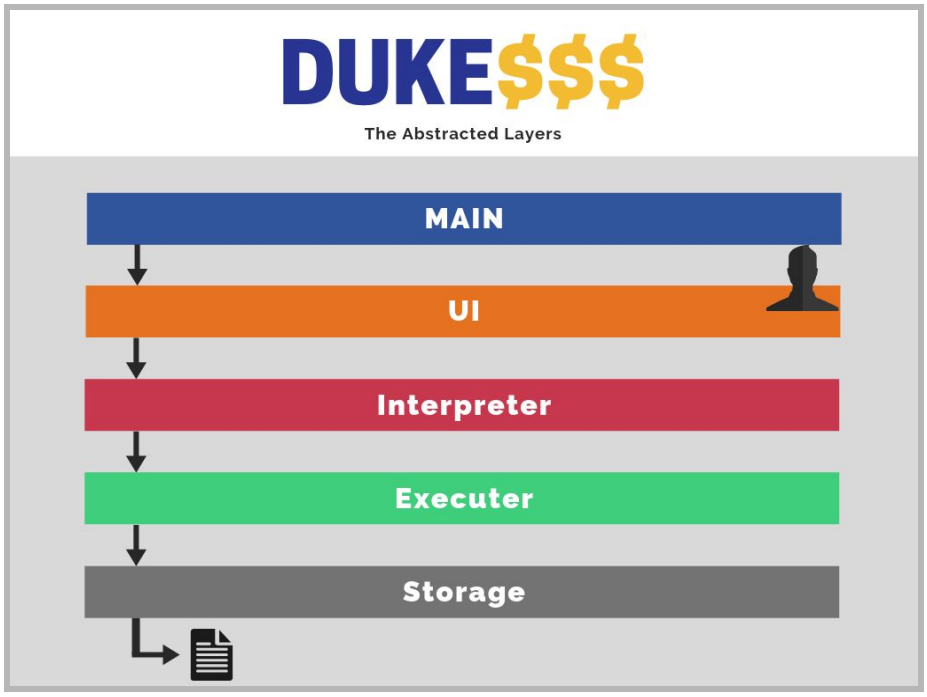

Figure 4.1.1. Overall Architecture of Duke$$$

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$$$` **[Ver 2.0]**
- Creating new accounts for new users **[Ver 2.0]**
- Providing a link to the User Guide **[Ver 2.0]**

4.3. Ui Layer

The `Ui Layer` (User Interface 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`. The diagram below illustrates the dependencies between different Ui Class.

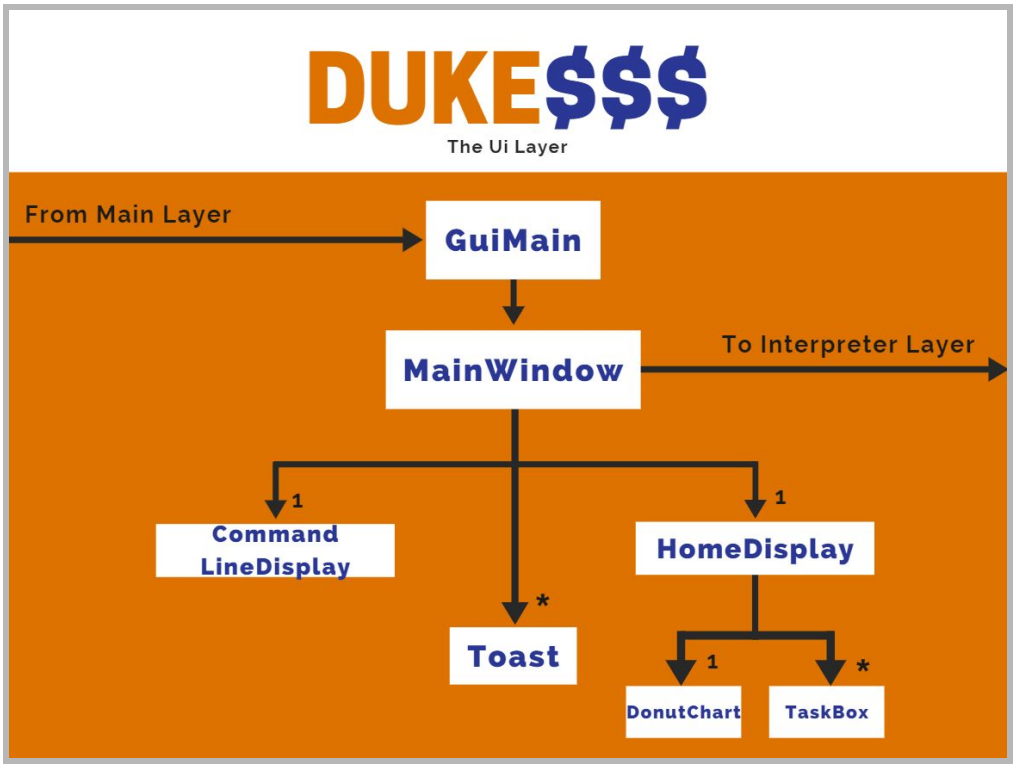

Figure 4.3.1. Overall Structure of the Ui Layer

As seen in the diagram above, the `Ui Layer` has two main classes: `GuiMain`, which interacts with the `Main Layer`, and `MainWindow`, which manages all other components in the `Ui Layer` and interacts with the `Interpreter Layer`. The sections below explore in greater detail the classes in this layer.

4.3.1. Main Ui Classes

The `Ui Layer` is managed by two main Java Classes.

`GuiMain`

- Launches the Graphical User Interface
- Manages any methods that need to be executed on start or end of the application

`MainWindow`

- Manages `User` inputs
- Interacts with the `Interpreter` Layer

- Displaying components that always remain visible to the `User`
    - E.g. Header, InputBar
- Managing other interchangeable displays
    - E.g. `HomeDisplay`, `CommandLineDisplay`

4.3.2. Ui Displays

The following details the various displays that can be shown on the `MainWindow`.

`HomeDisplay` displays:

- `User's` balance and expenses in the form of a `DonutChart`
- `User's` expenses based on trackable tags in the form of a StackedBarChart
- `User's` `Tasks` in the form of `TaskBoxes`

`CommandLineDisplay` displays:

- `Duke$$$'s` long, text-based response to any `User`-initiated `Commands`

4.3.3. Ui Components

The following details the various components that are used in Graphical User Interface:

`TaskBox` holds:

- `User's` `Tasks`

`DonutChart` holds:

- `User's` balance and expenses displayed in a pie chart with a hole in the center
- `User's` balance as a text in the center of the `DonutChart`

`Toast` holds:

- `Duke$$$'s` short, text-based response to any `User`-initiated `Commands`

4.3.4. UiCode

`UiCode` is an enumeration housed by `InfoCapsules` which inform `MainWindow` what changes need to be made to the Graphical User Interface.

These `UiCodes` include:

- **CLI -** Print to the `CommandLineDisplay`
- **CLEAR_CLI -** Clears the `CommandLineDisplay`
- **DISPLAY_CLI -** Display the `CommandLineDisplay`
- **DISPLAY_HOME -** Display the `HomeDisplay`
- **ERROR -** Inform the `User` of an error
- **EXIT -** Launch the exit sequence
- **TESTER -** Switch `Duke$$$` to Testing Mode
- **TOAST -** Display a `Toast` message

4.4. Interpreter Layer

The `Interpreter Layer` is responsible for interpreting the requests from the `Ui Layer` and calling on the `Executor Layer` to carry out these requests. It receives an `InfoCapsule` per call from the `Executor Layer` which it then passes on to the `Ui Layer`. The `Interpreter Layer` consists of solely the `Interpreter` Class.

4.5. Executor Layer

The `Executor Layer` receives instructions from the `Interpreter Layer` and executes either the `Command` or `Accessor` required of it, returning an `InfoCapsule` as a result.

4.5.1. Main Executor Class

All requests to the `Executor Layer` are handled by the `Executor` Class. The `Executor` is responsible for:

- Creating and Executing `Commands`
- Creating and Executing `Accessors`

- Returning an `InfoCapsule` detailing the outcome of running the `Command/Accessor`

4.5.2. Commands

`Commands` are features which the `User` may choose to ask `Duke$$$` to carry out. Each `Command` exists as a Java Class with a specific `CommandType` assigned to them.

These are the `Commands` available to the `User` in the format:

Java_Class [specific_command_type] - Description

- `CommandGetSpendingByDay` - Outputs the total spending for a given day
- `CommandGetSpendingByWeek` - Outputs the total spending for the current week
- `CommandGetSpendingByMonth` - Outputs the total spending for a given month
- `CommandGetSpendingByYear` - 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`
- `CommandEdit` - Edits the existing `Receipts`
- `CommandSchedule` - Displays the schedule based on a given date
- `CommandConvert` - Converts currency between countries requested by `User`
- `CommandWeather` - Displays real time weather information based on period requested by `User`
- `CommandBudget` - Allows `User` to set a budget and displays percentage statistics on amount used up or exceeded

- `CommandExport` - Allows `User` to export wallet expenditures real-time into an excel file so as to offer meaningful data arrangement
- `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
- `CommandHelp` - Outputs all the commands description
- `CommandSave` - Saves the `User`'s Data
- `CommandLoad` - Loads the `User`'s Data
- `CommandTaglist` - Lists all the receipts corresponding to the tag entered by `User`
- `CommandDateList` - Lists all the receipts corresponding to the date entered by `User`
- `CommandAdd` - Adds two numbers
- `CommandSub` - Subtracts two numbers
- `CommandMul` - Multiplies two numbers
- `CommandDiv` - Divides two numbers
- `CommandMajorExpense` - Gives a list of receipts where cash property is above/equal to the positive integer entered by `User`
- `CommandStats` - Gives statistics for the tag entered by `User`

4.5.3. Accessors

`Accessors` are Java Classes which `Duke$$$` uses to access data stored in the `Storage Layer`. Each `Accessor` has its own `AccessType` assigned to it.

