Project MooMooMoney

Contents

Setting up

Prerequisites

1. Java 11
2. IntelliJ

Setting up the project in your computer

1. Open IntelliJ (if you are not in the welcome screen, click File > Close Project to close the existing project dialog first).
2. Set up the correct JDK version.
3. Click **Configure** > **Structure** for new Projects (in older versions of Intellij:Configure > Project Defaults > Project Structure).
4. If JDK 11 is listed in the drop down, select it. If it is not, click **New...** and select the directory where you installed JDK 11.
5. Click **OK**.
6. Click **Import Project.**
7. Locate the project directory and click **OK**.
8. Click on the **build.gradle** file to import the gradle project.

Running the project

As the project uses gradle, you can use gradle to run the commands.

1. On the top right corner, click on **Add Configuration...** to add a gradle configuration by clicking on the **+** on the top left > **Gradle** > Choose the default value as the **gradle project** > **OK**
2. On the very right on the taskbar, there should be a Gradle tab. Click on it > Click on the elephant icon > type in any gradle command to run it.
3. **gradle run** will execute the project in the terminal of IntelliJ, **gradle test** will run the JUnit tests that are written and **gradle checkstyleMain checkstyleTest** will run checkstyle tests (configuration can be found in the folder **config**>**checkstyle**>**checkstyle.xml**).

Setting up Continuous Integration (CI)

This project uses 3 CI services: Travis, Appveyor and Azure Pipelines. Each of the steps can be found in the corresponding **.yml** files in the root of the project.

1. To activate the service for your own repo, you should login to the respective services web applications using your Github account for easier access.
2. Activate the corresponding repository using their web applications.

3. Create and push a commit that contains the corresponding **.yml** configuration files for the 3 services.
4. The build should start running based on the configuration file provided.

Having multiple CI services can be useful as different services uses different default operating systems and as such, being able to run the tests on different operating systems may allow you to catch certain bugs. Also, there may be times where one service may encounter an issue while the other services may run fine. To that end, it is necessary to check the logs of the respective service to find out what the issue is.

Documentation

Here is a guide to explain our markups.

1. Bolded words indicate a java class, object, variable or method. (e.g **budget**)
2. Italic words indicate a command that the user can key in. (e.g *moo*)

Architecture

1.1 Overall

MooMooLauncher is responsible for launching the application and is able to allow for launching into the Graphical User Interface (GUI) if it is implemented. It can either call **MooMoo** directly to launch the Command Line Interface (CLI) or use JavaFX's **Application** to launch a GUI written in JavaFX (**Main**) coming in v2.0. Figure 1.1.2 shows the overall architecture diagram of the application

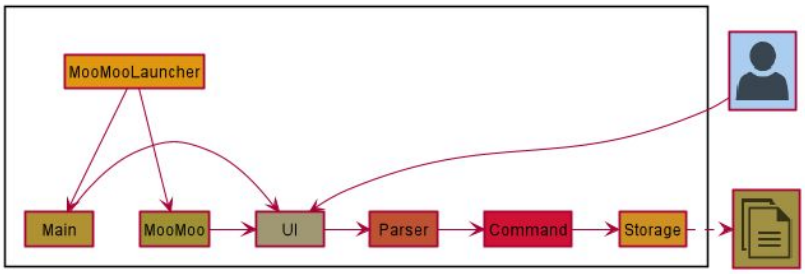

Figure 1.1.1 Architecture Diagram

There are 4 other components:

- UI: The user interface of the application.
- Storage: Reads and writes data to the hard disk
- Parser: Parses the user input into appropriate commands.
- Command: The command executor.

Figure 1.1.2 shows the class diagram for the main parts of the application

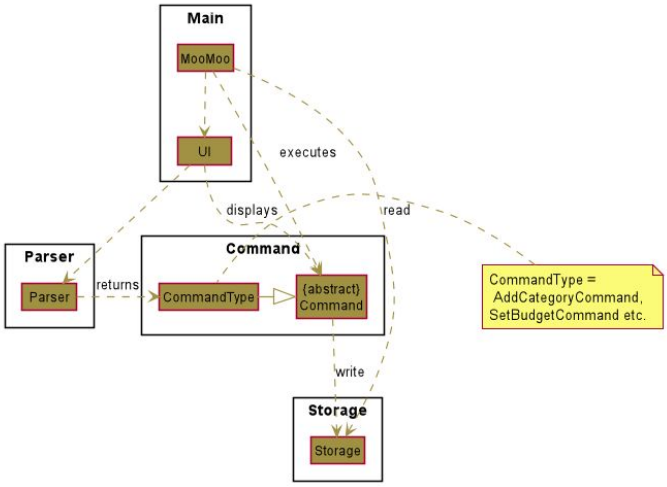

Figure 1.1.2. Class Diagram of the Application

1.2 UI Component

UI is responsible for displaying the main interface to the user as well as receive input and output to the user. As MooMooMoney is currently a CLI application, it uses the **Scanner()** class to read input from the user and **System.out.println()** to output to the user.

1. When the user first enters the program, a start up message will be displayed alongside a cursor waiting for the user's input via **Scanner.nextLine()**.
2. When the first input is given (when it ends in a space), **UI** will pass the input to **Parser.**
3. When a **command** is executed, the **command** will set **output** in UI, which will then be displayed after the command returns.
4. The entire **UI** will be in an infinite loop, allowing the user to keep entering input until the user enters the **bye** command.

1.3 Parser Component

Parser is responsible for parsing any input given by the user and then sending the parsed input to the respective command classes.

1. When the first input is received by **UI**, the first word that ends with a space will first be parsed by **parse()**. This will be the main command to be ran.
2. **Parser** will then continue to read in input via **Scanner()** delimited by spaces to get the next arguments. For example, if the first word is **budget**, it will then read the next arguments like **list c/food**.
3. When all the values are parsed, **Parser** will then put it into an appropriate data structure for the corresponding **command** class to read and execute the function accordingly.

1.4 Command Component

Command is an abstract class for the various commands like **SetBudgetCommand** and **AddCategoryCommand**. It is responsible for executing the respective functions based on the input parsed from the user.

1. Based on the input parsed by **Parser**, the corresponding Command subclass will be initialized. For example, if the user types **budget list** the **ListBudgetCommand** subclass will be initialized.
2. The corresponding **Command** subclass will then be initialized with the created data structures in the **Parser** class. For example, HashMap<String> to hold the categories in the **SetBudgetCommand** class. The **execute()** function in the **Command** subclass will then be called to take the corresponding action.

1.5 Storage Component

Storage is responsible for reading and writing data as necessary, for example, saving the budget to a file and reloading the data into the application upon launching.

1. The **Storage** class will always be initialized at the start with the filepath of the files to load in any saved values found in the corresponding text files (**budget.txt, schedule.txt, category.txt and expenditure.txt)** in the **data** folder. If there are no files or directories found, default files from the JAR file will be placed into a created **data** folder.
2. When a **Command** that is taken requires saving to storage such as adding a category or setting a budget, the corresponding method will be called. For example, **saveBudgetToFile()** will be called when a new budget has been set or changed.

Budget Feature

1.1 Proposed Implementation

The budget feature allows a user to manage a budget based on the categories created. A **Budget** class contains a **HashMap<String, Double>** that stores the categories as the key and the corresponding budget as its value. This allows for a user to set a different budget for every category.

When a *budget* command is given, a **BudgetCommand** class that extends the **Command** class will be called. **BudgetCommand** will use an abstract **execute()** method from the **Command** class. Depending on the input of the user, the command to be run will be different. The command will be in this format: *budget <set/edit/list/savings> <arguments>*

The **BudgetCommand** class will depend on the **CategoryList** class to allow for **BudgetCommand** to gain access to the added categories for verification. It would also depend on the same class to

get the total transactions spent per month to allow for the calculation of any savings accumulated using **budget - total expenditure**.

Upon any changes to the **Budget** object (set or changing it), the data in the **HashMap** will be stored in a text file containing the categories and its corresponding budget. Figure 1.1.1 shows the class diagram for **BudgetCommand**

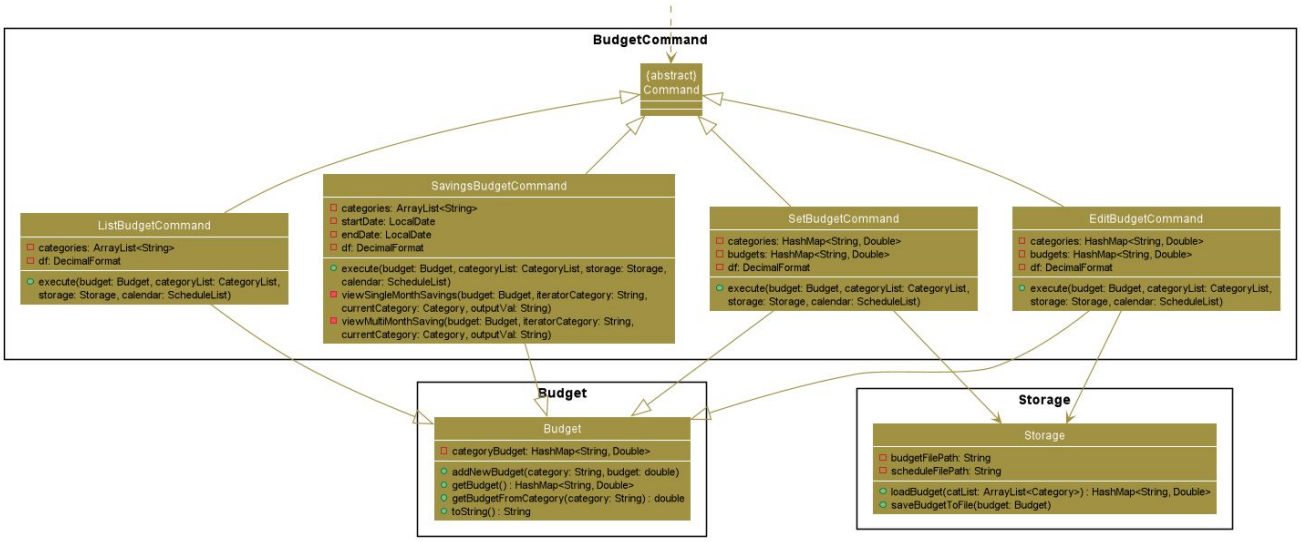

Figure 1.1.1 Class Diagram for **BudgetCommand**

Here is an example of the various scenarios using the budget class

1. When the user launches the application for the first time, the **HashMap** in the **Budget** object will be initialized into an empty **HashMap,** allowing values to be inserted. If the user has previously made changes to the budget with values stored on the hard disk, those values will be fed into a **HashMap** which the **Budget** object will initialize to.

