Diary (of the) Undergraduate Kommon Engineer (D.U.K.E.) Manager - Developer Guide

By: Team AY1920S1-CS2113-T13-4 Since: Aug 2019 Licence: MIT

Table of Contents

1. Setting up

1.1. Prerequisites

- IntelliJ IDE community edition or ultimate edition (for students) is recommended for the development of this project.
- Ensure that Gradle and JavaFX is enabled, this can be done by going to `File` > `Settings` > `Plugins`
- Ensure that IntelliJ has `JDK 11` or above

1.2. Setting up this project in your computer

- Fork the repo of this project and clone the forked repo to your computer
- Open IntelliJ and ensure that the welcome screen is displayed then click `Configure` > `Project Defaults` > `Project Structure` (If the welcome screen is not opened, click `File` > `Close Project` and the welcome screen will appear)
- Select the correct JDK version for the Gradle, eg.jdk-11.0.3
- After setting up the JDK, click `Import Project` and select the repo that is cloned to your computer
- Open the terminal tab at the bottom left corner and type `gradlew processResources`
- It should show `BUILD SUCCESSFUL` which indicates that ir has generated all resources required by the application and tests

1.3. Verifying the setup

- Using the terminal, type `gradlew run`
- It should show `BUILD SUCCESSFUL` which indicates that the project is set up properly
- Navigate to `test` > `java` > `duke` > `command` on the project panel at the left. Right-click on command and select `Run ‘Tests in ‘duke.command’’`
- Do the same for parser, storage, task. It should show that all tests passed

1.4. Setting up Continuous Integration using Travis

- Go to https://travis-ci.org/
- Sign in using GitHub and enter your GitHub account details if prompted
- Go to settings at the top right side of the page
- Find the switch for the forked repository and toggle it to a green tick
- Done! To check it is working, push a commit to your forked repo, it should show that the Travis is running
- After setting up Travis, you can optionally set up coverage reporting for your team fork (see UsingCoveralls.adoc)

2. Design

2.1. Architecture Design

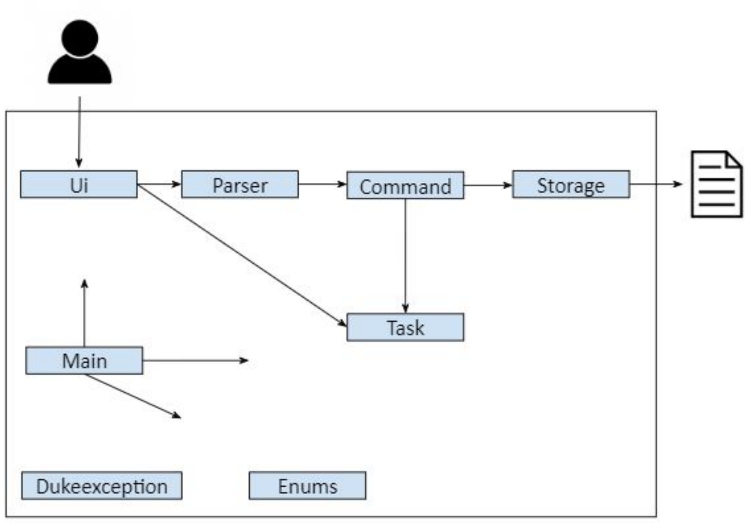


Figure 1. Architecture Diagram

The *Architecture Diagram* which is designed with reference from the Reference Project - Addressbook's Developer Guide explains the high-level design of the App. The following shall give a brief overview of each component.

Main: Main contains the classes Main, Launcher, Duke and DukeLogger. Our Launcher is utilised to Launch Duke Manager in intellij, whereas Main and Duke handles the pre-processing of the code. DukeLogger is a logger that allows us to log and keep track of duke's error messages.

The rest of the Components will be elaborated in the subsections.

- Ui: The User Interface (UI) of Duke Manager
- Parser: A component that handles user input
- Command: Command Executor
- Task: Holds the data used by Duke Manager in the in-app memory
- Storage: Reads and Writes the data that is utilised, or to be utilised by duke to the Hard Disk drive / Solid State Drives

How the architecture components interact with each other:

The *Sequence Diagram* below illustrates how the components interact with each other when the user inputs the command "delete 12".

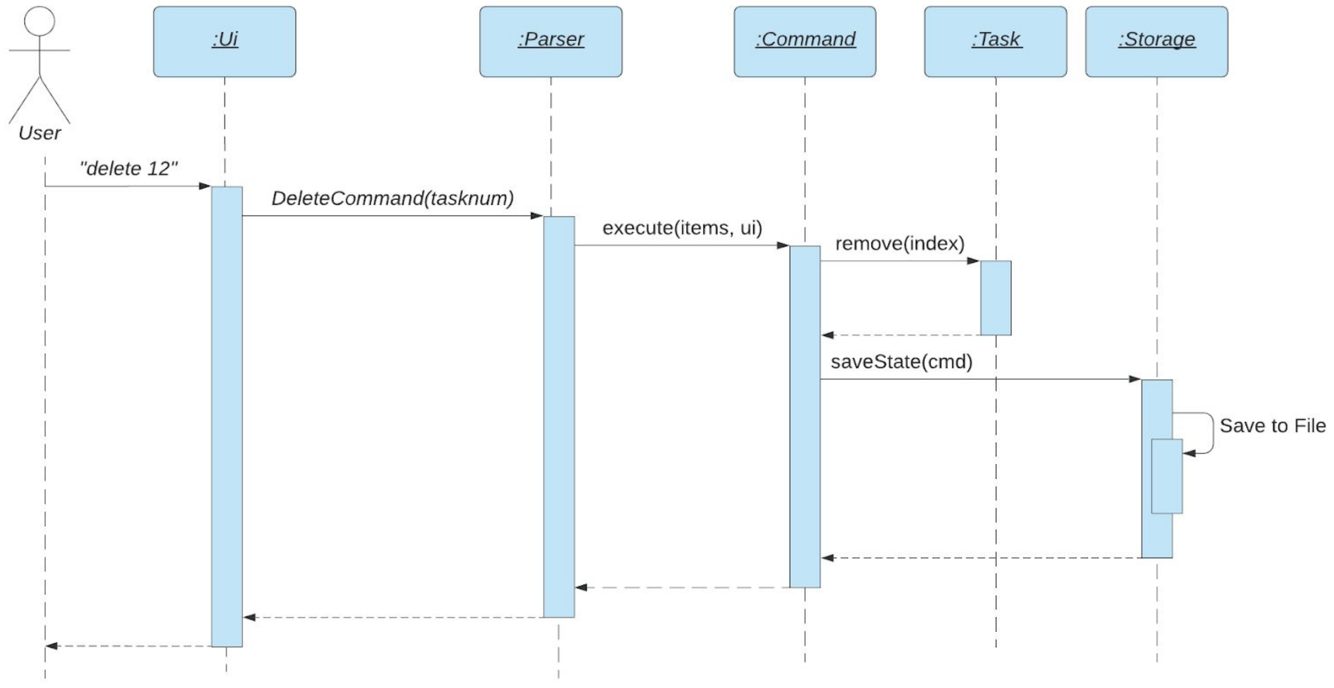


Figure 2. Component interaction when "delete 12" is sent

The sections below gives a more in depth analysis of each component.