These are the `Accessors` available to the `Duke$$` in the format:

Java_Class [specific_access_type] - ***Description***

- `AccessDeny` - Throws an error, indicating that access to requested data was denied
- `AccessPieChartData` - Accesses the `Wallet` data of the `User` and converts it into the required data type to be displayed in a `DonutChart`
- `AccessTaskList` - Accesses the `TaskList` of the `User` and returns it
- `AccessWallet` - Accesses the `Wallet` of the `User` and returns it
- `AccessWalletBalance` - Accesses the balance property of the `Wallet` of the `User` and returns it
- `AccessWalletExpenses` - Accesses the receipts property of the `Wallet` of the `User` and returns the sum of all expenses

4.6. Storage Layer

The `Storage Layer` is responsible for loading, holding and saving the `User`'s data.

4.6.1. Main Storage Classes

The `Storage Layer` is managed primarily by these three classes:

`StorageManager`

- Interacts with and handles any request received from the `Executor Layer`

`StorageTask`

- Handles all loading and saving of `Task`-related data into a .txt file

`StorageWallet`

- Handles all loading and saving of `Wallet`-related data into a .txt file

4.6.2. Models

The `Storage Layer` also holds the data models on which `Duke$$$` is built upon.

`Task` is an abstract Java Class that stores the `User`'s `Tasks`. Each `Task` is assigned their own `TaskType`. The different types of `Tasks` are as follows:

- `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'
- `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`

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 by the utility class `Parser`. 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:

<commandType 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:

commandType

This placeholder specifies the `CommandType` of the `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 ***commandType***.

- **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 ***commandType*** 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. Calculator

The calculator feature of DUKE$$$ is facilitated by the `CommandAdd`, `CommandSub`, `CommandDiv` and `CommandMul` respectively. All the above commands inherits from the parent class `Command` and overrides the execution method to output the result of the arithmetic between two numbers only.

Given below is a more detailed explanation of the feature and how it works.

5.3.1 Add

Step 1: `User` inputs command in the following format through the GUI.

add 1 / 2

Step 2: `UI Layer` captures the UserInput and sends it to the parser under the interpreter layer which parses for the command and `CommandAdd` is invoked

Step 3: Under the command, the primary input and flag input is parsed and the arithmetic logic is performed in the `Executor layer`

Step 4: The result is then printed out to the `User` from the `UI Layer`

5.3.2 Sub

Step 1: `User` inputs command in the following format through the GUI.

sub 1 / 2

Step 2: `UI Layer` captures the UserInput and sends it to the parser under the interpreter layer which parses for the command and `CommandSub` is invoked

Step 3: Under Command Sub, the primary input and flag input is parsed and the arithmetic logic is performed in the executor layer

Step 4: The result is then printed out to the `User` from the `UI Layer`

5.3.3 Div

Step 1: `User` inputs command in the following format through the GUI.

div 1 / 2

Step 2: `UI Layer` captures the UserInput and sends it to the parser under the interpreter layer which parses for the command and CommandDiv is invoked

Step 3: Under the command, the primary input and flag input is parsed and the arithmetic logic is performed in the executor layer

Step 4: The result is then printed out to the user from the `UI Layer`

5.3.4 Mul

Step 1: `User` inputs command in the following format through the GUI.

mul 1 / 2

Step 2: `UI Layer` captures the UserInput and sends it to the parser under the interpreter layer which parses for the command and `CommandMul` is invoked

Step 3: Under the command, the primary input and flag input is parsed and the arithmetic logic is performed in the executor layer

Step 4: The result is then printed out to the `User` from the `UI Layer`

Design consideration

Alternative 1: Put all the commands inside one command calculator class with 4 different constructors and use switch case for the different arithmetic functionalities

Pros: Allows for a better and more condensed program

Future implementation

These are the features considered for future implementation:

1. Scientific Calculator with more functionalities
2. Ability to input multiple numbers instead of just two

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 the execution method overrides to output the total amount of money spent by the user, categorically.

The general flow of the logic constitutes of the user input being parsed into the interpreter layer where the CommandType is identified. According to the commandType, the corresponding Command is invoked in the executor. In the Command, the various execute method calls the relevant methods and functions to get the expenditure using data from the `storageManager.` which is sent back to the GUI.

The diagram below shows the sequence diagram for the expenditure commands.

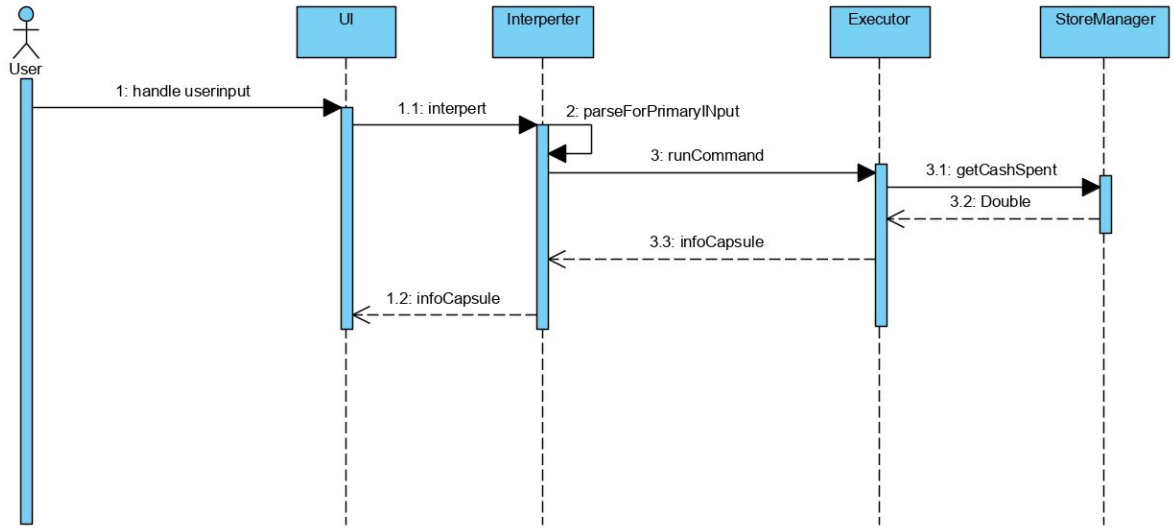

Figure 5.3.1. Sequence diagram for expenditure feature

Given below are the 4 different commands that outputs the expenditure for the day,week,month,year respectively..

5.3.1. Daily Expenditure

Gives the total amount of expenditure for the given day.

Given below is a more detailed explanation of the feature and a break down of how it works with examples.

Step 1: `User` inputs the command to get the total expenditure for the day in the following format through the GUI:

```
expendedday 2019-11-11
```

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

`CommandType` - **EXPENDEDDAY**

Step 3: Since the `CommandType` is **`EXPENDEDDAY`**, the program invokes `CommandGetSprendingByDay`.

Step 4: `CommandGetSpendingByDay` 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 date

Based on the sample userInput specified above,

PrimaryInput -- `2019-11-11`

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

November - `11` (Numerical value of the month)

Step 5: Thirdly, we pass the ***PrimaryInput*** to the `Executor Layer.` In the `Executor Layer` method **outputExpenditureForInput** is invoked. The program runs through all the receipts and gets the receipt which corresponds to the date details from the ***PrimaryInput***. Then **outputExpenditureForInput** gets details by accessing methods in the `storagemanager` in the `Storage Layer.`

Step 6: We make an ***outputStr*** containing the total amount of expenditure for the day and send it to the `Interpreter Layer` via an `InfoCapsule` . The `Interpreter Layer` then sends the `InfoCapsule` to the `Ui Layer.`

5.3.2. Weekly Expenditure

Gives the amount of money expended in the current week.

Given below is a more detailed explanation of the feature and a break down of how it works with examples.

Step 1: `User` inputs the command to get the total expenditure for the week in the following format through the GUI:

expendedweek

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

`CommandType` - **EXPENDEDWEEK**

Step 3: Since the `CommandType` is **EXPENDEDWEEK**, the program invokes `CommandGetSprendingByWeek`.

Step 4: The `CommandGetSprendingByWeek` is instantiated and calls the **execute** method to attain the total expenditure for the week. **checkIfInputIsEmpty** method checks if the input is not empty. Following that, **getListOfAllDaysInWeek** method finds the current value of the day of the week and finds all the days in the week that has passed and stores it in a list. The total amount of expenditure is saved in *total* variable.

Step 5: We make an ***outputStr*** containing the total amount of expenditure for the year and send it to the `Interpreter Layer` via an `InfoCapsule` . The `Interpreter Layer` then sends the `InfoCapsule` to the `Ui Layer.`

5.3.3. Monthly Expenditure

Gives the amount of expenditure for the given month.

Given below is a more detailed explanation of the feature and a break down of how it works with examples.