2. The user will then insert an input *budget set c/food b/1000***.** Since the ***budget*** command is given, the **BudgetCommand** class will be returned. The **execute()** method would then read in the ***set*** input and then proceed to read the **category** after *c/* and **budget** after *b/***.** It will then create a new entry in the **HashMap** in the initialized **Budget** object with the **category** as key and **budget** as value. An error will be shown if the category is not found in **CategoryList**. The user can then retrieve the value by using the **category** as the key. Since a change has been done to budget, the data will be stored into the hard disk.

3. After the budget is set, a user can also edit the current budget by using **budget edit** with the same arguments as **budget set**. Due to the use of HashMaps, the budget value will automatically be updated if **HashMap.put()** is used due to the property of a HashMap having unique keys. The user will be prompted if the budget has not been set or the category does not exist. Figure 1.1.2 shows the activity diagram for **budget set/edit**.

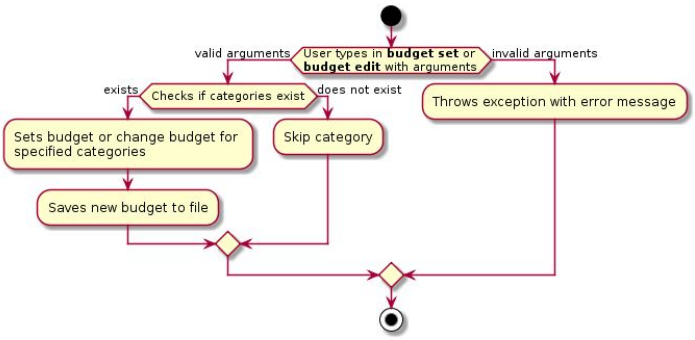

Figure 1.1.2 Activity diagram for **budget set/edit**

4. The user can view the budgets set using **budget list**. The command will loop through the HashMap and print out the key, value pairs or show a static message if no budgets are set.

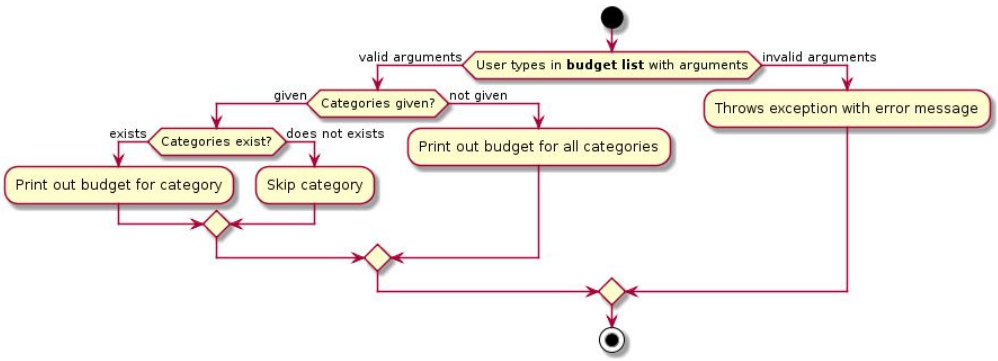

Figure 1.1.3 Activity diagram for **budget list**

5. Lastly, the savings in a month as well as total savings over a period per category can also be shown to the user. An example input will be *budget savings c/food s/01/2019 e/05/2019*. Similarly, the **category** used will be after *c/*. The start month/year and end month/year is given after *s/* and *e/* respectively. The **BudgetCommand** class will then parse the string values into appropriate **LocalDate** objects. However, as **LocalDate** objects require a "date", we can achieve this by adding a static "date" portion to the front of the string. Figure 1.1.4 shows the code for parsing an input date from string to **LocalDate**

```
String fullDate = "01/" + inputDate;
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");
return LocalDate.parse(fullDate, formatter);
```

Figure 1.1.4 Code for parsing input date

You can then get the month and year using **getMonth()** and **getYear()**.

6. We will then use the **getTotal()** method in the **Category** class when looping through the category objects to get the total expenditure in that categories for expenditures that are within that particular month and year.

7. There are several factors to consider when creating this method due to the different inputs that can be given.
    a. If no end date is given i.e only view the savings for one month.
    b. If no categories is given i.e view the savings for all categories.

8. Another issue to consider was the complexity of the loop if the start year and end year are different. As far as I can tell, there are no currently available APIs that allow us to get all the months between two different dates. As such, there was a need to craft one ourselves.

9. To achieve this, I first checked if the year values are different first. Figure 1.1.5 shows the code for parsing a range of months and years.

```
int numberOfMonths = 0;
int numberOfYears = end.getYear() - start.getYear();
double totalExpenditure = 0;

if (numberOfYears > 0) {
    int startMonthValue = start.getMonthValue();
    int endMonthValue = 12;
    for (int currentYear = start.getYear(); currentYear <= end.getYear(); ++currentYear) {
        for (int currentMonth = startMonthValue; currentMonth <= endMonthValue; ++currentMonth) {
            ++numberOfMonths;
            totalExpenditure += currentCategory.getCategoryTotalPerMonthYear(currentMonth, currentYear);

        }
        startMonthValue = 1;
        if (currentYear == end.getYear() - 1) {
            endMonthValue = end.getMonthValue();
        }
    }
```

Figure 1.1.5 Code for parsing month/year range

After that, I would then loop through the months in the first year, by setting **endMonthValue** to 12 (December). After the first loop, I would then reset the **startMonthValue** to be 1 (January) and check if the next year loop would be the last year. If it is, the **endMonthValue** to be set to the given **endMonth**. Otherwise, it will just loop through from January (1) to December (12). Figure 1.1.6 shows the activity diagram for **budget savings**

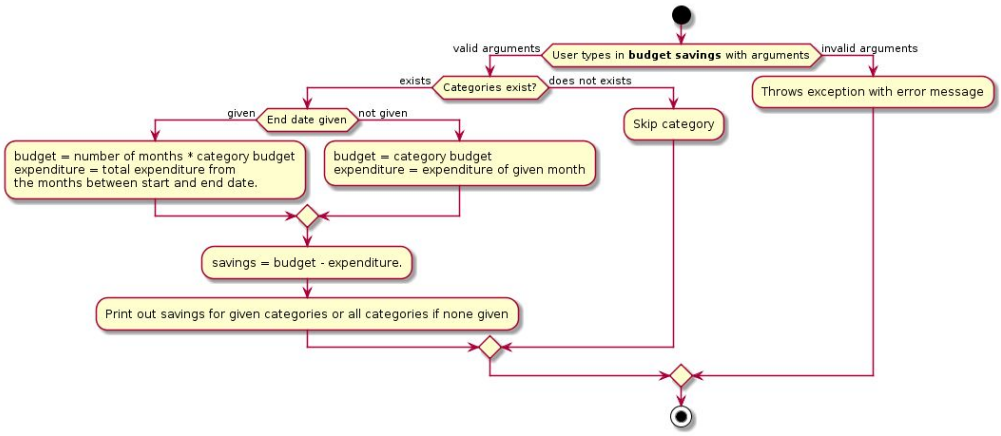

Figure 1.1.6 Activity diagram for **budget savings**

10. This is the sequence diagram for **SetBudgetCommand**. **EditBudgetCommand** will work in the same way, while ListBudgetCommand and **ViewSavingsCommand** will not touch the **Storage** class. **ViewSavingsCommand** will also call other methods such as **getTotal()** from **:Category** to get the total expenditure in that category. Figure 1.1.7 shows the sequence diagram for **budget set**

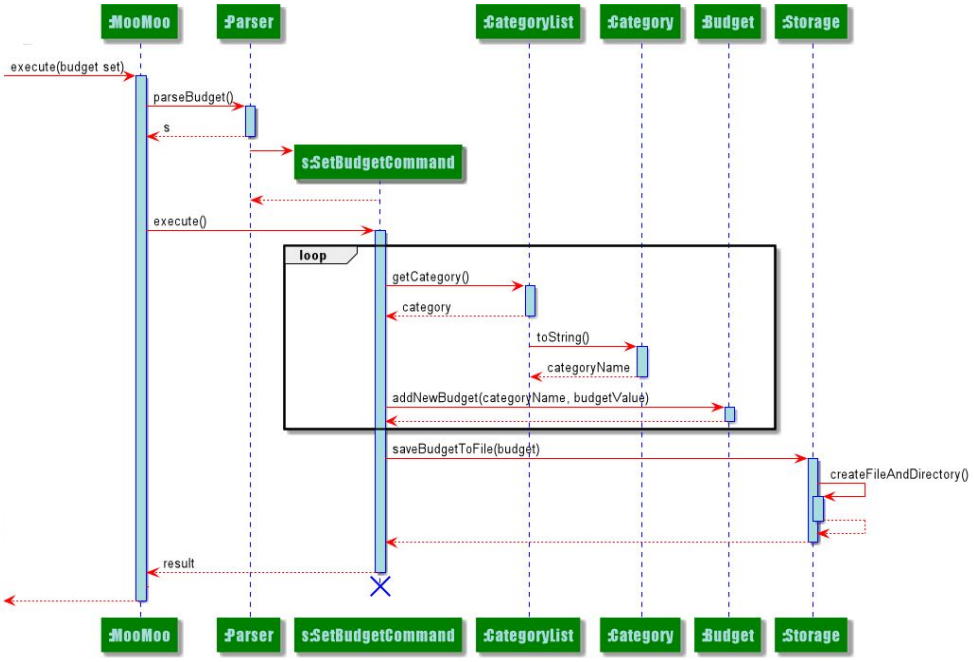

Figure 1.1.7 Sequence Diagram for **budget set**

1.2 Alternatives

Data structure to store budget

- Alternative 1 (Current Choice): Store in a HashMap
    - Pros: Allows user to set different budget for different categories.
    - Cons: May have performance issues if there are a lot of categories as we would need to iterate through the Map several times.
- Alternative 2: Store as a double
    - Pros: Fast performance as there is only 1 value to reference.
    - Cons: Only allow for storing of one budget for all the categories.
- Alternative 1 was chosen as having different budgets for different categories would be useful and an average user would likely not have too many categories.