2.2. Command Component

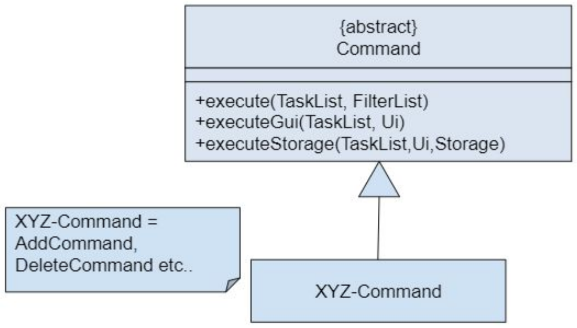

{abstract}
Command
+execute(TaskList, FilterList)
+executeGui(TaskList, Ui)
+executeStorage(TaskList,Ui,Storage)
XYZ-Command =
AddCommand,
DeleteCommand etc..
XYZ-Command

Figure 3. Methods in the Command component

In the command component, it contains the necessary classes that executes the interpreted user inputs. The individual commands are extensions of an abstract Command class, which contains execute(), executeGui() and executeStorage(). executeGui() and executeStorage() has overloading methods hence it was not stated due to repetition.

2.3. Task Component

In the task component, it contains the necessary classes that creates and manipulates the data of Duke Manager. It includes:

2.3.1. BudgetList

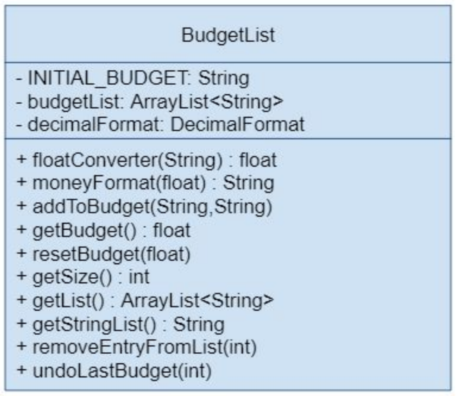

BudgetList
- INITIAL_BUDGET: String
- budgetList: ArrayList<String>
- decimalFormat: DecimalFormat
+ floatConverter(String) : float
+ moneyFormat(float) : String
+ addToBudget(String,String)
+ getBudget() : float
+ resetBudget(float)
+ getSize() : int
+ getList() : ArrayList<String>
+ getStringList() : String
+ removeEntryFromList(int)
+ undoLastBudget(int)

Figure 4. Methods in the Budgetlist component

BudgetList manages the expenditure data of Duke Manager. This includes making relevant changes with regards to budget such as adding, resetting and viewing of the budget.

2.3.2. ContactsList

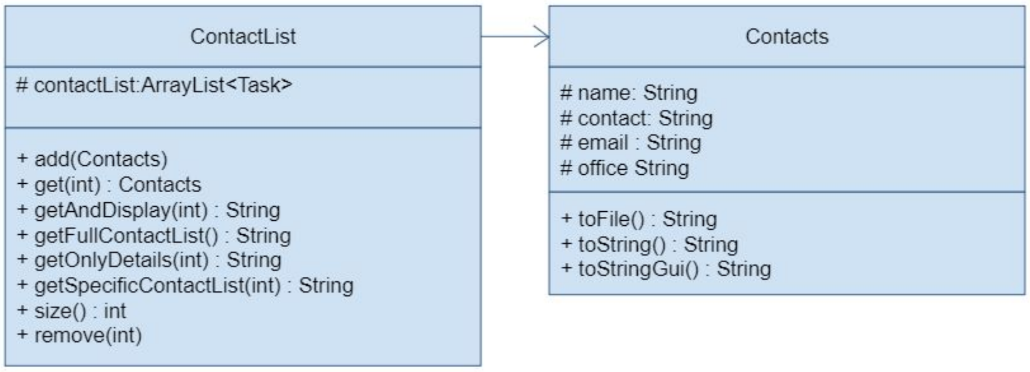


Figure 5. Methods in ContactList component

ContactList manages the contacts that the user inputs. It handles various contact management tasks such as obtaining and removing contact details.

2.3.3. FilterList

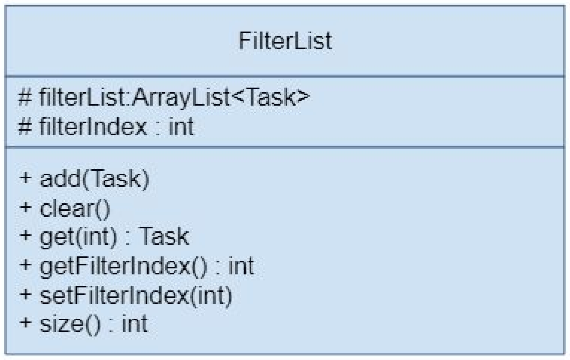


Figure 6. Methods in FilterList component

FilterList manages the filtering of the taskList to their specific task types such as deadlines and todos.

2.3.4. PriorityList

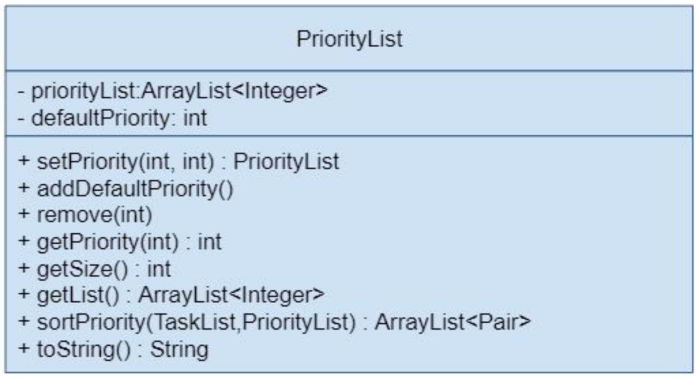

PriorityList
- priorityList:ArrayList<Integer>
- defaultPriority: int
+ setPriority(int, int) : PriorityList
+ addDefaultPriority()
+ remove(int)
+ getPriority(int) : int
+ getSize() : int
+ getList() : ArrayList<Integer>
+ sortPriority(TaskList,PriorityList) : ArrayList<Pair>
+ toString() : String

Figure 7. Methods in PriorityList component

PriorityList manages the priority of each task in tasklist as well as the sorting of the individual task priorities.

2.3.5. TaskList

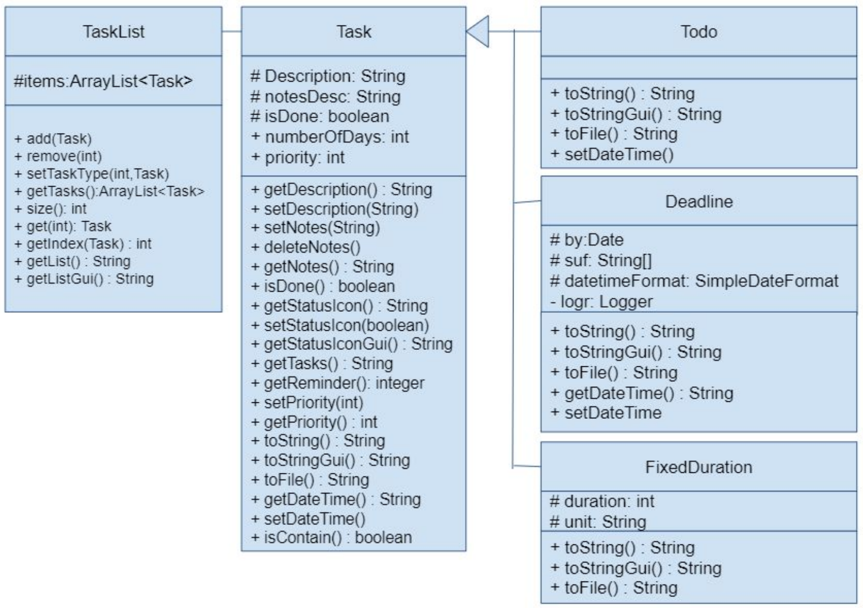


Figure 8. Methods in TaskList component

Tasklist manages the individual tasks of Duke Manager and consolidates them into a list. The individual tasks namely Todo, Deadline and FixedDuration, are extended from Task, which is stored in TaskList. It also facilitates the obtaining of various stored task information such as the date of a deadline task as well as whether the task is completed.