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

expendedmonth september /year 2019

Step 2: `Ui Layer` captures the `userInput` and sends it to the `Parser`, under the `interpreter layer` 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 calls 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 `Executor Layer.` In the `Executor Layer` method **findReceiptByMonthYear** is invoked. 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 by getting details by accessing methods in the `storagemanager` in the `Storage Layer.`

Step 6: We make an ***outputStr*** containing the total amount of expenditure for the month and send it to the `Interpreter Layer` via an `InfoCapsule` . The `Interpreter Layer` then sends the `InfoCapsule` to the `Ui Layer.`

5.3.4. Yearly Expenditure

Given below is a more detailed explanation of the feature and a break down of how it works with examples.

Step 1: `User` inputs the command to get the total expenditure for the year in the following format through the GUI:

`expendedyear 2019`

Step 2: `Ui Layer` captures the `userInput` and sends it to the `Parser`, under the `interpreter layer` 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 `Executor Layer.` In the `Executor Layer` method **findReceiptByYear** is invoked. 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 by getting details by accessing methods in the `storagemanager` in the `Storage Layer.`

Step 6: We make an ***outputStr*** containing the total amount of expenditure for the year and send it to the `Interpreter Layer` via an `InfoCapsule` . The `Interpreter Layer` then sends the `InfoCapsule` to the `Ui Layer.`

Design Consideration for Expenditure

Aspect: Input parameter specification

Consideration 1: (Currently using) The individual expenditure feature is implemented as a separate class.

Pro: Easier to debug.

Con: Harder to implement with multiple classes

Consideration 2: Combine all the expenditure features in a single class and use different parameter input to differentiate.

Pro: Easier to implement

Con: Might have a lot more of error and chances for bugs.

Aspect: Way of input by user

Consideration 1: (Currently using) GIve the year and month input to indicate the year and month, for which the expenditure is asked for.

Pro: More specific and can be used for a larger amount of data.

Con: Longer input for the user.

Consideration 2: Just give the command without any year or month input

Pro: Easier input for users

Con: Can only be used for a restricted amount of time period (this year)

Further Implementation for Expenditure

- Include percentage of expenditure of each tag
- Check if expenses are under budget
- Provide the statistics for the day with highest amount of expense in the week/month.

- Include time of each expenditure

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` in the `Executor Layer`. 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` in the `Interpreter Layer` 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` and is saved in the `Storage Layer`.

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

Further implementation

1. Allowing User to create new Receipts in other currencies
2. Allows Receipts to be automatically created by linking DUKE$$$ to bank accounts

5.5. Currency Converter

Currency Converter helps `User` to convert currencies between two countries using real-time exchange rates. At present this feature is able to convert currencies between 32 countries. The list of possible countries is provided in the appendix. The implementation of this feature requires the use of an API which provides real-time exchange rates using EUR as the base currency in json. We convert json into a java string and then parse to obtain the desired exchange rates for conversion.

The **currency conversion** feature of DUKE$$$ is facilitated by `CommandConvert`. It helps to do the logic for calculating the user's choice of desired currency and converted amount, along with the exchange rate that has been used.

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 Sequence diagram below shows the relay of messages between the various layers in sequence.

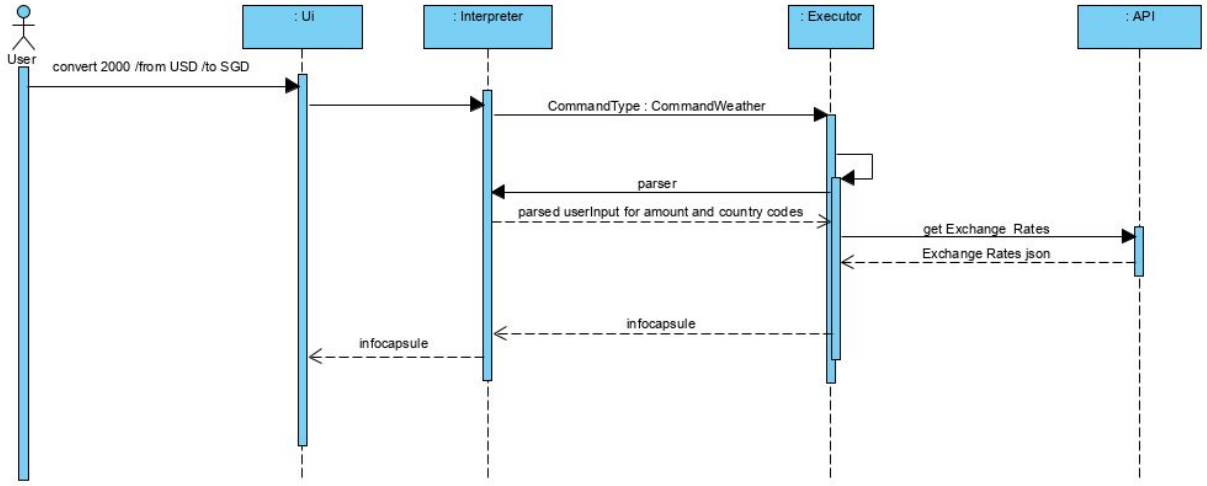

Figure 5.5.1. Sequence Diagram when converting currency between countries

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 GUI:

```
convert 2500 /from USD /to INR
```

Step 2 : `Ui Layer` captures the *userInput* and sends it to the `Interpreter Layer` . The `Interpreter Layer` identify its *CommandType*. Based on the sample *userInput* specified above,

CommandType -- **`CONVERT`**

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

Step 4: `CommandConvert` is instantiated with the *userInput* passed as an argument. *userInput* is parsed through 2 methods offered by `Parser` in the `Interpreter Layer` :

`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: Thereafter we pass the *flag1 details* and *flag2 details* to `CommandConvert` under the `Executor Layer` .

Step 6: Now, the method `consultTheCurrencyApi` will make an api call and return a string containing the json with the necessary exchange rates using 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 7: 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 8 : We make an *outputStr* containing the desired currency, amount and the exchange rate used and send it to the `Interpreter Layer` via an `InfoCapsule` . The `Interpreter Layer` then sends the `InfoCapsule` to the `Ui Layer`

Design Consideration

Aspect: Reduce Dependency

Alternative Implementation : Implement a class under `Storage Layer` which has the exchange rates at present from SGD to all other countries in the world.

Pros: This would reduce the dependency of the application on Internet access requirement for being able to get an approximation of converted amount. Also, at present this feature only works for countries which have been specified in the list which was mentioned earlier.

Cons : However, this would mean the application would not offer a real time solution especially since exchange rates fluctuate frequently. Hence the developing team would have to frequently change the stored exchange rate to offer close approximations.

Future Implementation

1. Allow users to specify countries which they want to favorite and the exchange rates of these currencies with SGD as base will be displayed under the home page
2. Provide nearest Money Changer outlet. This will be particularly useful for exchange students in Singapore.

5.6. Weather Display

Weather Display helps `User` to get real-time Weather Data in Singapore either for now, tomorrow or a forecast for 6 days including the present day.

The **weather display** feature of DUKE$$$ is facilitated by `CommandWeather`. The implementation of this feature requires the use of an API which provides real-time weather data of Singapore in json. 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 Sequence diagram below shows the relay of messages between the various layers in sequence.

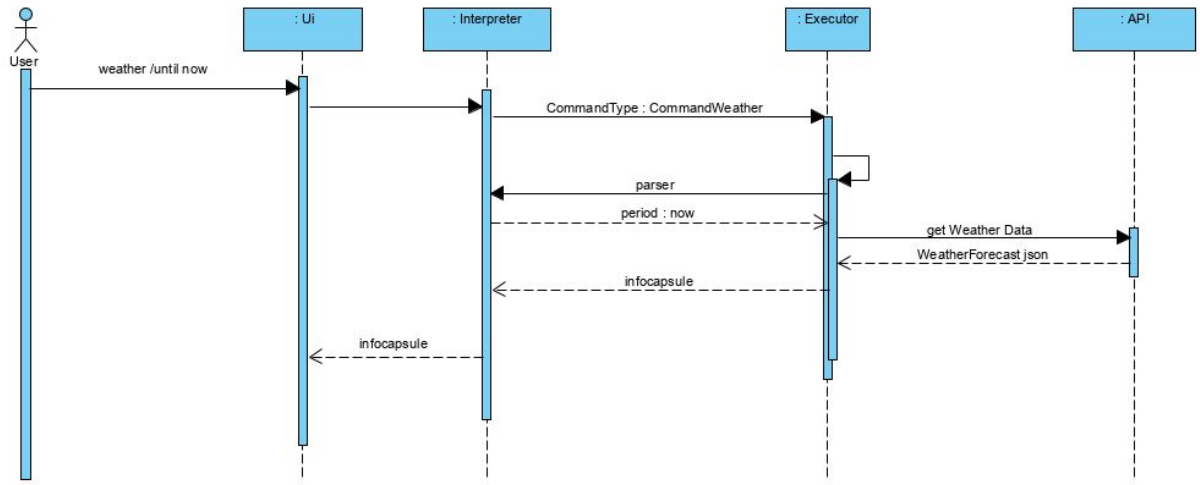

Figure 5.6.1. Sequence Diagram when displaying user requested weather data

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 GUI:

`weather /until tomorrow`

Step 2 : `Ui Layer` captures the ***userInput*** and sends it to the `Interpreter Layer` . The `Interpreter Layer` identifies the ***CommandType***. Based on the sample ***userInput*** specified above,

CommandType -- **`WEATHER`**

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

Step 4: `CommandWeather` is instantiated with the *userInput* passed as an argument. *userInput* is parsed by `Parser` in the `Interpreter Layer` :

> `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: *flag details* are passed to `CommandWeather` under the `Executor Layer` . Here, 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 6 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 make an *outputStr* containing the desired weather information send it to the `Interpreter Layer` via an `InfoCapsule` . The `Interpreter Layer` then sends the `InfoCapsule` to the `Ui Layer`

Design Consideration

Aspect: More OOP

Alternative Implementation : Abstracting out the current implementation of fetching data from a fixed url.