Saving the budget on hard disk

- Alternative 1 (Current Choice): Saving into its own file
    - Pros: Fast as there is only a need to store one line of data.
    - Cons: Would require management of several files by user.
- Alternative 2: Save into the same file as other data
    - Pros: Only one file is needed to be managed by the user.
    - Cons: Slow to read in the budget if there are other data in the file as there would be a need to iterate through and edit only one line if the budget changes.
- Alternative 1 was chosen as saving the budget into its own file would allow it to be simple to read and write information to and from the file. This is because having only one line would make it easier to read and overwrite the one single line.

Graph Feature

2.1 Proposed Implementation

The graph feature lets the user visualise the expenditure data in the form of a horizontal bar graph. There are 2 types of graphs that can be generated; the first type displays a graph of all the individual expenditures in a selected category (with month and year also selected by the user), and the second type displays a graph of the total expenditures of all categories (of the current month and year).

The first type of graph is generated in the **GraphCategoryCommand** class that extends the **Command** class. The second type of graph is generated in the **GraphTotalCommand** class that also extends the **Command** class. A standalone Graph class is not utilised as the generated graph is not intended to be stored.

When a *graph* command is given, either the **GraphCategoryCommand** class or the **GraphTotalCommand** class will be called, depending on the parameter that follows.

Entering “*total*” as a parameter after *graph* (ie. *graph total)* will call the **GraphTotalCommand** class.

Entering a category after *graph* (ie. *graph c/[CATEGORY]*) instead calls the **GraphCategoryCommand** class, and it can also accept optional parameters for month and year (ie. *graph c/[CATEGORY] m/[MONTH] y/[YEAR]).*

If only *graph* is given, the user will be prompted to complete the command. Only a one-line input command is accepted.

The **GraphTotalCommand** class and the **GraphCategoryCommand** class will depend on the **CategoryList** class to allow for both classes to gain access to the different categories, and the total expenditure costs for each category. Both will also depend on the **Category** class to gain access to the individual expenditures inside each category.

Here is an example of the steps that will be taken if a user were to display the graph for the category “**food”** for the month of February**.**

1. After the application is launched, the user enters the input *graph c/food m/2***.** Since the *graph* command is given with a category parameter, the **GraphCategoryCommand** class is returned.

2. The **execute** method would then read through the **food** category from the **CategoryList.** A string attribute, **output**, will be initiated with multiple “┬” symbols to be displayed as the top border of the bar graph. The **output** attribute will then be repeatedly be concatenated in this format:

   a. Name of the expenditure (If the name is longer than 14 characters, it will be truncated to the 11th character and concatenated with “...”
   b. Full blocks “█” and half blocks “▌” corresponding to the percentage of that expenditure out of the total expenditure costs (The greater the percentage, the more full blocks will be concatenated)
   c. Numerical percentage of the expenditure (This number will be be accurate to 2 decimal places)
   d. New Line (The next line will be for the next expenditure in the category)

   Lastly, multiple "┴" symbols will be concatenated to ***output*** to be displayed as the bottom border of the bar graph, which is then printed by the **UI** class using the method **UI.setOutput().**

3. In the event that the category is empty or there are no categories, the user will receive a notification letting the user know that there is no data to be displayed.

4. Should the user input *graph total*, the graph displayed will show the total expenditure costs for each category. Instead of reading through the expenditure in a category, the **execute** method now reads through the entire **CategoryList** to obtain the

category names and their respective total expenditure costs for the current month and year.

5. Below is the activity diagram for **GraphTotalCommand**.

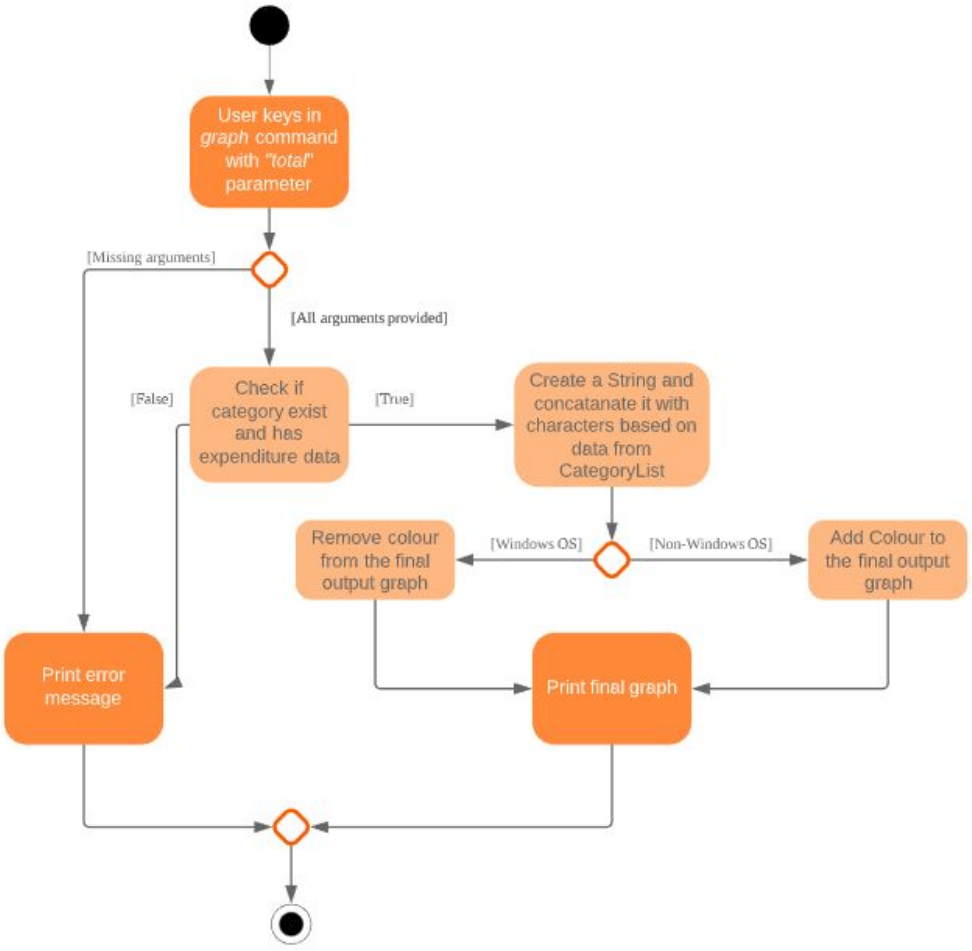

Figure 2.1.1 Activity Diagram for **GraphTotalCommand**

6. Below is the sequence diagram for **GraphCategoryCommand:**

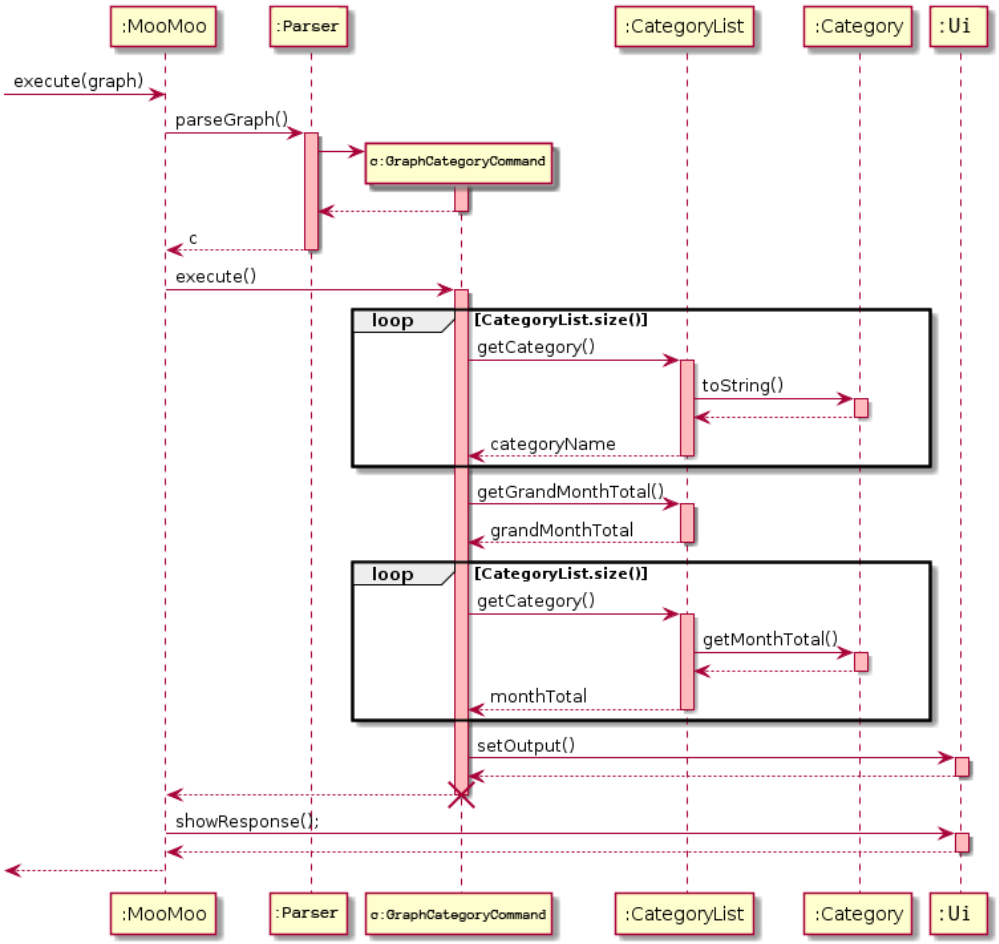

Figure 2.1.1 Sequence Diagram for **GraphCategoryCommand**

2.2 Alternatives

Plotting the graph

- Alternative 1 (Current Choice): Plotting a horizontal bar graph
    - Pros: Allows the names of the expenditure/categories to be displayed horizontally
    - Cons: Horizontal bars could potentially wrap over to the next line if the width of the terminal is not long enough
- Alternative 2: Plotting a vertical bar graph
    - Pros: Looks more conventional, easier to see and compare the differences in percentage of each expenditure/category
    - Cons: Too many expenditures/categories could cause the output to become garbled if the width of the terminal is not long enough.