2.3.6. Reminders

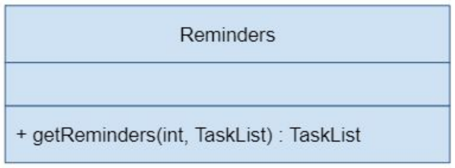


Figure 9. Methods in Reminder component

Reminder obtains the date and time of date-related tasks and check whether the user needs to be reminded of the task.

2.4. Storage Component

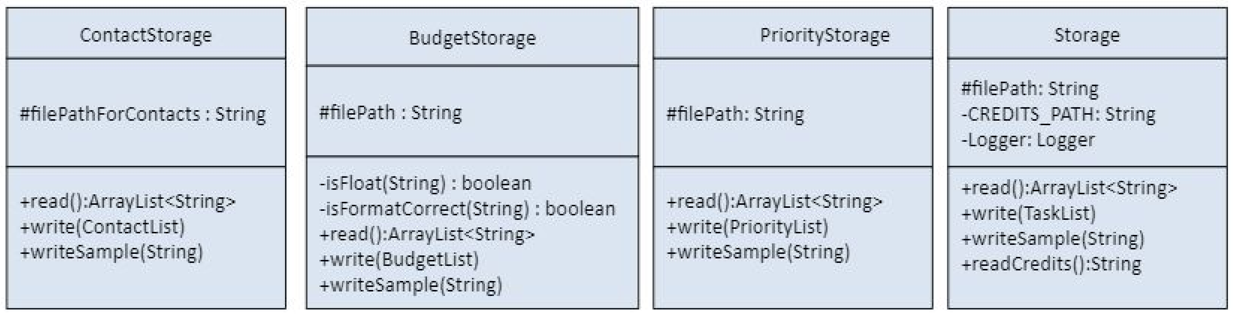


Figure 10. Methods in Storage component

In the storage component, there are 4 main storages being utilised to store each of the lists that contains duke's data, namely the budget list, contact list, priority list and task list.

Each of the storage will have read, write as well as a writesample methods. The writeSample() method will obtain the sample dataset that is stored in src/main/resources/sample folder and populate the write into the storage path. The read method will populate the current list with the data list and the write method will write the lists to the data files on the user's computer.

2.5. Enums Component

The Enums component consists of two main classes, ErrorMessages and Numbers. ErrorMessages enum contains a list of commonly used error messages to be shown if the user accidentally types an incorrect command. Numbers enum contains a list of commonly used numbers.

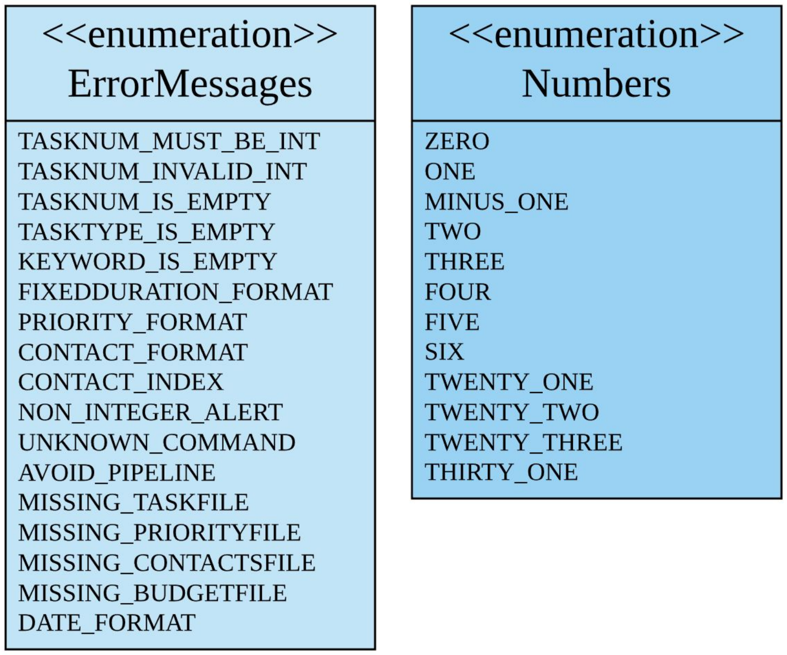


Figure 11. Messages and values used in Enums component

2.6. Parser Component

The parser component handles the interpretation of the user command and then calls the relevant Command Components to execute.