Pros: This would allow more reusability of the method which fetches json from the api url. Also, we can set url as a parameter which enables the program to be able to fetch multiple weather data across various countries.

Cons : However, abstract classes cannot be instantiated and they do not support multiple inheritances

Future Implementation

1. Allow users to specify the country for which they want weather data. By getting specific weather forecast can help exchange students to plan their travel to nearby countries over the weekends or during holidays while on exchange in NUS.

5.7. Export Wallet into csv

Export Wallet into csv helps `User` to export expenditure data which includes income receipts and expenses receipts. The implementation of this feature made use of the library opencsv. `CommandExport` executes the logic behind exporting wallet into a csv file.

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

Step 1: `Ui Layer` captures the *userInput* and sends it to the `Interpreter Layer` . The `Interpreter Layer` identifies the *CommandType*.

Step 2: Since the *CommandType* is **`EXPORT`**, `CommandExport` is instantiated with the *userInput* passed as an argument in the `Executor Layer`

Step 3: The `Storage Layer` is accessed now to get all the receipts stored in `Wallet`.

Step 4: Using `CSVWriter`, we export the `receipts` row by row into useful columns like `ID , Tag, Expenditure and Date` through string parsing. The income receipts are taken as negative expenditures indicated by a negative amount.

Step 5: We make an *outputStr* containing the folder location of the exported wallet data and send it to the `Interpreter Layer` via an `InfoCapsule` . The `Interpreter Layer` then sends the `InfoCapsule` to the `Ui Layer`

Design Consideration

Aspect: Automated csv storage and import csv ability

Alternative Implementation : Creating a class under `Storage Layer` which automatically stores wallet expenses into a csv file. Also this class can specify methods which can include the ability to read imported csv.

Pros: This would reduce the need for the application to run based on user commands. Just like how scripts make certain set up configurations easier, having data stored into meaningful storage methods can be automated rather than the need for a command.

Cons : This would increase the size of the project folder if the number of receipts are huge.

Future Implementation

1. Allow users to import csv files with existing wallet receipts data

5.8. SetBudget

As a `User` one would want to set a budget for expenditures. Here we are strictly looking into the expenses receipts. Based on the sum of expenditures, we compare against the budget set by `User` and offer statistics on how much of the budget has he spent as expenditures. This helps `User` to get notified if he or she has exceeded the set budget. The implementation of this feature is facilitated by `CommandBudget`.

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

Step 1: `Ui Layer` captures the *userInput* and sends it to the `Interpreter Layer` . The `Interpreter Layer` identifies the *CommandType*.

Step 2: Since the *CommandType* is **BUDGET**, `CommandBudget` is instantiated with the *userInput* passed as an argument in the `Executor Layer`

Step 3: *userInput* is parsed by `Parser` in the `Interpreter Layer` :

Step 4: The amount of budget set is returned to `CommandBudget`. Thereafter the `Storage Layer` is accessed to get the total wallet expenses. This is the sum of expenditures due to expenses receipts.

Step 5: The total `Wallet` expenses and budget set are compared to check if `User` has exceeded the budget and depending upon that percentage of budget used up or budget exceeded is calculated.

Step 6: We make an *outputStr* containing the message of whether user is still within budget and percentage statistics of expenditure against budget and send it to the `Interpreter Layer` via an `InfoCapsule` . The `Interpreter Layer` then sends the `InfoCapsule` to the `Ui Layer`

Design Consideration

Aspect: Real-time comparison of budget and expenditure

Alternative Implementation : Creating an accessor command under the `Executor Layer` so that the other adjacent layers get access to the budget. The budget can be constantly kept in comparison with the expenditures by the program.

Pros: Instead of having to set budget to see the percentage statistics on budget against expenditures, the aforementioned implementation reduces user effort.

Cons : This would mean creating additional commands to read and update budget value.

Future Implementation

1. Allow user to be able to see a GUI representation of the percentage exceeded or used up with respect to the budget.

5.9. Listing based on Date

Listing receipts based on date feature

The **DateList** feature is facilitated by `CommandDateList`. It extends from `Command` and overrides execution to output the list of receipts for the specific date inputted by the `User`.

The general flow of logic constitutes the receipts that are being inputted by the `User` tracked by the `ReceiptTracker`. The `User` inputs the command `DateList` followed by the date in the format yyyy-mm-dd and the logic finds all the receipts that contain the date and outputs the list.

Given below is a more detailed explanation.

Step 1: Under the `Executor layer`, the `CommandDateList` is instantiated.

Step 2 : User input is passed into the method `getReceipt` and the program accesses the `storagemanager` under the storage layer

Step 3: Under storagemanager, the method **getReceiptsByDate** returns a **ReceiptTracker** containing all the receipts that contain the input date

Step 4: This data is passed back to the executor layer and an output string is made

Step 5: The relevant information (list of receipts containing date) is passed to the Interpreter layer and then the UI layer for it to be printed out to the user.

Additional information:

1. A boolean function within the executor command class isDateFormat checks if user adheres to the date input format and gives error message for invalid input dates.
2. If the ReceiptTracker returned by the getReceipt method is empty (in other words no such date input exists in the receipt list), program will throw a new DukeException

Design Considerations

Aspect: Input method of users

- Alternative 1: Input datelist without any primary input

Pros: Easier for users to input and get the information

Cons: Users will only be able to get list of receipts for a restricted date range i.e Today only

- Alternative 2 (currently using) : Input datelist with exact date in the correct input format

Pros: Users are able to get the receipts for the exact date specified

Cons: Longer input for users who must adhere to the exact specified date input format

Aspect: The way datelist outputs data

- Alternative 1: Output empty datelist when there are no receipts containing input date

Pros: Additional method in the class that checks for empty list is not required and storagemanager can be called right within the execute() method

Cons: Users might not know that they entered a wrong date input and hence viewed an empty list

- Alternative 2 (currently using) : Throw error when the input date is not found in the list

Pros: Users are able to know if they entered wrongly

Cons: Additional implementations in the class

Further implementation

Datelist lists out the time of the receipt entry in the list as well

5.10. Help

The Help feature of DUKE$$$ is facilitated by the CommandHelp which inherits from parent class command and overrides the execution method to output the description of the commands available to the user.

This feature works in two ways

1. By typing in [help], it outputs all the commands and their descriptions.
2. By typing in [help] <Command_Name>, it outputs the description for that particular command.

The following diagram gives the sequence diagram of the Help feature.

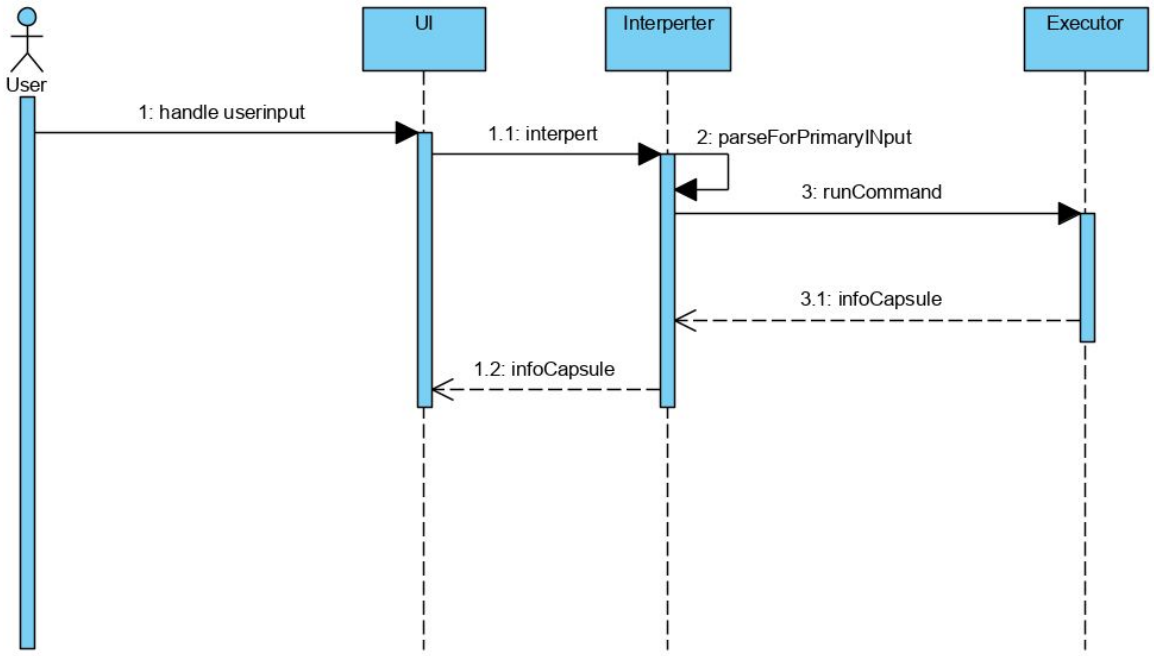

Figure 5.10.1. Sequence Diagram for Help feature

Given below is a more detailed explanation of the feature and a break down of how it works with examples.