Alternative 1 was chosen as it would be more functional for users to be able to see the category/expenditure name properly, making the graph more visually informative.

Category Feature

3.1 Proposed Implementation

The Category feature lets the user create categories to organize his expenditure. All categories are stored in a **CategoryList** class which contains an **ArrayList<Category>**. And a **Category** class contains an **ArrayList<Expenditure>**, **categoryName,** and a **sortOrder.**

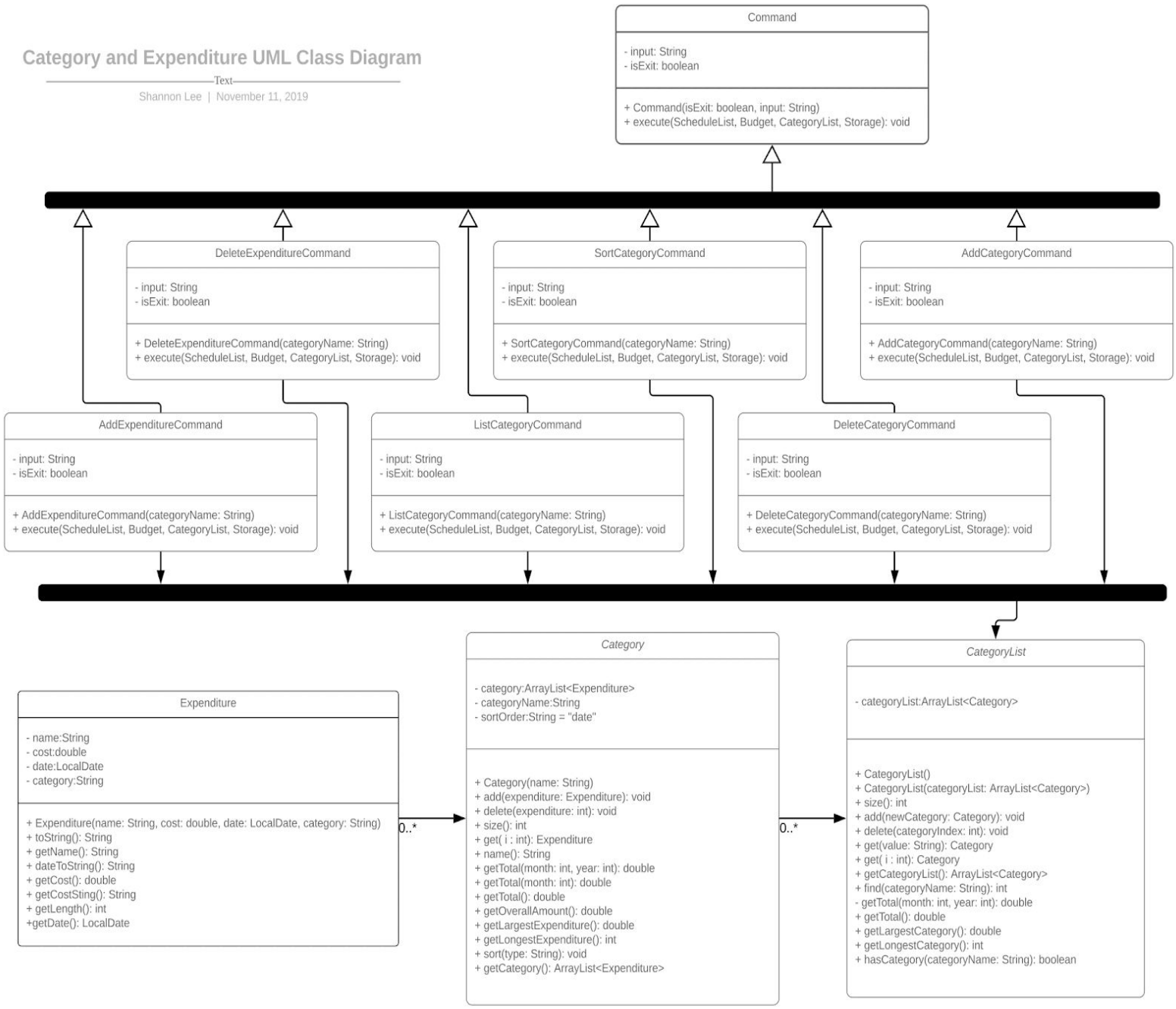

Figure 3.1 Category and Expenditure Class Diagram

Users can *list*, *add*, *delete*, and *sort* categories using the corresponding command which all extend the **Command** class as shown in Figure 3.1.

3.1.1. ListCategoryCommand

When users enter *list*, a **ListCategoryCommand** is returned from the parser. The **execute()** method populates an **ArrayList<String>** with the name of each category and its total expenditure for the current month and year. The **ArrayList** is passed into the **Ui.showList()** method along with the length of the longest category name, which generates a box around the list of categories and an ASCII cow beneath it. The output is set and then printed out in the **Ui.showResponse()** method.

3.1.2. AddCategoryCommand

New categories can be created with the **AddCategoryCommand** when the user enters *category add.* The **CategoryParser** class then reads in the **categoryName**, which is not allowed to contain the character "/" to prevent parsing confusion, and then returns an **AddCategoryCommand** initialized with the **categoryName**. The **execute()** method creates a new **Category** initialized with the **categoryName** and checks if the category already exists in the **CategoryList**. If there is no existing category with that name, it is then added to the **CategoryList** and the **CategoryStorage**. Lastly, the **Ui.showCategoryMessage()** method sets the output to the ASCII cow and a speech box showing the confirmation message and then printed out in the **Ui.showResponse()** method.

3.1.3 DeleteCategoryCommand

Categories can also be deleted with the **DeleteCategoryCommand** when the user enters *category add*. The **CategoryParser** class then reads in the **categoryName** to be deleted. The **categoryName** can either be an integer corresponding to the index of the category, or a string matching the name of one of the existing categories. The parser then returns a **DeleteCategoryCommand** initialized with the **categoryName**. The **execute()** method checks if the category specified can be found. If so, it is removed from the **CategoryList** and from the **CategoryStorage**. **Ui.showCategoryMessage()** and **Ui.showResponse()** then prints the confirmation message as explained above.

Here is an example of the steps that will be taken if a user were to add a category:

1. After the application is launched, the user enters the input *category add games*. Since the *category add* command is used, the **CategoryParser** returns an **AddCategoryCommand** initialized with *games* as the **categoryName** field.

2. When the **execute()** method is called in **AddCategoryCommand**, a new **Category** is initialized with *games* as **categoryName** and added to the **CategoryList** if games category does not already exist. The new category will be added to the storage file as well using the **saveCategoryToFile()** method in the **Storage** class. And lastly, the **UI** class will confirm to the user that the category is added with the **showNewCategoryMessage()**.

3. After adding a category, the user enters *category delete 5* and a DeleteCategoryCommand is will be initialized with **categoryName** 5.

4. The **execute()** method is called and deletes the category using the **delete()** method in **CategoryList** which takes in an integer as the **categoryNumber**. Then it removes the category from the storage file using the **removeCategoryFromFile()** method in the **Storage** class. And the **showRemovedCategoryMessage()** method in **UI** confirms to the user that the category has been deleted.

3.1.4 SortCategoryCommand

The **SortCategoryCommand** allows users to set how the expenditures in each category should be sorted by entering *sort*. The **ExpenditureParser** reads in **sortType** and returns a **SortCategoryCommand** initialized with the **sortType**. The **execute()** method checks if it should sort by *name*, *date*, or *cost,* then iterates through each category and calls the **category.sort()** method which sorts all the expenditures within the category. Finally, it shows the **MainDisplay** using the **Ui.printMainDisplay()** method.

3.1.5 CategoryList

The **CategoryList** is initialized with categories stored on the storage file. If there is no data to read, default categories Misc, Food, Transportation, and Shopping will be added instead. Within the **CategoryList** class are several methods which are used by the **Graph, Notification, and Budget** classes. The first is a **getGrandMonthTotal()** method which takes one integer parameter for the month. It iterates through the CategoryList and sums up the retrieved total of all expenditure within the category for the specified month.

The **getLargestExpenditure()** method takes one integer parameter for the month, and iterates through the CategoryList to find the category with the largest total expenditure. The return value is the largest total expenditure from the categories in **CategoryList**.

The **getLongestCategory()** method iterates through the **CategoryList** and returns the integer number of characters in the name of the category with the longest **categoryName**. If any category name has more than fourteen characters, it returns fourteen.

3.1.5 CategoryList

A **Category** is initialized with one parameter which is the **categoryName**. The **ArrayList<Expenditure>** is initialized as an empty **ArrayList**, which stores all expenditures added to that category, and a sortOrder which has default value **date,** which specified how the **ArrayList** is sorted.

The **getCategoryMonthTotal()** method in the **Category** class returns the total expenditure in that category for a specified month. The method takes one integer parameter for the month and finds all expenditures entered within that month.

Expenditure Feature

4.1 Proposed Implementation

The Expenditure feature lets the user create different expenditures under a specific category. All expenditures are stored in **Category** class which contains an **ArrayList<Expenditure>.** An **Expenditure** class contains a **cost,** an **expenditureName**, and a **dateTime**.

New expenditures can be added using the **AddExpenditureCommand**, which extends the **Command** class. When an ***add*** command is given, it takes in two parameters which are **expenditureName** and **amount**, adds a new **Expenditure** to the specified **Category,** and stores the new Expenditure into the disk file.