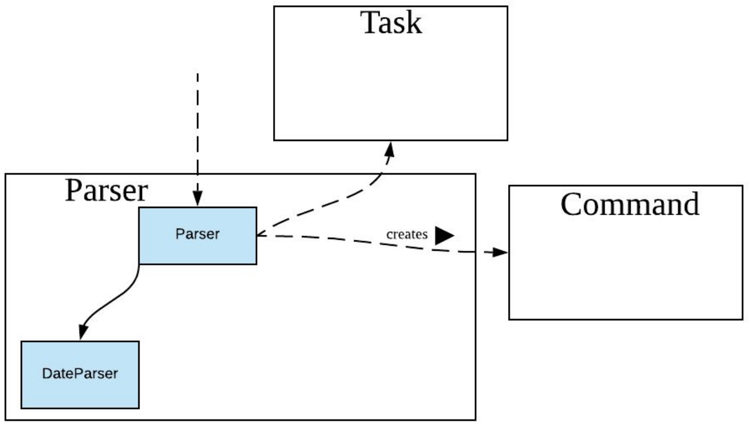


Figure 12. Interaction of Parser with other classes

2.7. UI Component

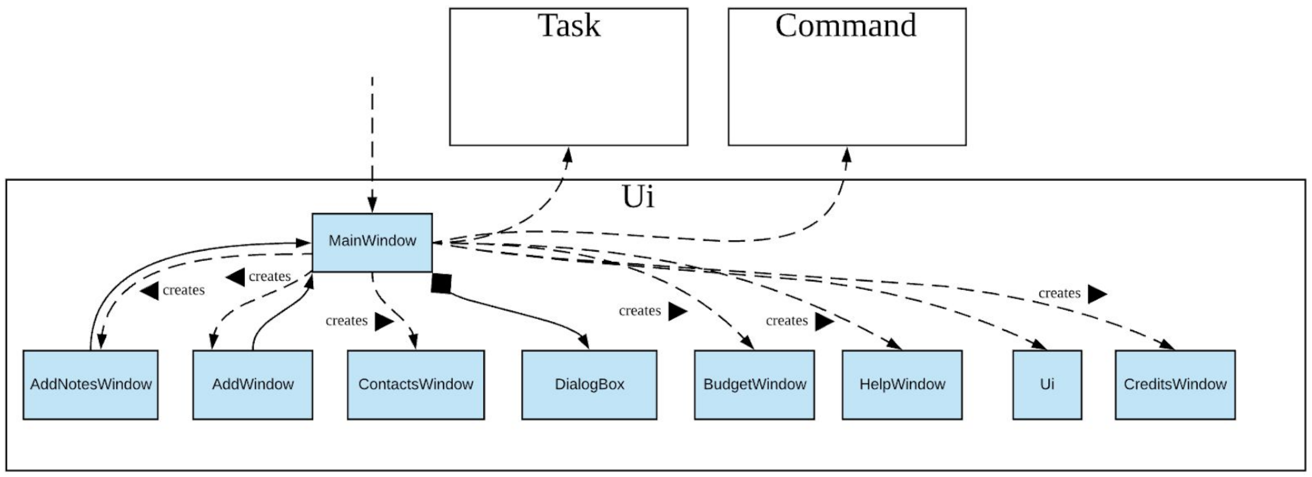


Figure 13. Interaction of Ui with other classes

In the UI component, it consists of two main classes, MainWindow and Ui.

For class Ui.java, it mainly handles the output of messages to the Gui interface for the user to view.

For class MainWindow, it uses the JavaFx framework which are defined in matching .fxml files which are stored in the src/main/resources/view folder. AddNotesWindow, DialogBox, BudgetWindow, AddWindow and HelpWindow are similarly defined.

3. Implementation

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

3.1. Contacts Feature

The contacts feature is built to allow engineers to manage and store their contacts. They can add, view, find or delete their contact list.

3.1.1. Implementation for contacts feature

The following is an activity diagram to show the execution of all events.

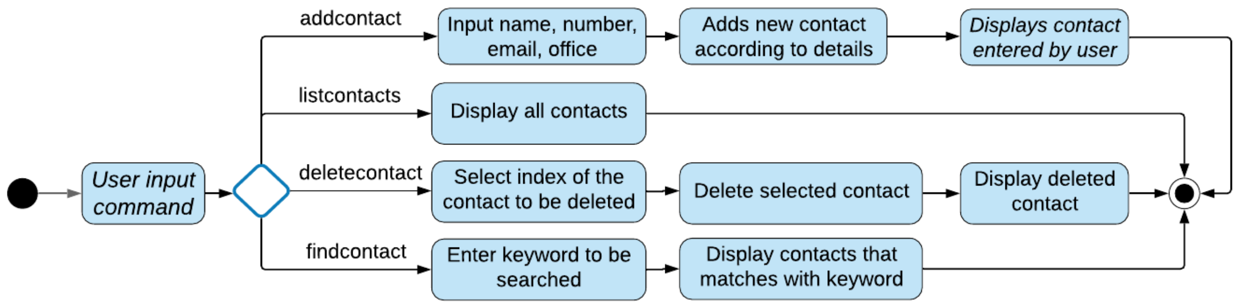

addcontact
listcontacts
deletecontact
findcontact
User input command
Input name, number, email, office
Adds new contact according to details
Displays contact entered by user
Display all contacts
Select index of the contact to be deleted
Delete selected contact
Display deleted contact
Enter keyword to be searched
Display contacts that matches with keyword

Figure 14. Activity diagram of contacts feature

1) addcontact()

The addcontacts() feature allows the user to store details about a person's name, number, email and office location.

The following sequence diagram shows how addcontact() operation works:

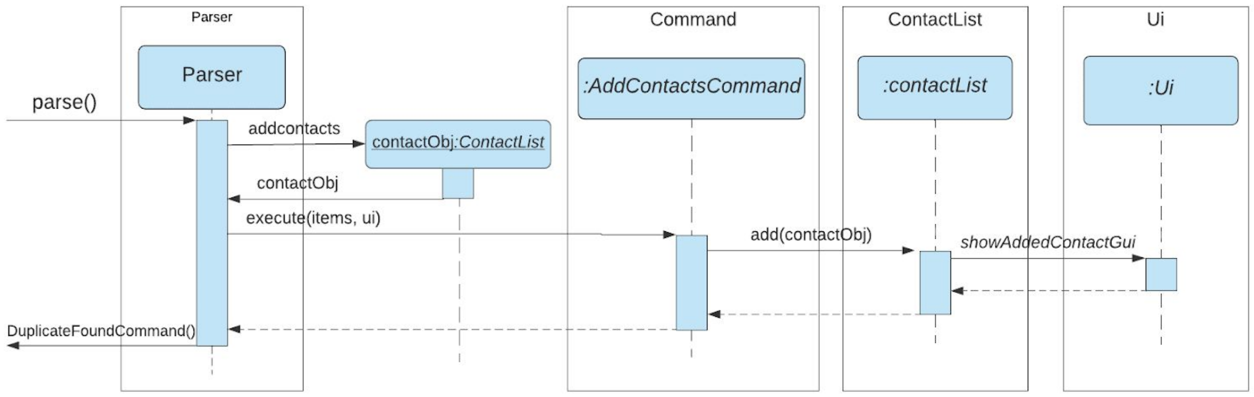


Figure 15. Sequence diagram of addcontact()

Step 1. Duke Manager is launched for the first time. It attempts to read data stored in the ContactStorage() and since the file to store the contacts is not found, Duke Manager automatically creates an empty file. It then waits for user to command addcontact before executing the function respectively.

Step 2. The user executes addcontact command and inputs the contact details to be added. The Parser passes the details into Contacts() and retrieves contactObj which is then passed into AddContactsCommand(). AddContactsCommand() performs the executeGui() method which adds the contactObj into a list and shows the user the contact that has just been added.

2) listcontacts()

The listcontacts() feature allows the user to view all stored contacts available in Duke Manager.

The following sequence diagram shows how the listcontacts() operation works:

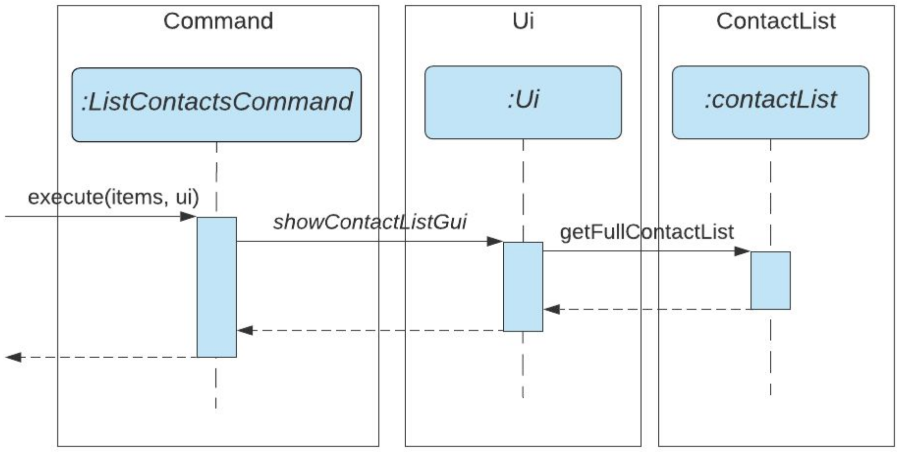


Figure 16. Sequence diagram of listcontacts()

Step 1: Duke Manager waits for an input by the user. If listcontacts is entered, Parser calls for ListContactsCommand() class which then calls out the executeGui() method.