Step 1: The `userInput` <help add> is taken in by the GUI.

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

`CommandType` **- HELP**

Step 3: Since the `CommandType` is **HELP**, the program invokes `CommandHelp`.

Step 4: `CommandHelp` is instantiated with `userInput` passed as an argument. The execute method calls the other methods associated with the output of the description.

Firstly, we parse the input through method:

> **Parser.parseForPrimaryInput** -- this will identify the command to which the description is required

Based on the sample userInput specified above,

> ***PrimaryInput*** -- **`add`**

Step 5: Secondly, the ***PrimaryInput*** is passed to the `Executor Layer`. In the `Executor Layer` method `getDescriptionOfSpecificCommand` is invoked. The program runs through all the receipts and takes the command which corresponds to the command detail from the ***PrimaryInput*** . which attains the description through the method `getDescription`.

Step 6: We make an ***outputStr*** containing the description of the command and send it to the `Interpreter Layer` via an `InfoCapsule` . The `Interpreter Layer` then sends the `InfoCapsule` to the `Ui Layer.`

.

Design Consideration

Aspect: Storage of descriptions of commands

Consideration 1: (Currently using) Create an abstract command to call the description by running a loop through all the available commands. Then call the abstract class whenever the description is needed to be output.

Pro: The descriptions are all together with the commands and easily available as a whole chunk and does not require addition of description in multiple places.

Con: More complex backend coding required.

Consideration 2: Create a seperate class to store all the descriptions in one place and access the class to get description.

Pro: Easier to implement

Con: Need to be updated manually and relatively easier to forget while implementing new commands.

Further Implementation

- Able to edit the description through the help function.

5.11. Edit

The Edit feature of DUKE$$$ is facilitated by the CommandEdit which inherits from parent class command and overrides the execution method to change certain parts of the already existing receipts in the list of receipts.

The feature works by taking in the receipt index as the PrimaryFlag and one out of three flags (tag/value/date) to alter the corresponding variable in the receipt stated as the flag suggests.

The editing is done as such,

- To change the tag name, set flag to **`/tag`**
- To change the cash value, set flag to **`/value`**
- To change the date, set flag to **`/date`**

The following diagram is the sequence diagram for Edit feature

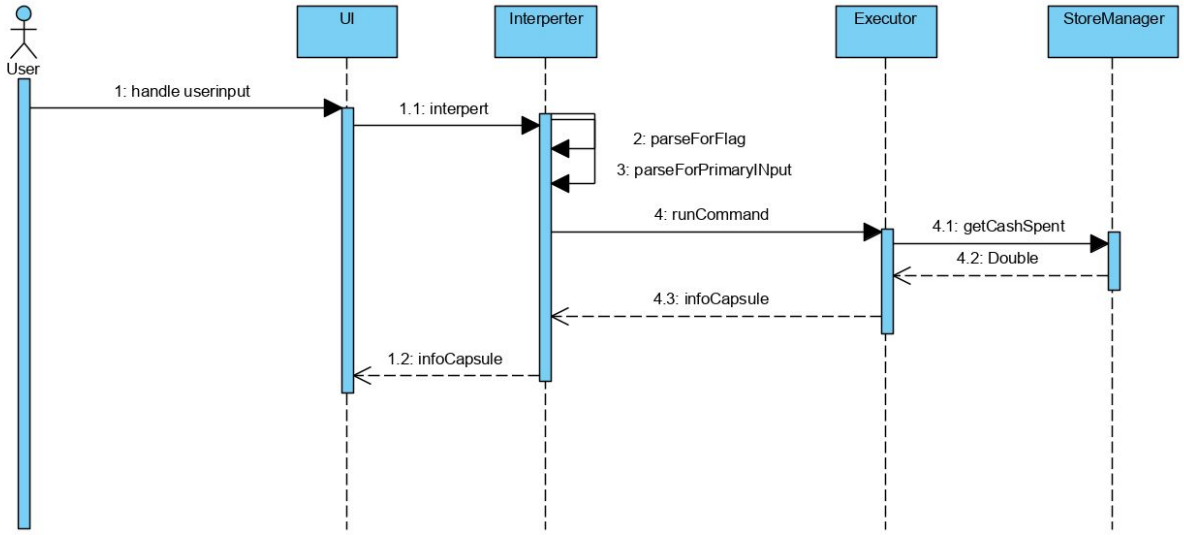

FIgure 5.10.1. Sequence Diagram for Edit Feature

Given below is a more detailed explanation of the feature and a break down of how it works with examples.

Step 1: `User` inputs the command to edit the tag name for the receipt in the following format through the GUI:

`edit 1 /tag food`

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

`CommandType` **- EDIT**

Step 3: Since the `CommandType` is **`EDIT`**, the program invokes `CommandEdit`.

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

Firstly, we parse the input through 2 methods:

Parser.parseForPrimaryInput -- this index of the receipt that needs to be edited

Parser.parseForFlag -- this method is invoked to parse for the flag "tag" in order to obtain the new tag description required .

Based on the sample userInput specified above,

PrimaryInput -- `1` (index)

flag1 details -- `food` (new tag)

Step 5: Secondly, we pass the ***PrimaryInput*** and ***flag1 details*** to the `Executor Layer.` In the `Executor Layer` method `checkAndUpdateFlag` is invoked. The program runs through all the receipts present in the list and makes the changes tag variable according to the details from the ***PrimaryInput*** and ***flag1 details*** . Then `checkAndUpdateFlag` returns this ArrayList by getting details by accessing methods in the `storagemanager` in the `Storage Layer.`

Step 6: We make an ***outputStr*** containing the total amount of expenditure for the month and send it to the `Interpreter Layer` via an `InfoCapsule` . The `Interpreter Layer` then sends the `InfoCapsule` to the `Ui Layer.`

Design Consideration

Aspect: Number of flags that can be edited

Consideration 1: (Currently using) Command only takes in a single flag and changes it.

Pro: Reduces the exceptions produced and number of inheritances used within the code thus leading to simple implementation.

Con: User has to call the function multiple times to change various parts of the same receipt.

Consideration 2: Command able to take in multiple flag variables.

Pro: Able to edit multiple parts of a receipt with a single command.

Con: Highly complex backend implementation.

Further Implementation

- Able to add in new parameters to the receipt.

5.12. Providing Statistics

Command Statistics provides the `User` with statistics for the particular input tag. These statistics include:

1. The percentage of the total wallet expenses spent on the input tag
2. The total expenditure for the tag
3. The list of receipts containing the tag

The **statistics** feature of DUKE$$$ is facilitated by `CommandStatistics` and it helps to do the logic for outputting the percentage, expenditure and the list of receipts.

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

Step 1: `User` inputs the command stats along with the tag for which they want to retrieve the statistics:

`stats transport`

Step 2 : `Ui Layer` captures the `userInput` and sends it to the `Interpreter Layer` . The `Interpreter Layer` identifies its `CommandType`. Based on the sample `userInput` specified above,

`CommandType` -- **`STATS`**

Step 3: Since the `CommandType` is **`STATS`**, the program invokes `CommandStatistics` under the `Executor Layer`

Step 4: `CommandStatistics` is instantiated with the `userInput` passed as an argument. `userInput` is parsed by a method under `Parser` in the `Interpreter Layer` :

`Parser.parseForPrimaryInput` -- this will identify the tag to find

Based on the sample userInput specified above,

```
PrimaryInput -- transport
```

Step 5: Thereafter we pass the `primaryInput` details to `CommandStatistics` under the `Executor Layer`

Step 6: Now in order to get the total expenditure on transport, a method within the command class `getReceipt` is called that takes in the UserInput and `storagemanager` that holds user data as input parameters.

Step 7: Following which `storagemanager` is accessed in the `Storage Layer` and the method `getReceiptsByTag` is called.

Step 8: `getReceiptsByTag` method returns a `ReceiptTracker` containing all the receipts that contain the tag transport.

Step 9: The information is passed back to the method in the command class and this method is invoked within the `Executor Layer` in order to get Total expenses , which is in `ReceiptTracker`, for transport

Step 10: In order to get the total wallet expenses, the method `getWalletExpenses` in the `Storage Layer` is called and this data is passed back to the executor

Step 11: The percentage is calculated using the TotalExpenses as numerator and the WalletExpenses as the denominator

Step 12: An `outputStr` is created containing the percentage value (decimal formatted to two dp) and is sent to the `Interpreter Layer` via an `InfoCapsule` . The `Interpreter Layer` then sends the `InfoCapsule` to the `Ui Layer`

Step 13: In order to get the total expenditure for transport, Steps 6-9 are repeated

Step 14:In order to get the list of receipts that contain the tag, the method `getReceipt` invokes `getPrintableReceipts`.