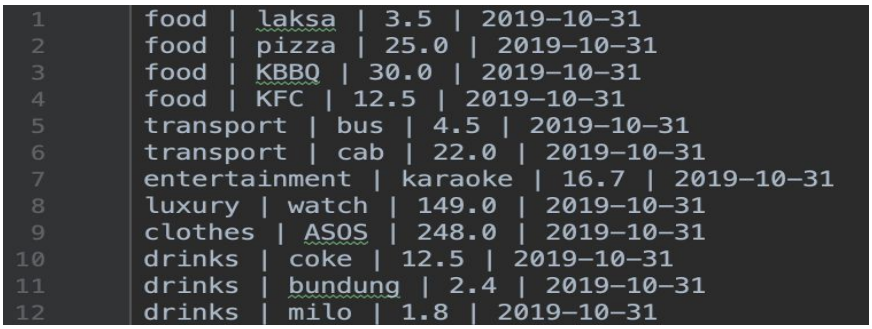

Figure 4.1.1 Expenditures stored in **expenditure.txt file**

Expenditures can also be deleted with the **DeleleExpenditureCommand**, which extends the Command class. When a delete Expenditure command is given, it takes two parameters in which are **categoryNumber** and **amount**, and deletes the expenditure corresponding to the amount in the specified **Category**.

Expenditures can also be edited with the **EditExpenditureCommand**, which extends the Command class. When an edit Expenditure command is given, it takes three parameters in which are **categoryNumber**, **oldAmount**, and **newAmount**, and edits the expenditure corresponding to the newAmount in the specified **Category.**

The **getCategoryMonthTotal()** method under the **Category** class, iterates through the different Expenditures under a specified Category and sums up the retrieved total of all the expenditures within that Category.

Here is an example of the steps that will be taken if a user were to add, delete or edit an expenditure:

1. After the application is launched, the user enters the input add *n/laksa a/10 d/20/12/2019 c/food*. Since the add command is used, as the start of the command does not contain a “c/”, the parser recognises that this is not an addition of category command. The **AddExpenditureCommand** will then be called and initialized with food as **expenditureName,** 10 as **amount** and 20/12/2019 as **dateTime.**
2. When the **execute()** method is called in **AddExpenditureCommand**, a new **Expenditure** is initialized with **expenditureName** as laksa and added to the **Category** food with the **add()**

method. The new expenditure will be added to the storage file as well using the **saveExpenditureToFile()** method in the **Storage** class. And lastly, the **UI** class will confirm to the user that the category is added with the **showNewExpenditureMessage()**.

3. After adding an expenditure, the user can enter *edit n/laksa a/20 c/food* and **EditExpenditureCommand** will be initialized with a new **amount** 20, to make changes to the Expenditure under the **Category** food.
4. When the **execute()** method is called in **EditExpenditureCommand**, the method **exitExpenditure()** in **Category** class is called which will edit the Expenditure laksa and change the amount which has been specified. It then updates the expenditure from the storage file using the **editExpenditureFromFile()** method in the **Storage** class. And the **showEditedExpenditureMessage()** method in **UI** confirms to the user that the expenditure has been edited.
5. If the user decides to delete the Expenditure, the user can enter *delete n/laksa c/(***int** *categoryNumber)* and a **DeleteExpenditureCommand** will be called.
6. When the **execute()** method is called in **DeleteExpenditureCommand**, the method **deleteExpenditure()** in **Category** class is called which will delete the Expenditure laksa under the specified category that has been listed by **list().** Then it removes the expenditure from the storage file using the **removeExpenditureFromFile()** method in the **Storage** class. And the **showRemovedExpenditureMessage()** method in **UI** confirms to the user that the expenditure has been deleted.

Notifications Feature

5.1 Proposed Implementation

The notifications feature is an automated alert. It will alert the user when the budget for a category is exceeded. It will also show the user the budget balance automatically when the user adds an expenditure. This feature is implemented by the **NotificationsCommand** class which extends the **Command** class.

When the user adds an expenditure, a NotificationsCommand object is created which consists of a String for category name and Double for total expenditure of the category.

Here is an example of what happens after the user adds an expenditure.

1. A NotificationsCommand object is created and is instantiated in this manner: **NotificationCommand alert = new NotificationCommand(categoryName, totalExpenditure)**
2. The value passed in for totalExpenditure uses the method from **Category** class **getCategoryMonthTotal()**.
3. When **NotificationCommand** is executed, it compares **totalExpenditure** with the budget set for the **categoryName** passed in.
4. The budget set for the **categoryName** is called using the method **budget.getBudgetFromCategory(categoryName)**.

5. If the user exceeds the budget (**totalExpenditure > budget**), a **String alert** will be set to inform the user that the budget has been exceeded. This alert will be red in colour.

   If the user has reached the budget limit (**totalExpenditure** = **budget**), the **String alert** will be set to inform the user that he has reached the limit. This alert will be red in colour.

   If the user is reaching the budget limit (**totalExpenditure** > 90% of **budget**), the **String alert** will be set to tell the user that he is reaching the limit. This alert will be yellow in colour.

   If the user still has a significant amount of balance left (**totalExpenditure** < 90% of **budget**), only the budget balance will be displayed. This balance will be green in colour.

6. Next the budget balance is stored in the variable **double balance**. It is calculated by subtracting the budget of the category by the total expenditure of the category.
7. Lastly, the **Ui** class is called to use the method **ui.setOutput()** to display the alert and budget balance to the user.

Scheduled Payments Feature

6.1 Proposed Implementation

The scheduled payments feature allows user to set a reminder in advance for an incoming bill/expenditure that will occur in the future. When the programme runs, the scheduled payments for the day is automatically shown to the user. Users can also see the list of all their schedules manually. This feature is implemented by the **ScheduleCommand** class which extends the **Command** class.

When *schedule* command is used, the **ScheduleCommand** class will be called. To add a scheduled payment, the user should input the scheduled date, amount that needs to be paid and the task for the payment in the same line as *schedule* command. If the date/amount/task is not provided, the user will be prompted to fill in these information. If the date keyed in is an invalid date or if the amount keyed in is not a number, the user will also be prompted to fill in these information.

To see all the schedules added, the user has to input *schedule list*. This will display all the schedules added sorted by date.

The **ScheduleCommand** class uses **ScheduleList** class. The **ScheduleList** class is a HashMap using the date as the key and an ArrayList of schedules as the value. When a user adds a schedule, it will be added to the **ScheduleList** object created. After the user finishes adding the schedule, the updated schedule will be saved in a text file by calling the method **saveScheduleToFile(ScheduleList)** from **Storage** class. The associations between each component can be seen in the class diagram in Fig 6.1.

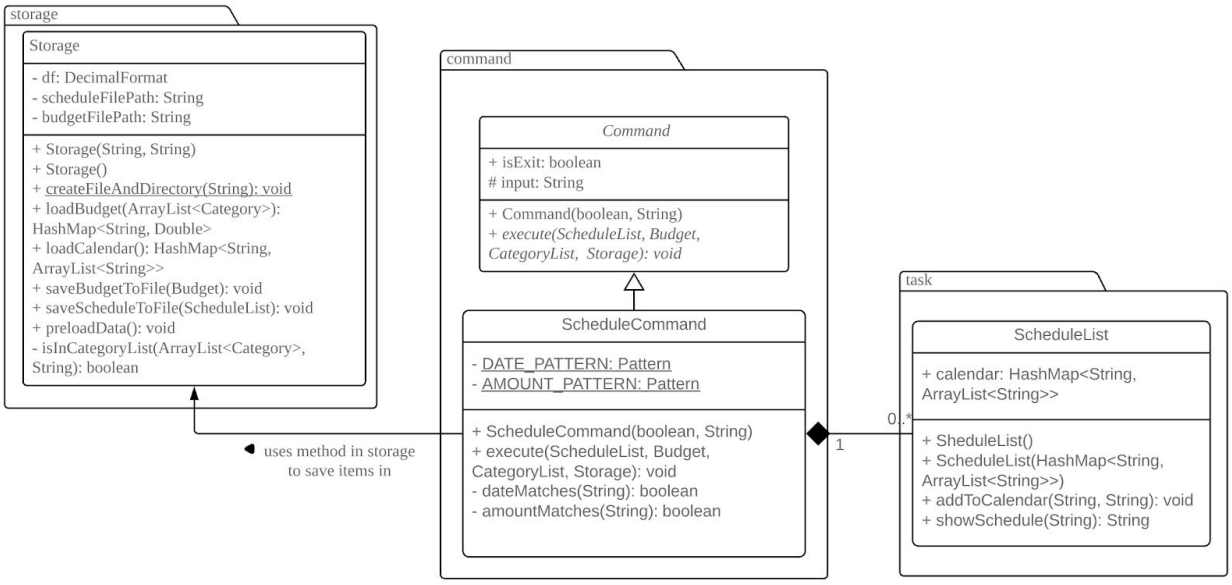

Fig 6.1 Class diagram for **ScheduleCommand**

Here is an example of what happens when the user adds a scheduled payment. Refer to Fig 6.2.1 to view the activity diagram for this flow.

1. User keys in *schedule d/23/10/2019 a/200 n/electricity bills*. As schedule command is used, the **ScheduleCommand** class will be executed.
2. The user input will be split to filter out the date and combine the task and amount into one string: task.
3. A **SchedulePayment** object is created for this task and is instantiated in this manner: **SchedulePayment newTask = new SchedulePayment(date, task)**.
4. **newTask** object is then added to the **ScheduleList** class ***calendar*** which was instantiated from the start. ***calendar*** contains all the tasks that the user has scheduled.
5. Once the schedule has been successfully added, **Storage** class is called using the method **storage.saveScheduleToFile(*calendar*)** to store the **ScheduleList calendar** in a text file.

When MooMooMoney starts running, the schedule for the day is automatically shown. This is done by using the method in **ScheduleList** class: **showSchedule(*date*)**. This method takes in the current date as a String value and returns a String output which contains the entire schedule for the day. In the method, a for loop is used to loop through ***calendar*** to search for the tasks that are scheduled for the current date. The UI class will then set the output of this list and print it to

be displayed at the start of the programme.