Step 2: In executeGui(), the showContactList() method in Ui is called to display the contacts list.

3) findcontact()

The findcontact() feature allows the user to quickly find a contact using a keyword, this saves the trouble of having to navigate through the list and finding a specific contact.

The following sequence diagram shows how the findcontact() operation works:

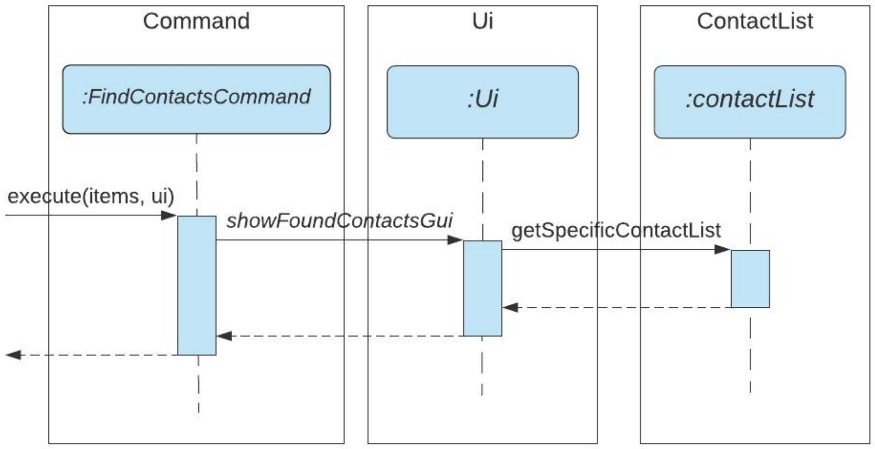


Figure 17. Sequence diagram of findcontact()

Step 1: User inputs a keyword to be searched across all contacts. Parser converts the keyword into lowercase and passes them into FindContactCommand() class which calls out execute() method.

Step 2: The executeGui() method calls out showFoundContacts() method inside the Ui class. In the method, it extracts the details from the contactList getOnlyDetails() method and removes the punctuation and converts them to lowercase. It then proceeds to find keywords that matches inside the list.

Step 3: If there are no contacts found, it will display that no matching tasks were found, else, it will show the user a list of contacts found to be matched.

4) deletecontact()

The deletecontact() feature allows the user to remove a specific contact from its contact list.

The following sequence diagram shows how the deletecontact() operation works:

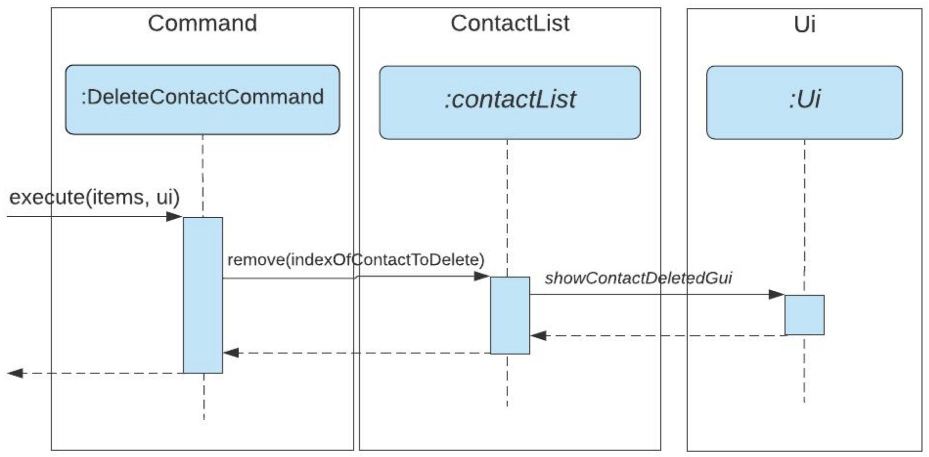


Figure 18. Sequence diagram of deletecontacts()

Step 1: User inputs an index to delete a contact. Parser passes the index into DeleteContactCommand() class which calls out executeGui() method.

Step 2: The execute() method calls out getAndDisplay() method inside the contactList class to show the user the contact that has been removed. It then proceeds to remove it by calling remove() in the contactList class. If there are no contacts or an invalid input has been detected, it will display an error message accordingly.

3.1.2. Design Considerations for Contacts

Aspect: How the command for deleting contact is executed

- Alternative 1 (Current Choice): Checks for any error before going to the main code which deletes and displays the contact removed.

    - Pros: This uses the code quality of “make the happy path prominent” and gives a very clear view of the execution path for a successful scenario.
    - Cons: Multiple return statements are needed.
- Alternative 2: Use if-else to check if the conditions are met else it will check for the mistake and display the error message accordingly.
    - Pros: The main execution is at the top of the code.
    - Cons: Method is longer and looks more cluttered due to multiple if-else statements.

3.2. Backup Feature

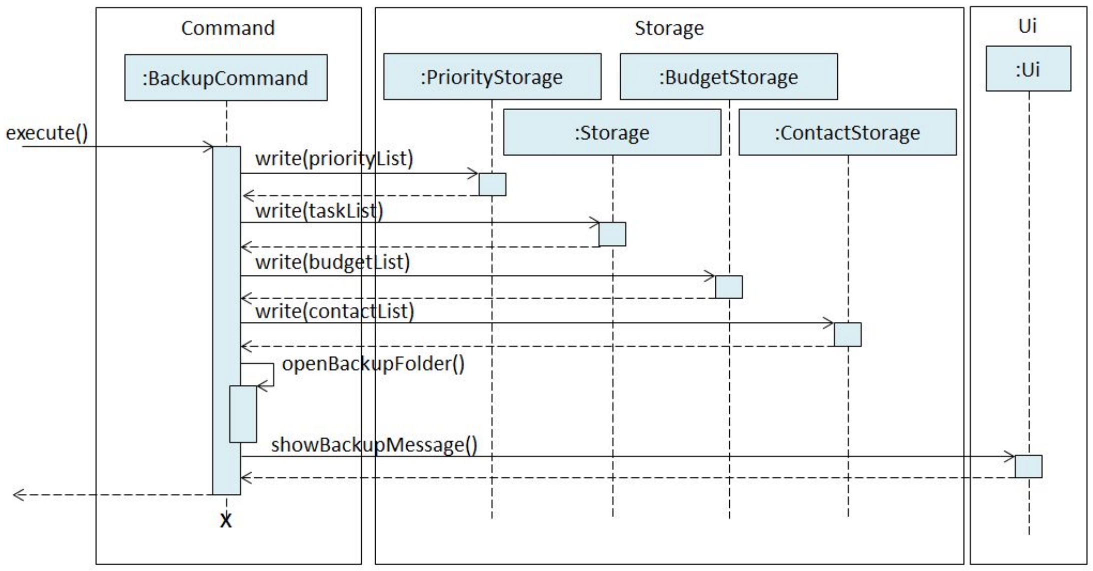
The backup feature will allow the user to manage the data files of Duke Manager. This will save the current state of the application, and launch the file location in the user’s file explorer.