Step 15: An `outputStr` containing the total expenditure and list of receipts is then sent to the `Interpreter Layer` via an `InfoCapsule` . The `Interpreter Layer` then sends the `InfoCapsule` to the `Ui Layer` to display the information to the user

The diagram below gives a more detailed explanation about the sequence:

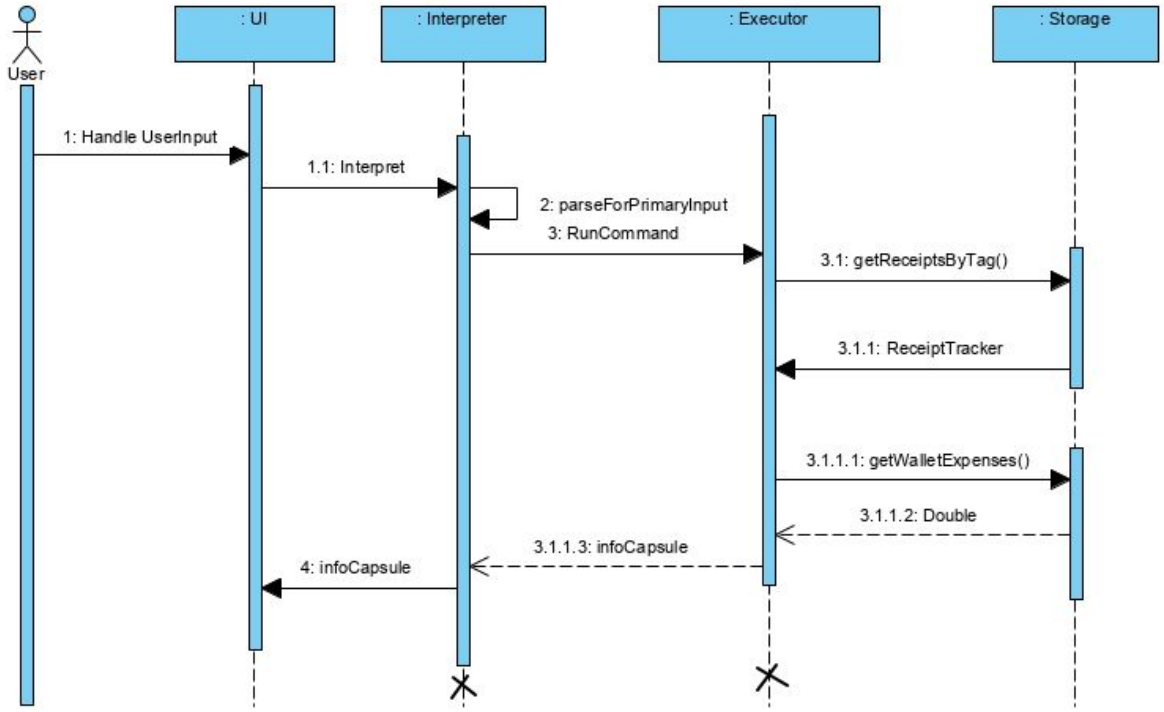

Figure 5.12.1. Sequence Diagram for Command Statistics

Design Consideration

Aspect: The way statistics are shown

Alternative approach : Using a GUI to give the statistics

Pros: More user friendly and user is able to get statistics in the form of graph/pie charts which can improve tracking

Cons: GUI takes more of the computer system resources than the CLI

Aspect: Percentage output

Alternative: Use integer data type across methods that return numerical values in the program

Pros: When using a double data type across methods such as getWalletExpenses and getTotalExpenses, if both values are 0.0 and are divided, the result is a math error and NaN% is displayed which might confuse some users and an integer data type will not cause this error

Cons: Since the program is dealing with cash spent and earned, it is essential to involve decimal places and hence double data types across the methods is still preferable for better precision.

Future Implementation

These are features considered for future implementation:

1. Percentage of tag income out of wallet income
2. Total income gained from a particular tag
3. List of all income receipts only
4. Nett income gained in a month

5.13. Displaying Major Expenses

The majorexpense feature of DUKE$$$ is facilitated by the `CommandMajorExpense` which inherits from parent class `Command` and overrides the execution method to output the list of receipts that have cash spent property more than or equal to that of the integer input by the user.

This feature works in two ways

1. By taking in an empty primary input , in other words just the command itself, in order to give a list of receipts with cash spent property above/equal to $100
2. By taking in a primary integer input in order to give a list of receipts with cash spent property above/equal to that of the integer input by the `User`

Given below is a more detailed explanation of the feature and a break down of how it works with examples.

Step 1: `UI Layer` captures the string userinput in the following format

```
majorexpense 50
```

Step 2: This information is then sent to the parser under the `interpreter layer` to parse for the command type in this case based on the user input above,

```
CommandType - majorexpense
```

Step 3: As such, the program invokes `CommandMajorExpense` which is instantiated and the user input is passed in as argument under the `executor layer`

Step 4: The parser parses for the primary input in this case 50 and the string input is passed into the method `getMajorExpense` in storageManager under the `storage layer.`

Step 5: This method `getMajorExpense` returns a `ReceiptTracker` , an arraylist that contains all the receipts with cash spent property more than or equal to $50

Step 6: The executor sends this list to the interpreter and the UI through the InfoCapsule to print out the list.

Additional information:

In the event that no primary input is detected by the Parser in Step 4, the method `getMajorReceipt` under the `storagemanager` is used to return a `ReceiptTracker` , an arraylist containing all the receipts with cash spent property more than or equal to $100.

Design considerations

Aspect: User input

Alternative 1: Split the commands into two separate ones, one for printing out list of receipts with cash spent property above/equal to $100 and the other for printing out receipts with cash spent property above/equal to user input

Cons: User has to remember two different command names and is more time consuming to implement

Alternative 2 (Current implementation) : Both functionalities in one command

Pros: Easier to implement and user has to remember one command only

Future implementations

These are features considered for future implementation:

1. Command can take in double values which allows for more precision

5.14. Tracking an Expense By Tag

5.14.1. Input Format

To track/untrack a tag, the user must input the following command:

<track tag>

Tag refers to the tag the user wishes to track.

5.14.2. Sequence of Events

The following will detail the sequence of events during the two phases of tracking an expense by a tag.

Editing the Data

Upon receiving instructions from the `Interpreter Layer`, the `Executor` will create `CommandTrackTag` and call it's **execute()** method. `CommandTrackTag` will then request the `Storage Layer` to track a particular tag and if no errors are thrown, it will update its own internal `InfoCapsule` accordingly.

The `StorageManager` will receive this request and adjust the `Wallet`'s `ReceiptTracker` to start tracking this particular tag. `Executor` will return `CommandTrackTag's` `InfoCapsule` back to the `Interpreter Layer`, where it will be sent back to the `Ui Layer`.

The following Sequence Diagram illustrates this chain of events:

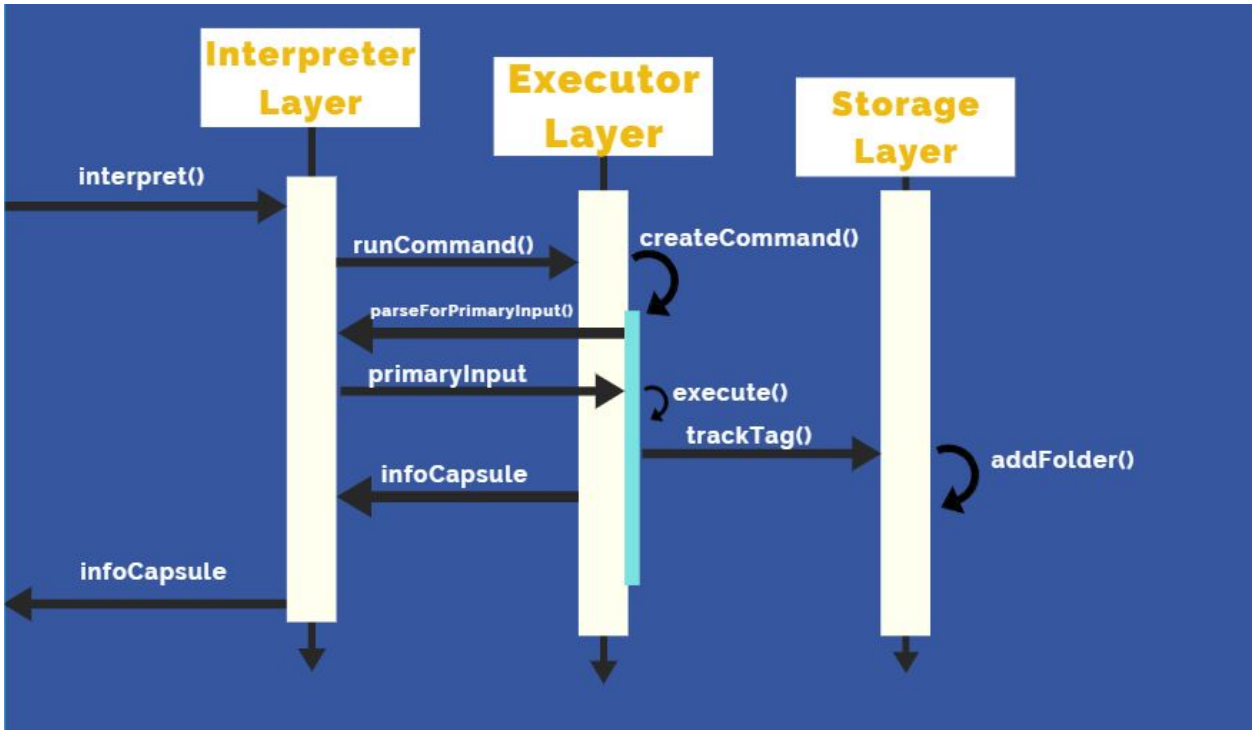

Figure 5.14.2.1 Sequence Diagram for CommandTrack

Displaying the Data

Upon receiving the `InfoCapsule`, the `Interpreter Layer` will forward it to the `Ui Layer` who will then unpack its contents. Then, the `Ui Layer` will seek to update its bar chart by requesting the data from the `Interpreter Layer` who in turn will ask the `Executor Layer` to access this data.