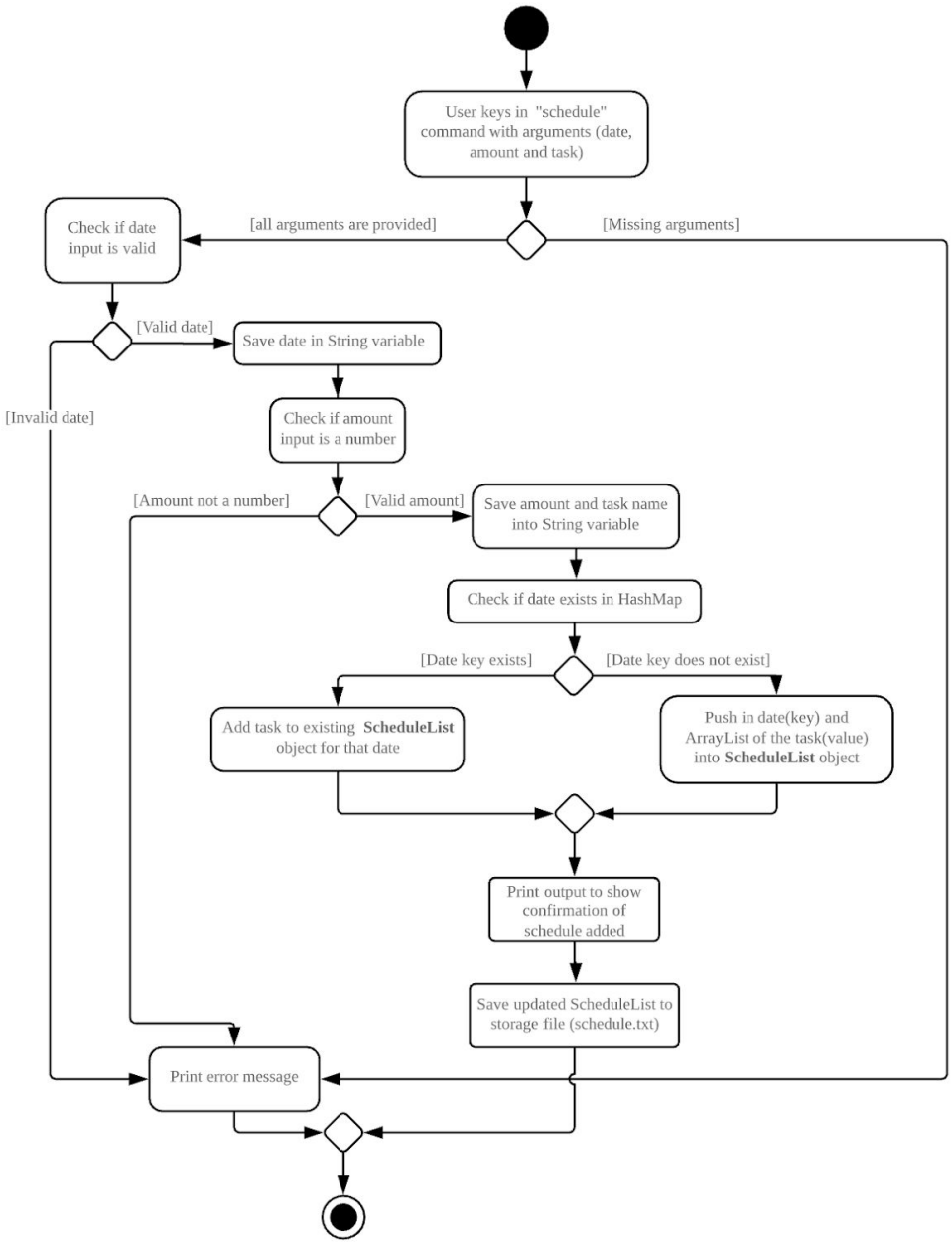

Fig 6.1.2 Activity Diagram for Scheduled Payments Feature

Here is an example of what happens when the user wants to view the entire schedule.

1. User keys in *schedule list*. As *schedule* command is used, the **ScheduleCommand** class is executed.
2. **calendar** object is iterated through to get all the date keys as a String.
3. For each date represented as a String, **SimpleDateFormat** is used to format the String date into a **Date** object.
4. The Date object obtained after the conversion of the String date will be pushed into the **TreeMap** object called **sorted** as the key and the corresponding value (ArrayList of schedules) will be inserted as the value.

5. Once the original **calendar** object has been copied over to the **sorted**, a while loop is used to iterate through sorted to print out the ArrayList of schedules for each date key in order (earliest date first).

6.2 Alternatives

How the schedule is structured and stored

- Alternative 1 : Create a SchedulePayment class that consists of the date and task name. Add each SchedulePayment object instantiated to an ArrayList of SchedulePayments.
    - Pros : Easy implementation. When number of schedules set is small, it is easy to loop through all the schedules to find the schedules that match for a specific date.
    - Cons : When the number of schedules created is a lot, ArrayList of SchedulePayment objects will get too huge. Looping through the entire list to find schedules for the current date may be inefficient. Also, this method requires the use of 3 classes
- Alternative 2 (Current Choice) : Create a HashMap where the keys are the dates and values are the ArrayList of payments scheduled on a specific date. ScheduledList class contains the HashMap that stores the ArrayList of payments for each date.
    - Pros : More organised data. More efficient. All the payments scheduled on a certain date can be easily found just by searching for the key in the HashMap.
    - Cons : Slightly more complicated to implement.

The HashMap alternative was chosen since it is much more organised in storing the data. It also allows easy and efficient access to all the payments on one date. In contrast to alternative 1, alternative 1 is slow since the program would have to run through every single scheduled payment to get all the payments due for a specific date. When our data set gets even larger, it will be significantly slower to use alternative 1.

Main Display Feature

7.1 Proposed Implementation

The Main Display feature allows the user to view an overall summary of all their Categories/Expenditures/Budgets/Savings as well as the period of interest (month/year) which can be specified when calling the **MainDisplayCommand**. The table is generated in the **MainDisplay** class and is generated by the method **newToPrint()**.

When a MainDisplay command is given, the **MainDisplayCommand** class that extends the **Command** class will be called. **MainDisplayCommand** will use an abstract **execute()** method from the **Command** class. Depending on the input of the user, the command to be run will be different. The command will be in either three of these formats: *view current, view all,* or *view <m/> [MONTH] <y/> [YEAR]*. This allows for a user to either view a current overall summary, or a certain month's summary (all defaults to the current month and year).

```
view all

.----------------.-------------------------------------------------------------------------------------------------------.
|Month: All      |                                          <Categories>                                                 |
|Year: All       |-----------------------.-----------------------.-----------------------.-----------------------.-------|
|                |food                   |transport              |entertainment          |logistics              |accomo|
|                |-----------------------------------------------------------------------------------------------------|
|                |laksa        |$5.00    |bus          |$3.40    |karaoke      |$20.30   |glue         |$1.90    |hostel|
|                |pizza        |$35.00   |taxi         |$24.50   |movie        |$12.20   |papaer       |$8.00    |      |
|                |curry        |$3.50    |van          |$22.00   |skating      |$30.20   |             |         |      |
|                |fishball     |$2.20    |heli         |$60.00   |             |         |             |         |      |
|                |sushi        |$12.50   |             |         |             |         |             |         |      |
|                |ramen        |$14.50   |             |         |             |         |             |         |      |
.----------------.-----------------------.-----------------------.-----------------------.-----------------------.-------.
|Total:          |$72.70                 |$109.90                |$62.70                 |$9.90                  |$546.0|
.----------------.-----------------------.-----------------------.-----------------------.-----------------------.-------.
|Budget:         |$500.00                |$300.00                |$300.00                |$200.00                |$600.0|
.----------------.-----------------------.-----------------------.-----------------------.-----------------------.-------.
|Savings:        |$427.30                |$190.10                |$237.30                |$190.10                |$54.00|
.----------------.-----------------------.-----------------------.-----------------------.-----------------------.-------.
```

Figure 7.1.1 Month's Main Display from **MainDisplayCommand**

The **MainDisplayCommand** class will depend on the **CategoryList** and **Budget** class to allow for **MainDisplayCommand** to gain access to the added categories as well as their expenditures, and the budget set for each categories. It would also depend on the same classes to get the total transactions spent per month to allow for the calculation of any savings accumulated using **budget - total expenditure**. The copy of the display table will not be stored in a text file as every time the command is run, the table will be generated from existing data that is already stored. (budget.txt, category.txt, expenditure.txt)

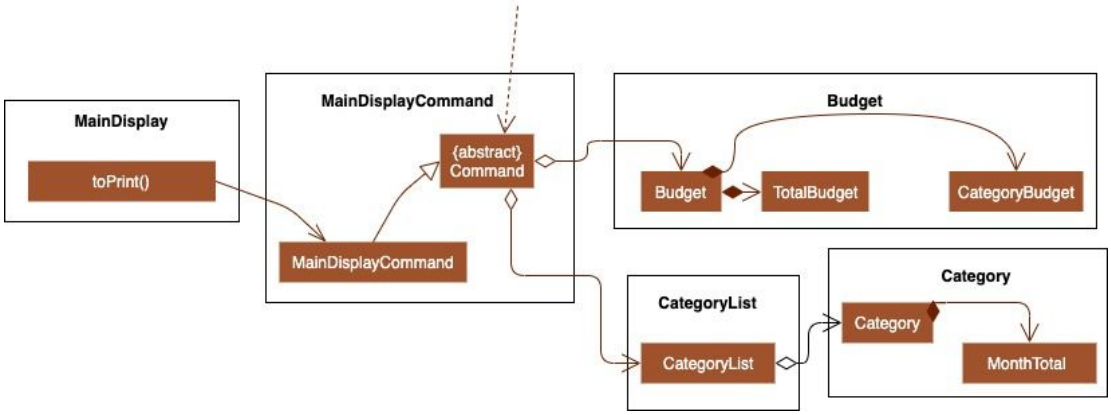

Figure 7.1.2 Class diagram for **MainDisplayCommand**

Here is an example of the Main Display when a user calls the MainDisplayCommand:

1. User keys in '*view current*'. As 'view' is called without a month <*m*/> or year <*y*/> argument, the parser assumes user is planning to view the current month's summary and will call the **MainDisplayCommand** passing in parameters with the current month and year as inputs. This is done with the help of java's LocalDate method.

```
} else if (input.equals("current")) {
    int month = now.getMonth().getValue();
    int year = now.getYear();
    return new MainDisplayCommand(month, year);
```