3.2.1. Implementation of Backup feature

The feature is activated by the user entering “backup” into Duke Manager. Upon entering the class BackupCommand, the following steps will be done:

1) It will write the current state of the lists in Duke Manager by calling storage.write(), priorityStorage.write(), budgetStorage.write() and contactStorage.write().
2) After writing successfully into the files, it will call the method openBackupFolder, which opens the directory of the saved data files in file explorer.
3) After opening the file, it will call showBackupMessage, where Duke Manager shows the user that backup is complete and that the file location is opened by Duke Manager.

Below is the sequence diagram for the backup feature.



3.2.2. Design Considerations for backup feature

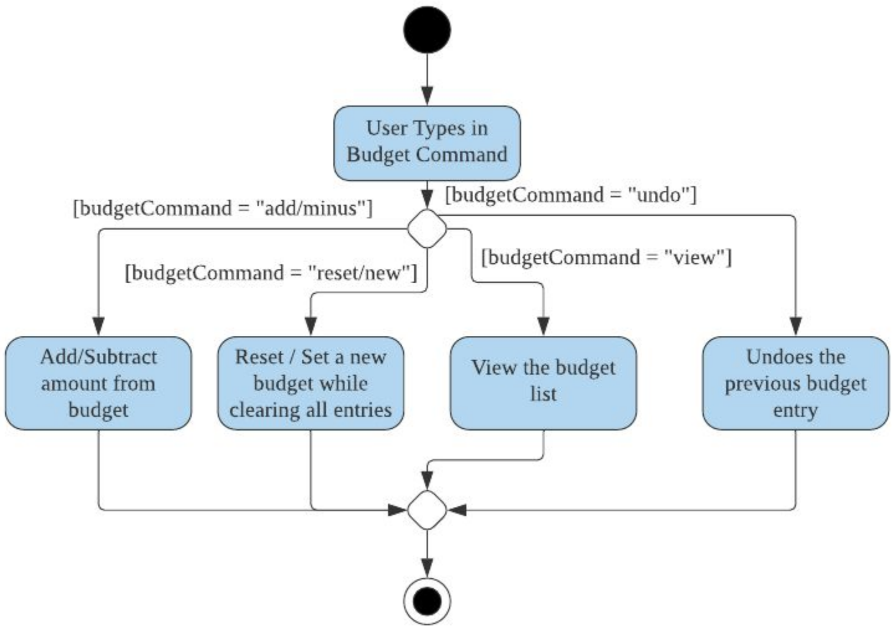
<u>Aspect: Process of backing up.</u>
As the normal case of backing up is for the importing / exporting of the data in Duke Manager, it is more appropriate to have the ease of opening the file location immediately after the command backup, instead of manually trying to find where the files are at after typing backup. Therefore it was implemented so that upon pressing backup, the user can have access to the data files directly.

3.3. Budget Feature

Duke Manager also has a built in budget storing feature for engineers to monitor their budget. They can add or subtract from their budget, as well as view and reset their budget.

3.3.1. Implementation

The following diagram shows the activity diagram of the process of calling the budget in Duke Manager.

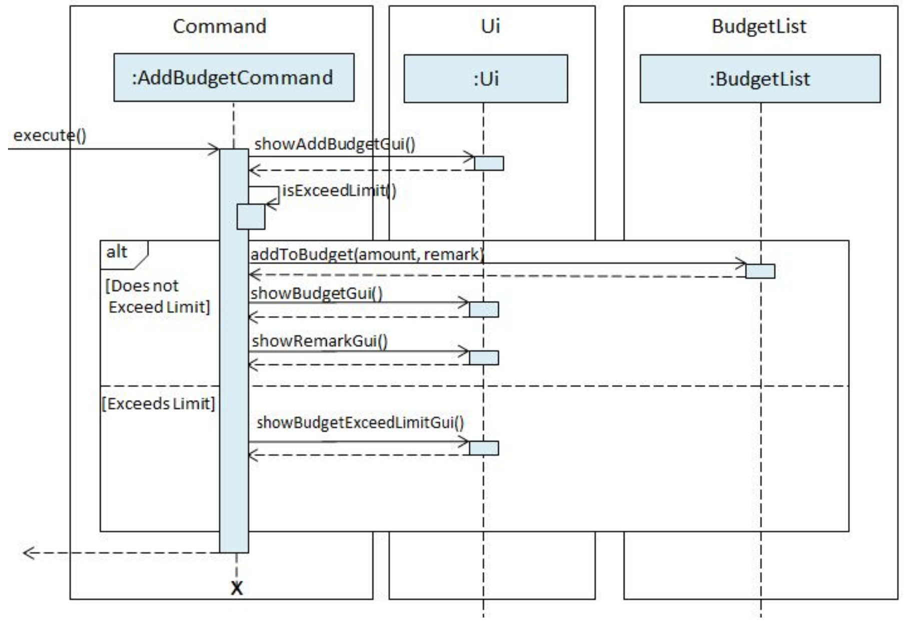

User Types in Budget Command
[budgetCommand = "add/minus"]
[budgetCommand = "undo"]
[budgetCommand = "reset/new"]
[budgetCommand = "view"]
Add/Subtract amount from budget
Reset / Set a new budget while clearing all entries
View the budget list
Undoes the previous budget entry

The feature is triggered with the starting word “budget”. It will then check for the next word in the string command, whether it is one of the valid terms such as “view” and “new”. Upon validating, it will enter the different Commands (AddBudgetCommand, ViewBudgetCommand, ResetBudgetCommand, UndoBudgetCommand) where it will perform the necessary actions for the stated command.

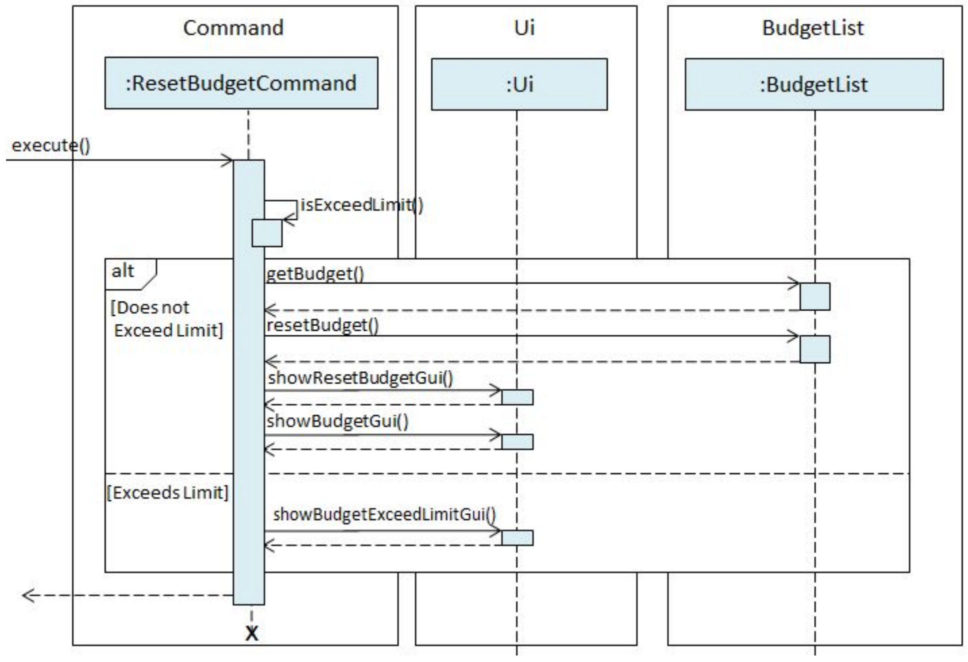
The feature is triggered with the starting word “budget”. It will then check for the next word in the string command, whether it is one of the valid terms such as “view” and “new”. Upon validating, it will enter the different Commands (AddBudgetCommand, ViewBudgetCommand, ResetBudgetCommand, UndoBudgetCommand) where it will perform the necessary actions for the stated command.