To accomplish this, the `Executor Layer` will create an `AccessWallet` Object and and call it's **execute()** method. `AccessWallet` will request the `Storage Layer` to return its `Wallet` Object which it will store a reference of in its `InfoCapsule` and this `InfoCapsule` will bubble back to the `Ui Layer` who can then update its bar chart accordingly.

The Sequence Diagram below illustrates the chain of events during the data display phase:

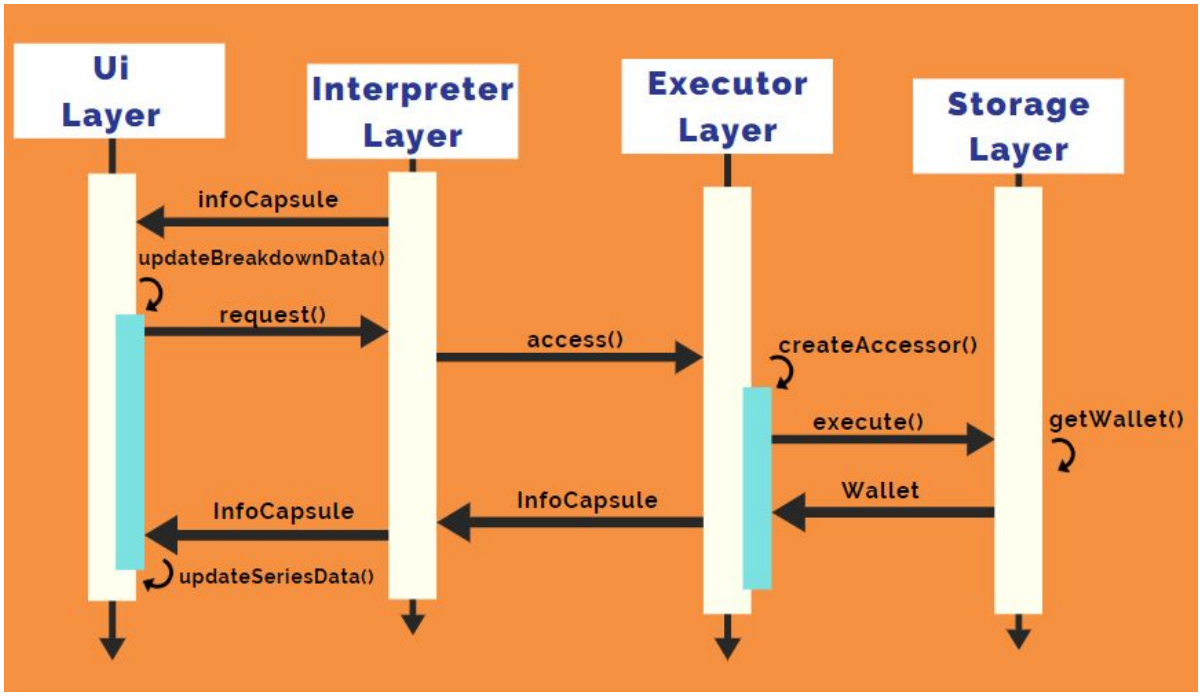

Figure 5.14.2.2 Sequence Diagram for CommandTrack

5.14.3. Design Considerations

This segment will cover the design considerations that were taken before implementing this feature.

Initial Design

The initial idea was to have the `Ui Layer` hold a direct reference to the same `Wallet` as the one in the `Storage Layer`. The pros and cons of this design is as follows:

> **Pros**: Easy to implement as there is no need to traverse layers
>
> **Cons**: High coupling and low scalability as two very separate layers are tightly interlinked

Alternative Design

The alternative is to create the `Accessor` Class which facilitates the access of data in the `Storage Layer` by bubbling a new utility class, `InfoCapsule`, through the layers. The pros and cons of this design is as follows:

Pros: Difficult to implement

Cons: Reduces coupling and increases scalability as layers exist more modularly and separate from each other.

Decision

The team decided to go with the alternative design as we prioritized scalability over ease of implementation as this will be more beneficial in the long term.

5.14.4. Future Implementations

In the future, we would probably want to decouple the layers even further by creating a method to return a `HashMap` containing all the required data instead of the `Wallet` Object instead.

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$$$.

Use case 1: Adding a Spending Receipt

- **MSS:**
    1. User inputs `OUT` Command with necessary arguments.
    2. Duke$$$ adds a spending receipt to the list of receipts.
    3. Use case ends.
- **Extensions**
    - 1a. Duke$$$ 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 for Day

- **MSS:**
    1. User inputs `expendedday` Command with necessary arguments.
    2. Duke$$$ takes in the argument and displays the total amount of expenditure for the given day in the input
    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 for Week

- **MSS:**
    4. User inputs `expendedweek` Command with no arguments.
    5. Duke$$$ displays the total amount of expenditure for the current week and the number of days till the end of the week. Takes Monday to be the start of any week.
    6. Use case ends.

Use case 6: Display expenditure for Month

- **MSS:**
    7. User inputs `expendedmonth` Command with necessary arguments.
    8. Duke$$$ takes in the argument and displays the total amount of expenditure for the given month in the input
    9. 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 7: Display expenditure for Year

- **MSS:**
    10. User inputs `expendedYear` Command with necessary arguments.
    11. Duke$$$ takes in the argument and displays the total amount of expenditure for the given Year in the input
    12. 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 8: Display list of receipts by date

- **MSS:**
    1. User inputs datelist Command with necessary arguments.
    2. Duke$$$ takes in the argument and displays the list of receipts based on date input
    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 9: Adding an Income Receipt

- **MSS:**
    1. User inputs `IN` Command with necessary arguments.
    2. Duke$$$ adds an income receipt to the list of receipts.
    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 10: View help section

- **MSS:**
    13. User inputs `help` Command with necessary arguments.
    14. Duke$$$ displays the help section content.
    15. Use case ends.

- **Extensions**
    - 1a. Duke$$$ detects invalid arguments in the given user input.
        - 1b.1. Duke$$$ outputs an error message.
        - Use case ends.

Use case 11: Edit receipt

- **MSS:**

16. User inputs `edit` Command with necessary arguments.
17. Duke$$$ takes in the argument and changes the part of the receipt according to the flag used.
18. 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 12: Export WalletData into csv

- **MSS:**
    4. User inputs `export` Command
    5. Duke$$$ analyses the user input and exports wallet data into csv.
    6. Use case ends.

- **Extensions**
    - 1a. Duke$$$ detects unnecessary parameters along with `export` command.
        - 1a.1. Duke$$$ outputs an error message
        - Use case ends.
    - 1b. Duke$$$ detects incorrect spelling of `export` like “expot”
        - 1b.1. Duke$$$ outputs an error message.
        - Use case ends.

Use case 13: List major expense receipts

- **MSS:**
    7. User inputs `majorexpense` Command with necessary arguments
    8. Duke$$$ takes in the argument and lists out receipts based on the integer input
    9. 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 list of receipts that have cash spent property above/equal to $100.
        - Use case ends.

Use case 14: Budget

- **MSS:**
    10. User inputs `budget` Command
    11. Duke$$$ analyses the arguments and sets budget and returns percentage statistics.
    12. 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 15: Display statistics for a tag

- **MSS:**
    4. User inputs stats Command with necessary arguments.
    5. Duke$$$ takes in the argument and displays the percentage of total expenses on tag, total expenditure on tag and the list of receipts that contain the tag
    6. 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

Abstraction

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

Agile Design

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

Contractor

Refers to the authority that engaged the team in creating this product (i.e. NUS CS2113T Professors)

Duke$$$

Financial Tracker

Layers

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

Mainstream OS

Windows, Linux, Unix, OS-X

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

Users

Users refer to individuals using Duke$$$

Appendix F. Instruction for Manual Testing

This section gives the instructions to manually test DUKE$$$.

1. Downloading and beginning the application

Step 1: Download the latest released .jar file of DUKE$$$ through here.
Step 2: Place the .jar file in an empty folder.
Step 3: Click the file or open the command prompt and launch the application using the startup command:

```
java -jar duke.0.1.5.jar
```

The home screen of DUKE$$$ should appear.

Step 4: Type in "EnableTest" to enter manual testing mode. Duke$$$ should populate itself with a list of dummy data for your testing purposes.