2. The **execute** method would then create a new **MainDisplay** object for this task.
3. The methods **getMonthsExpSize()** and **getCatListSize()** from **MainDisplay** class will then be used to determine the number of rows and columns respectively, that will be in the Main Table.
4. The horizontal length of the MainDisplay table will be determined by the amount of Categories present in the CategoryList. The vertical length of the MainDisplay table will be determined by the Maximum number of expenditures in a present Category class. The parameters of the table will change as the user adds/deletes Categories of Expenditures.
5. For each Expenditure, amounts will be extracted as double(s) and formatted into Strings till up to 2 decimal places. This is done using java.text.DecimalFormat.

```
private static DecimalFormat df = new DecimalFormat( pattern: "0.00");
```

6. If no Categories have been added in the CategoryList, output will return a default table with only one Category called 'misc' and no expenditures.

```
.--------------------.--------------------------.
|Month: All          |       <Categories>       |
|Year: All           |--------------------------.
|                    |misc                      |
|                    |--------------------------.
|                    |              |           |
.--------------------.--------------------------.
|Total:              |                          |
.--------------------.--------------------------.
|Budget:             |                          |
.--------------------.--------------------------.
|Savings:            |                          |
.--------------------.--------------------------.
```

7. If there are Categories, the method **newToPrint()** from **MainDisplay** will go through all the expenditures in the period specific and append them to a long String, **output**.
8. We will get the Total expenditure from a Category for the specified period by calling **getTotal(month, year)** from **Category** class.
9. Then we will find the Budget from that Category for the specified period by calling **getBudgetFromCategory(categoryName)** from **Budget** class.
10. To calculate the Total Savings from a Category for the specified period, we will simply subtract the Total expenditure from the Budget.
11. Lastly, the method **newToPrint()** from **MainDisplay** class is called with the parameters *month, year, rows, cols, categoryList, budget and t (device type)*, which converts all the data into a long String, **output**, to be printed out using **ui.printMainDisplay(output).**

```
String output = newMainDisplay.toPrint(month,year,rows,cols,categoryList,budget);
ui.printMainDisplay(output);
```

12. The **toPrint()** method calculates the amount of blank space to print out after each input (if any entry exists) and appends it to the final string to be printed out. The method makes use of ANSI standard to give colour to the output table. As ANSI is incompatible with some versions of Windows, we will disable the coloured output for users on Windows. This is done by passing a *t* value of 1 (default 0 for mac), into the **newToPrint()** method.

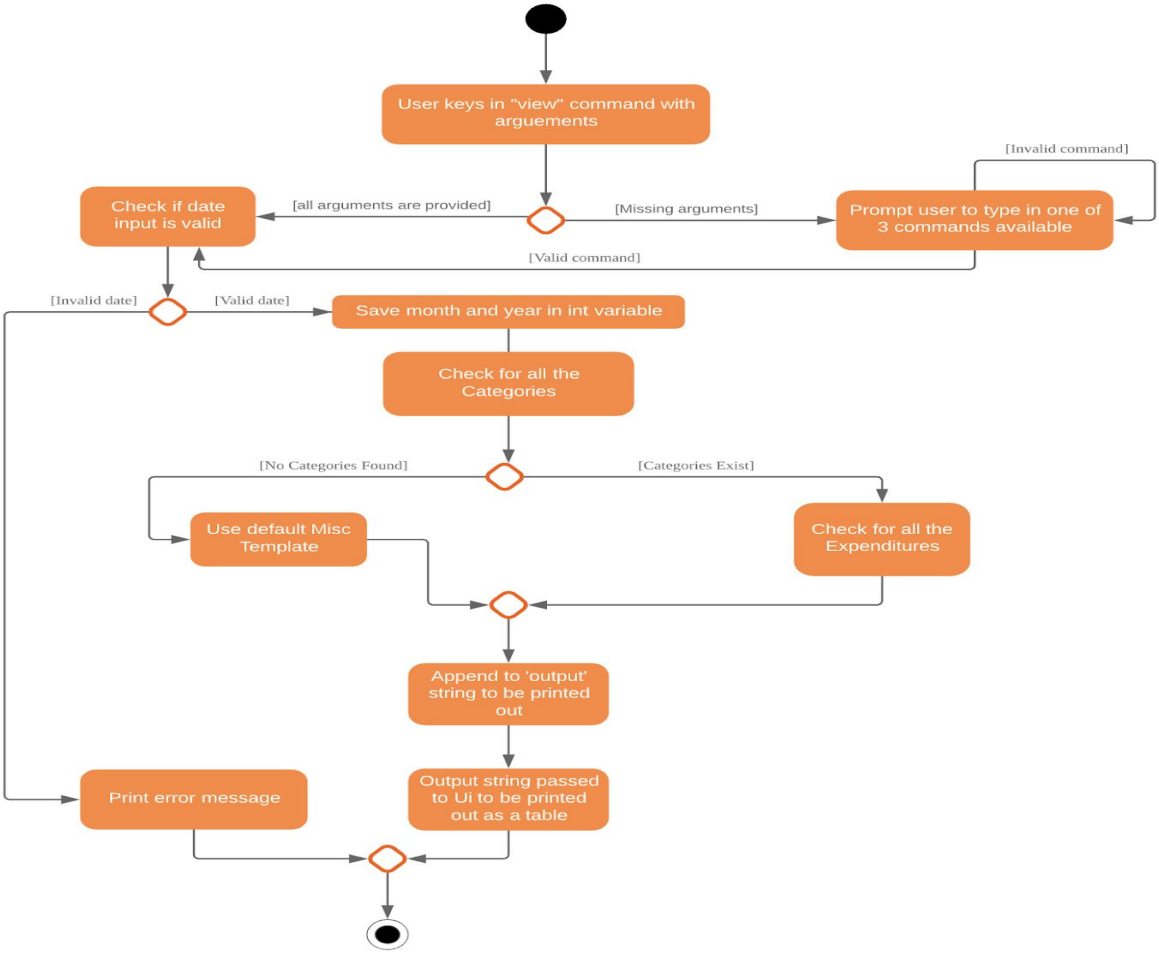

Figure 7.1.3 Activity Diagram for **MainDisplayCommand**

7.2 Alternatives

How the table is generated and printed

- Alternative 1 (Current Choice) : Extract data from saved files, converting them into strings before appending into a final String to be printed.
    - Pros : Variable sized table. Size of the table generated is flexible due to the availability of data (e.g. if Category has expenditures, if budget has been set/ new Category added leads to table size increasing)
    - Cons : Currently only expendable downwards as new Categories are being added, but there is still a size limit for number of expenditures added as currently table is not able to expand sideways. (e.g. many Expenditures will lead to thin walls)
- Alternative 2 : Use an existing package or plug-in API that generates table or String API (e.g. StringBuilder) for more efficient appending of data to final output String
    - Pros : More efficient as different types of variables can be appended to String. Might implement in future.
    - Cons : More complicated to implement as lack knowledge on using external libraries.

Calculator Feature (coming in v2.0)

8.1 Proposed Implementation

The application will take in a simple equation such as **100 - 50** as a string and then converting it to the appropriate equation and then printing out the results. It will read in each number and operator until the end of line to allow for multiple operations such as **100 - 50 - 25 + 25**.

Data Encryption Feature (coming in v2.0)

9.1 Proposed Implementation

To ensure that the application is secure, due to the need of inputting financial information, we will perform encryption on the data. There are multiple considerations that we have to take care of, to implement Data Encryption appropriately.

As such, to implement the program, would take the following steps:

1. Allow users to set a username and password that will allow the user to login to view the data.

2. The username and password which will be hashed using salted SHA256 at the very least, will be saved into a seperate file.

3. Once the user logins, the username and password with a static string either generated from part of the username or password will be used as the key for symmetric encryption and decryption (AES) of the actual data of the user.

4. Since the file will be encrypted, the user can login to the application, decrypt the contents of the file, edit and save it before encrypting it again using a command in the program.

Appendix

Appendix A: Product Scope

Project Scope: Money managing app

Project Direction: Morphing

User Profile: University Students who want to better manage their money

Problem Addressed: Currently, students tend to not track how much they have spent on online shopping or on food and before they realized it, they have spent more than they wanted to. As such, this application can help them budgetize every month or day to make sure they do not overspend.

Value Proposition: To make money management for students more intuitive and simplified.

Impact on Society: People can now better spend their money and not be in a situation where they are in urgent need of money but realized that they have already spent everything that they have.

Appendix B: User Stories

1. As a student, I want to be able to use the application from a command line so that it will be using an interface I am comfortable with.

2. As a student, I want to be able to view my expenditure in a visual format so I will be able to see trends at a glance.

3. As a student, I want to be able to see my expenditure in different visual formats so that I can select one that suits my preferences.

4. As a student, I want to be able to use different currencies so that I can use the application overseas.

5. As a student, I do not want the app to depend on a personal server so that I will be able to use it after I graduate from school.

6. As a student, I want to have a calculator in the application so that I can conveniently make math calculations when using the app.

7. As a student, I want to have a way to limit my expenses a month in addition to keeping track of my expenses so that I do not exceed my monthly budget.

8. As a student, I want to be able to categorise my expenditure so that I can see what type of goods am I spending the most in (eg food, travel, stationery)

9. As a student, I want the app to be more personalised so that I have the flexibility to create my own categories especially so that I can create categories for unique items.

10. As a student, I want a fast way to add my expenses so that I can easily log down whatever I've spent easily without taking too much time.

11. As a student, I want to be able to control my expenses throughout the month by being alerted at intervals about my current budget so that I do not overspend at the start and have to save towards the end of the month

12. As a student, I want to be able to view my past month's expenditure exported as an excel file so that I can view it in a nicer format.

13. As a student, I want to be able to add a transaction in a single command line so that I can just copy and paste a command without having to keep retyping.

14. As a student, I want my cow to turn into a bull and change color when I exceed my budget for the month so that I can tell easily when I have exceeded my budget.

15. As a student, I want to see my total saving over the past few months so that I can tell how much I have saved and if I am improving or not.

16. As a student, I want the program to be very efficient so that I can also run multiple other applications as well.

17. As a student, I want the commands to be very short so that I can easily remember the command.

18. As a student, I want to be able to save different types of wallets so that I can view the different transactions from multiple bank accounts.