There are 3 cases to consider, alongside their sequence diagram:

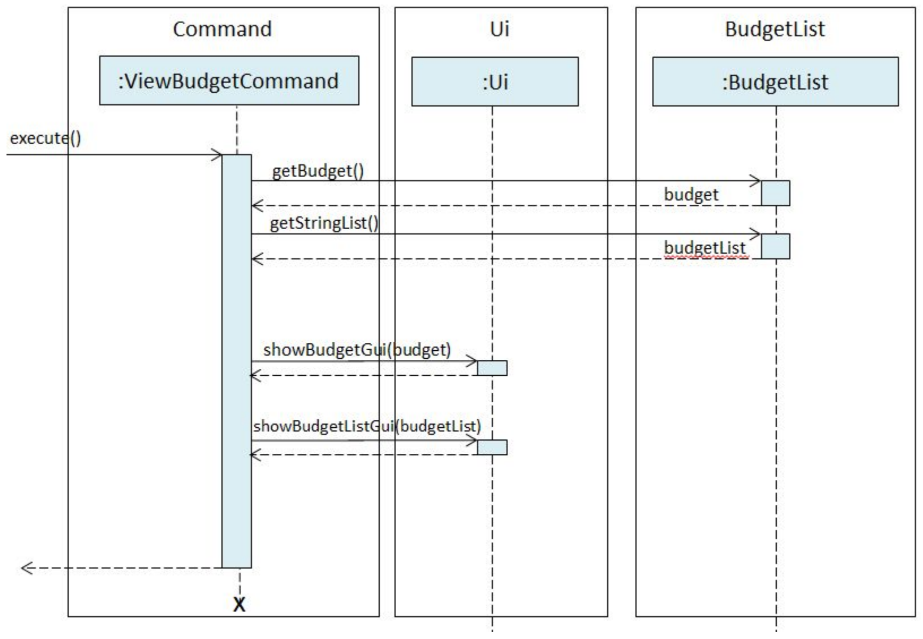
1) Adding/Subtracting from budget: Upon receiving the command, Duke Manager will enter the class AddBudgetCommand(), and shows the user the amount before adding, and after adding. It will also store the change to the existing BudgetList



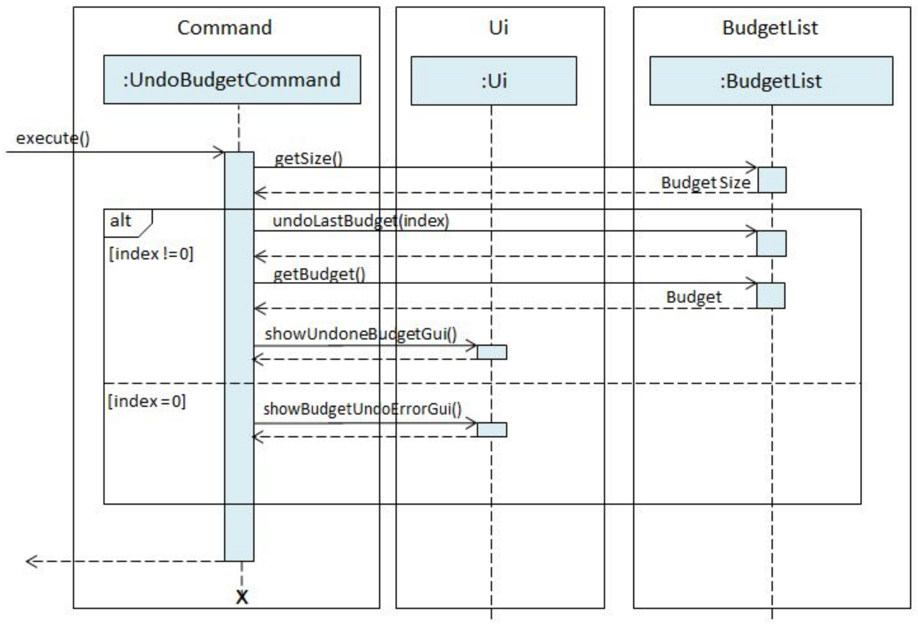
2) Reset/New budget: Upon receiving the command, Duke Manager will show the user its previous budget, update the budget with the input amount inside BudgetList and then show the user the new budget.



3) View budget: Upon receiving the command, Duke Manager will show the budget to the user by calling showBudget().



4) Undo budget: Upon receiving the command, Duke Manager will proceed to remove the latest budget in the list from the stack, recalculate the budget and then show the updated budget to the user.



3.3.2. Design considerations

Aspect: Simplicity
The choice of implementing just the addition, subtraction, resetting and undoing of budget is due to the consideration of making the budget tracking aspect more intuitive and user friendly. Due to the fast pace of our academic life, noting down of how much we spent should not be tedious, but straightforward and easy. Therefore a simple input for the amount would be ideal. The description is made optional to give a choice for users who do not require the details to be input.