2. Addition of receipts

2.1. Addition of income receipts

Test case 1: `IN $5.00 /date 2019-08-25 /tags bank`
Expected: A new income receipt of amount $5.00, date 2019-08-25, tagged bank is added into the wallet.
Message: `Added Income Receipt: $5.00 with tags: bank`

Test case 2: `IN $5 /tags bank lunch`
Expected: A new income receipt of amount $5.00, tagged bank and lunch is added into the wallet. The date is automatically generated as the current date.
Message: `Added Income Receipt: $5.00 with tags: bank, lunch`

Test case 3: `IN $5.00 /date 2019-08-25`
Expected: A new income receipt of amount $5.00, date 2019-08-25 is added into the wallet. No tags are added.
Message: `Added Income Receipt: $5.00 with tags:`

Test case 4: `IN 0 /date 2019-08-25 /tags bank`
Expected: No new income receipt is added due to no cash input value. An error
message is displayed as toast.
Message: `Cash value cannot be $0.`

2.2. Addition of Spending receipts

Test case 1: `OUT $10.00 /date 2019-12-25 /tags gift`
Expected: A new spending receipt of amount $10.00, date 2019-12-25, tagged gift is added into the wallet.
Message: `Added Spending Receipt: $10.00 with tags: gift`

Test case 2: `OUT $10 /tags gift card`
Expected: A new spending receipt of amount $10.00, tagged gift, card is added into the wallet. The date is automatically generated as the current date.
Message: `Added Spending Receipt: $10.00 with tags: gift, card`

Test case 3: `OUT $10.00 /date 2019-12-25`
Expected: A new spending receipt of amount $10.00, date 2019-12-25 is added Into the wallet. No tags are added.
Message: `Added Spending Receipt: $10.00 with tags:`

Test case 4: `OUT 0 /date 2019-12-25 /tags gift`
Expected: No new spending receipt is added due to no cash input value. An error message is displayed as toast.
Message: `Cash value cannot be $0.`

3. Deletion of receipts

Test case 1: `DELETERECEIPT 3`
Expected: The receipt at index 3 is deleted.
Message: `Receipt 3 has been deleted.`

Test case 2: `DELETERECEIPT`
Expected: No receipts are deleted due to violation of style. An error message is
Displayed.
Message: `Invalid index input. Please enter an integer.`

4. Balance display

4.1. Get total balance

Test case 1: `BALANCE`
Expected: The current balance is displayed.
Message: `Total Balance: $50.00.`

Test case 2: `BALANCE 37fhsd`
Expected: The second string is parsed and the current balance is displayed.
Message: `Total Balance: $50.00.`

4.2. Get total expenditure

Test case 1: `EXPENSES`
Expected: The current total expenditure is displayed.
Message: `Total Expenditure : $50.00.`

4.3. Get expenditure for a day

Test case 1: `EXPENDEDDAY 2019-03-21`
Expected: The total expenditure for 2019-03-21 is displayed.
Message: `The total amount of money spent on 2019-03-21 is $5.00.`

Test case 2: `EXPENDEDDAY 2300-12-21`
Expected: The total expenditure for the current date is displayed.
Message: `The total amount of money spent on 2019-11-11 is`

`$5.00.`
`NOTE: The date input is in the future.` (Based on 2019-11-11)

4.4. Get expenditure for the current week

Test case 1: `EXPENDEDWEEK`
Expected: Displays the total expenditure for the current week.
Message: `The total amount spent this week is $50.00 and there is/are 3 day(s) to the end of week.`

4.5. Get expenditure for a month

Test case 1: `EXPENDEDMONTH JULY /year 2019`
Expected: The total expenditure between 2019-07-01 and 2019-07-31 is displayed.
Message: `The total amount of money spent in JULY 2019 : $50.`

Test case 2: `EXPENDEDMONTH NOVEMBER /year 2019`
Expected: A toast message indicating violation of style is displayed.
Message: `The total amount of money spent in NOVEMBER 2019 : $50. Number of day(s) left in this month is/are 19.` (Based on 2019-11-11)

Test case 3: `EXPENDEDMONTH /year 2019`
Expected: A toast message indicating violation of style is displayed.
Message: `Wrong format! FORMAT : expendedmonth <month> /year <year>`

4.6. Get expenditure for a year

Test case 1: `EXPENDEDYEAR 2019`
Expected: The total expenditure for the year 2019 is displayed.

Test case 2: `EXPENDEDYEAR k2fj`

Expected: A toast message indicating violation of style is displayed.
Message: `Year input is either a double or contains String values.`

Test case 3: `EXPENDEDYEAR 200`
Expected: A toast message indicating violation of style is displayed.
Message: `Year input contains lesser/extra number of variables.`

5. Listing receipts

5.1. List receipts by index

Test case 1: `LIST`
Expected: All saved tasks and receipts are listed.

5.2. List receipts by date

Test case 1: `DATELIST 2019-01-27`
Expected: Displays all receipts dated 2019-01-27.

Test case 2: `DATELIST JULY`
Expected: A toast message indicating violation of style is displayed.
Message: `Invalid date input. FORMAT : datelist yyyy-mm-dddd`

6. Tracking receipts

6.1. Track receipts by tag

Test case 1: `TRACK breakfast`
Expected: All receipts with 'breakfast' as a tag are shown in the statistics.
Message: `Tracking tags: breakfast`

Test case 2: `TRACK`

Expected: A toast message indicating violation of style is displayed.
Message: `Please enter a tag to track.`

Test case 3: `TRACK breakfast` (Again)
Expected: Displays a toast message indicating pre-existing tag.
Message: `Category already exists!`
`If you wish to untrack this tag, try UNTRACK <tag>.`

Test case 4: `TRACK HOME` (non-existent tag)
Expected: Adds the tag to the list of tracking tags.
Message: `Tracking tags: HOME`

6.2. Untrack receipts by tag

Test case 1: `UNTRACK breakfast`
Expected: Untracks all receipts with the tag 'breakfast'.

Test case 2: `UNTRACK`
Expected: A toast message indicating violation of style is displayed.
Message: `Please enter a tag to untrack.`

7. Checking statistics

Test case 1: `stats breakfast`
Expected: Gives the percentage of expenditure spent on breakfast.
Message: `35.00% of your wallet expenses is spent on Breakfast.`
`You spent a total of $35.00 on breakfast`
`1. [expenditure, breakfast] $35.00 2019-11-11`

Test case 2: stats
Expected: A toast message indicating violation of style is displayed.
Message: `Tag input is missing. FORMAT : stats <tag>`

8. Currency Conversion

Test case 1: `CONVERT 5.00 /from SGD /to INR`
Expected: The converted amount in INR is displayed.
Message: `DUKE has converted SGD 5.00 to INR 262.86`
`Exchange rate used: 78.9905` (Based on 2019-11-11)

Test case 2: `CONVERT hello /from KRW /to USD`
Expected: A toast message indicating violation of style is displayed.
Message: `Please enter a valid amount.`

Test case 3: `CONVERT USD JPY`
Expected: A toast message indicating violation of style is displayed.
Message: `DUKE$$$ could not understand the input.`
`Please follow the following format to convert :`
`For example : convert <amount> /from USD /to SGD`

Test case 4: `CONVERT 5.00 /from SGD /to USD (`No internet available)
Expected: A toast message indicating error is displayed.
Message: `Exchange rate data is unavailable.`
`1. Please ensure you have active internet access.`
`2. Also, please follow the correct format for currency conversion available under CONVERT if you type help on the CLI.`
`3. Please ensure that you enter proper ISO 4217 Country codes`

9. Checking the weather

Test case 1: `WEATHER /until now`
Expected: The weather details for today is displayed.
Message: `DUKE$$$ has predicted the following weather forecast :`
`Forecast date : 2019-11-11`
`Minimum Temperature in Degrees Celcius : 26.64`
`Maximum Temperature in Degrees Celcius : 30.93`
`Average Temperature in Degrees Celcius : 19.4`
`State of Weather : Heavy rain`

Test case 2: `WEATHER /until 3`
Expected: A message indicating an invalid input is displayed.
Message: `Please enter in either of the following format:`

1. `Weather /until now`
2. `Weather /until later`
3. `Weather /until tomorrow`

10. Basic arithmetic

10.1. Addition

Test case 1: `ADD 15 / 17`
Expected: The added value for 15 + 17 is shown.
Message: `15 + 17 = 33`

Test case 2: `ADD 15 / nine`
Expected: A toast message indicating violation of style is displayed.
Message: `Invalid input please enter the second number. Format : add <num1> / <num2>`

Test case 3: `ADD 15 % 6`
Expected: A toast message indicating violation of style is displayed.
Message: `Enter forward slash and second number. Format : add <num1> / <num2>`

10.2. Subtraction

Test case 1: `SUB 30 / 12`
Expected: The value for 30 - 12 is shown.
Message: `30 - 12 = 18`

Test case 2: `SUB 15 / nine`
Expected: A toast message indicating violation of style is displayed.

Message: `Invalid input please enter the second number. Format : sub <num1> / <num2>`

Test case 3: `SUB 15 % 6`
Expected: A toast message indicating violation of style is displayed.
Message: `Enter forward slash and second number. Format : sub <num1> / <num2>`

10.3. Multiplication

Test case 1: `MUL 5 / 12`
Expected: The value for 5 x 12 is shown.
Message: `5 * 12 = 60`

Test case 2: `MUL 15 / nine`
Expected: A toast message indicating violation of style is displayed.
Message: `Invalid input please enter the second number. Format : mul <num1> / <num2>`

Test case 3: `MUL 15 % 6`
Expected: A toast message indicating violation of style is displayed.
Message: `Enter forward slash and second number. Format : mul <num1> / <num2>`

10.4. Division

Test case 1: `DIV 30 // 5`
Expected: The value for 30 divided by 5 is shown.
Message: `30 / 5 = 6`

Test case 2: `DIV 15 // nine`
Expected: A toast message indicating violation of style is displayed.
Message: `Invalid input please enter the second number. Format`

`: div <num1> // <num2>`

Test case 3: `DIV 15 / 6`
Expected: A toast message indicating violation of style is displayed.
Message: **`Enter forward slash and second number. Format : div <num1> // <num2>`**

11. Help

Test case 1: `HELP BYE`
Expected: The description for command 'bye' is displayed.
Message: **`BYE - Exits the program`**

Test case 2: `HELP BYEIANCINCIC`
Expected: A toast message indicating violation of style is displayed.
Message: **`Command invalid. Enter 'help' to see all the available commands`**

12. Edit

Test case 1: `EDIT`
Expected: A toast message indicating violation of style is displayed.
Message: **`Index has to be an INTEGER`**
`FORMAT : edit <index> /<part to be edited> <new-input>`