19. As a student, I want to be able to schedule weekly/monthly/yearly reports on my expenses and incomes so that I can have an overview of my expenses.

20. As a student, I would like to receive warnings and notices when i am about to hit my budget so that I know when I should stop spending.

21. As a student, I would like an interactive interface for a more personalised experience (e.g. Cow "moos", or Cow sending reminders) so that I will not get bored of this application.

22. As a student, I would like to be able to dig up past transactions/history so that I can tell what I have purchased in the past.

23. As a student, I would like to view a graphical summary of my monthly expenses so that I can have an easier reference on my spendings (e.g. split up transport/food/accommodation etc)

24. As a student with multiple computers, I want to be able to back up the data so that I can use it on another computer.

25. As a student concerned with privacy, I want to be able to login to the program so that others cannot view my data.

26. As an administrator, I want to be able to have an overview of the users that I am in charge of so that I can view their transactions and how much they are saving.

27. As a student who shops regularly, I want to be able to add the different websites that I have purchased from so that I can know which website have I purchased the most from.

28. As a student with multiple computers, I want to be able to use it on any operating system without having to configure extra settings.

29. As a student, I want also want to be able to set deadlines for certain events such as school fees so that I can know when I should set aside money to pay for those activities.

30. As a student who plays games, I want to be able to account for a recurring payment every month such as a game subscription so that my budget is allocated accordingly.

31. As a student who loves discounts,, I want to be able to set events on certain dates such as for sales happening so that I can know when I should purchase from the store.

Appendix C: Use Cases

1. System: MooMooMoney
   Actor: User
   Use Case: Set a budget limit per month
   Main Success Scenario:
   1. User enters the budget command
   2. MooMooMoney prompts for the command type (set, list, savings)
   3. User enters the set option
   4. MooMooMoney prompts for the amount.
   5. User enters the amount.
   6. MooMooMoney sets budget accordingly

   Extensions
   3a. MooMooMoney detects an invalid command.
   3a1. MooMooMoney prompts for the correct input.
   3a2. User enters new input.
   Steps 3a1 to 3a2 is repeated until data entered is correct
   Use case resumes from Step 4.

   5a. MooMooMoney detects an invalid input.
   5a1. MooMooMoney prompts for the correct input.
   5a2. User enters new input.
   Steps 5a1 to 5a2 is repeated until data entered is correct
   Use case resumes from Step 6.

2. System: MooMooMoney
   Actor: User
   Use Case: View past transactions in a category
   Main Success Scenario:
   1. User enters the category command.
   2. MooMooMoney prompts for the command type (add, edit, delete, enter).
   3. User selects the enter command.
   4. MooMooMoney display categories and prompts for the category.
   5. User selects the desired category.
   6. MooMooMoney prompts for the subcommand. (add, edit, delete, view).
   7. User types the list transaction command.
   8. MooMooMoney display past transactions for the category.

Extensions
3a. MooMooMoney detects an invalid command.
3a1. MooMooMoney prompts for the correct input.
3a2. User enters new input.
Steps 3a1 to 3a2 is repeated until data entered is correct.
Use case resumes from Step 4.

5a. MooMooMoney detects an invalid command.
5a1. MooMooMoney prompts for the correct input.
5a2. User enters new input.
Steps 5a1 to 5a2 is repeated until data entered is correct.
Use case resumes from Step 6.

7a. MooMooMoney detects an invalid command.
7a1. MooMooMoney prompts for the correct input.
7a2. User enters new input.
Steps 7a1 to 7a2 is repeated until data entered is correct.
Use case resumes from Step 8.

3. System: MooMooMoney
Actor: User
Use Case: Add new categories
Main Success Scenario:
    1. User enters the category command.
    2. MooMooMoney prompts for the command type (add, edit, delete, enter).
    3. User selects the add command.
    4. MooMooMoney prompts for the category to be added.
    5. User enters the category to be added.
    6. MooMooMoney adds the new category.

Extensions
3a. MooMooMoney detects an invalid command.
3a1. MooMooMoney prompts for another input.
3a2. Use enters new input.
Steps 3a1 to 3a2 is repeated until data entered is correct.
Use case resumes from Step 4.

5a. MooMooMoney detects that the category already exists.
5a1. MooMooMoney informs user.
5a2. MooMoo Money prompts for another input.
5a3. Use enters new input.
Steps 5a1 to 5a3 is repeated until data entered is correct
Use case resumes from Step 6

Appendix D: Non Functional Requirements

1. The application should be intuitive as not every user is tech-savvy.

2. There should be an easy way to add multiple transactions if there are a lot of transactions to add.
3. The application should be secure as it deals with personal transactions.
4. The application should be fast in responding even if there is a lot of data.

Appendix E: Glossary

c/ - Category
b/ - Budget (for budget)
d/ - Date
s/ - Start date (for budget)
e/ - End date (for budget)
n/ - Name (for expenditure name)
a/ - Amount (for expenditure costs)
t/ - Task (for scheduling)

m/ - month
y/ - year

Appendix G: Instructions for Manual Testing

1. Initial Launch

a. Download the JAR file and copy into an empty folder
b. Open a terminal (Linux/MacOS) or command prompt/powershell (Windows)
c. Ensure **java -version** shows **11.0.x** and that there is no folder called **data** in the current directory.
d. Change into the directory of the downloaded JAR file and run **java -jar MooMooMoney-v1.4.jar**
Expected: Start up screen will launch and a folder called **data** with 4 text files filled with default values will be created.

2. Add a category

a. Test case: *category add pet*
Expected: pet category will be added
b. Test case: *category add car*
Expected: car category will be added
c. Test case: *category add window*
Expected: window category will be added
d. Test case: *category add lorry*
Expected: lorry category will be added

3. Add an expenditure

a. Test case: *add n/Curry a/5.50 c/Food*
Expected: Curry expenditure will be added to food category with $5.50
b. Test case: *add n/shirt a/50 c/shopping*
Expected: Shirt expenditure will be added to shopping category with $50

4. Set budget

Set budget for a category.

a. Test case: *budget set c/pet b/100*
   Expected: Budget for pet category will be set to $100
b. Test case: *budget set b/100*
   Expected: Budget will not be set as category is not specified
c. Test case: *budget set c/pet b/50*
   Expected: Budget for pet will remain as $100
d. Test case: *budget set c/car*
   Expected: Budget will not be set as budget is not specified
e. Test case: *budget set c/window b/52 c/car b/35.50*
   Expected: Budget for window will be set to $52 and budget for car will be set to 35.50

5. Edit Budget

Edit a budget for a category.

a. Test case: *budget edit c/window b/40*
   Expected: Budget for window will be change from $52 or previous value to $40.
b. Test case: *budget edit c/hello b/100*
   Expected: Budget will not be changed as category does not exist
c. Test case: *budget edit c/lorry b/50*
   Expected: Budget for lorry category will not be changed as it is not set.
d. Test case: *budget edit c/pet*
   Expected: Budget for pet category will not be changed as amount is not specified.
e. Test case: *budget edit b/100*
   Expected: Budget will not be set as no category is given.
f. Test case: *budget edit c/pet b/100.50 c/food b/200*
   Expected: Budget for cup will be changed to $100.50 and budget for food will be changed to $200

6. List Budget

List the budget for the categories

a. Test case: *budget list*
   Expected: Budget for all categories will be listed
b. Test case: *budget list c/pet*
   Expected: Budget for pet category will be listed
c. Test case: *budget list c/table*
   Expected: Budget will not be listed as table category does not exist.
d. Test case: *budget list c/food c/shopping*
   Expected: Budget for food and shopping category will be listed

7. List Savings

List the savings for the categories

a. Test case: *budget savings s/11/2019*
   Expected: Savings for all categories for NOVEMBER 2019 will be listed.
b. Test case: *budget savings c/food s/11/2019*
   Expected: Savings for food category for NOVEMBER 2019 will be listed.
c. Test case: *budget savings c/pet*
   Expected: Savings for cup category will not be listed as month is not given.
d. Test case: *budget savings c/food s/11/2019 e/12/2019*
   Expected: Savings for food for NOVEMBER to DECEMBER 2019 will be listed.
e. Test case: *budget savings*
   Expected: Savings will not be shown as no start month is given.

8. Main display

a. Test case: *view all*

Expected: Table will be displayed showing all the categories/ expenditures and budgets and month and year should reflect 'All' and 'All' respectively

b. Test case: *view m/01 y/2019*

Expected: Table will be displayed showing expenditures and budgets from the specified period and month and year should reflect 'January' and '2019' respectively

c. Test case: *view current*

Expected: Table will be displayed showing expenditures and budgets from the current period.

9. Add Schedule Payment

a. Test case: *schedule d/02/11/19 a/100 n/pay school fees*

Expected: Schedule has been successfully added will be shown represented by the cow speech bubble. The text in the speech bubble should show Date as '02/11/19', Task as 'pay school fees' and Amount as '100'.

b. Test case: *schedule d/31/11/19 a/100 n/electricity bills*

Expected: Exception thrown with the message that it is an invalid date.

10. Notifications

a. Test case: Adding an expenditure that does not exceed 90% of the budget

Expected: Green notification box that displays budget balance.

b. Test case: Adding an expenditure that is more than 90% of the budget but has not exceeded the budget

Expected: Yellow notification box that displays budget balance and inform user that they are reaching the budget limit.

c. Test case: Adding an expenditure that has reached the budget limit

Expected: Red notification box that displays budget balance = 0.0 and inform user that they have reached the budget limit.

d. Test case: Adding an expenditure that has exceeded the budget

Expected: Red notification box that displays budget balance (-ve value) and inform user that they have exceeded the budget limit.

11. Graph

a. Test case: *graph total* (When there are no categories)

Expected: Notification informing user that there are no categories available for viewing.

b. Test case: *graph total* (When there are categories with expenditure in them)

Expected: Bar graph showing all categories and the percentage of their individual total expenditure among the entire expenditure across all categories.

c. Test case: *graph c/food*

Expected: Assuming there is a valid category called "food" with expenditure in it, a graph of all expenditure within the "food" category for the current month and year will be displayed.