3.4. Update function

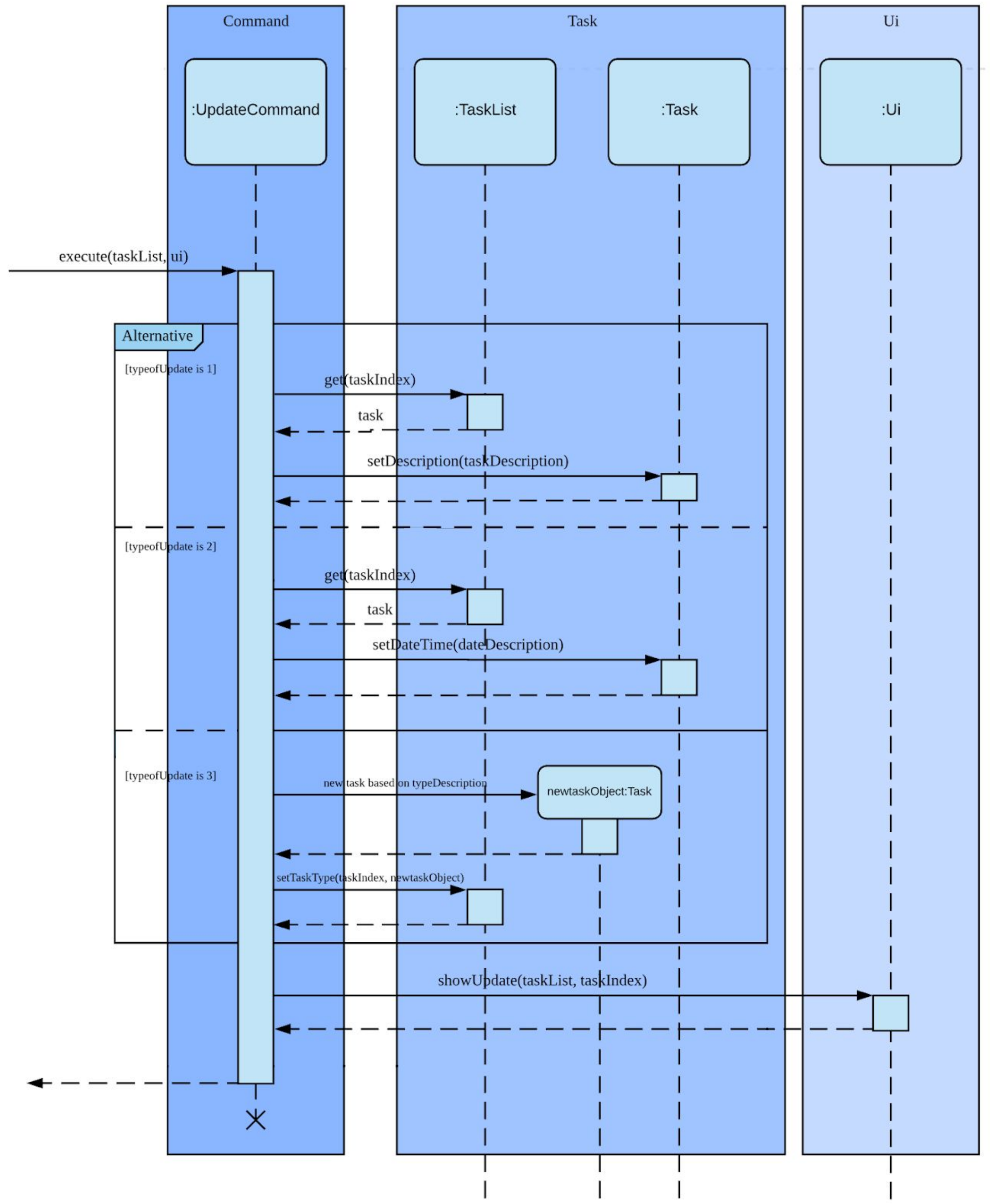
The update function will allow the user to update the existing tasks in the task list of Duke Manager. This will allow the user to update either task description, date and time of the task, or type of task.

3.4.1. Implementation

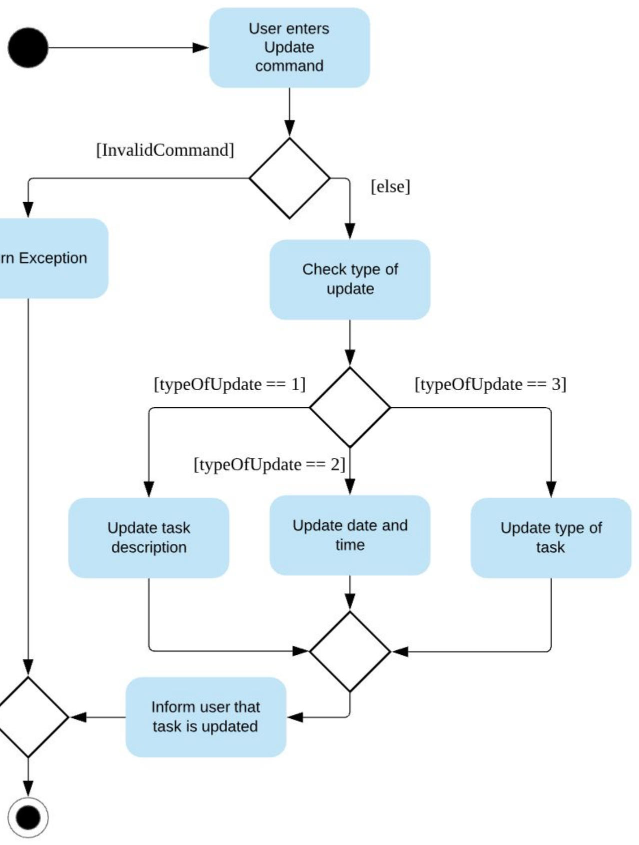
Upon entering the class UpdateCommand, the following steps will be done:

1. If the typeofUpdate equals to 1, this means that it is updating the task description. It proceeds to call get(taskIndex), where taskIndex refers to index of task, to retrieve the task to be updated. Next, it calls setDescription(taskDescription), where taskDescription refers to the new task description, to overwrite the description of the retrieved task.
2. If the typeofUpdate equals to 2, this means that it is updating the date and time of the task. It proceeds to call get(taskIndex), where taskIndex refers to index of task, to retrieve the task to be updated. Next, it calls setDateTime(dateDescription), where dateDescription refers to the new date and time in string format, to overwrite the date and time of the retrieved task.
3. If the typeofUpdate equals to 3, this means that it is updating the type of task. It proceeds to create a new task object based on the typeDescription, where typeDescription refers to the new type. Next, it calls setTaskType(taskIndex, newtaskObject),where taskIndex refers to index of task and newtaskObject refers to the task object that was created moments ago, to overwrite the type of the task.
4. After either Step 1, 2, or 3 is completed, it then calls showUpdate(taskList, taskIndex), where taskList refers to the task list and taskIndex refers to index of task, to tell the user that the update has been completed.

Below is the sequence diagram of updating task.



The following diagram shows the activity diagram of the process of calling the update command in Duke Manager.



3.4.2. Design Considerations

Aspect: Flexibility of update

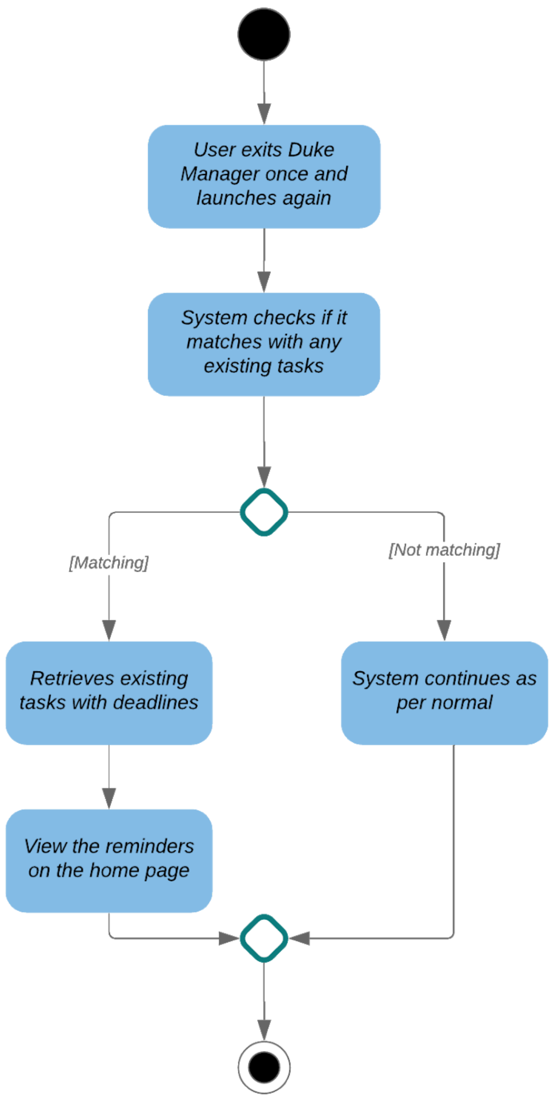
- Alternative 1 (current choice): Updates either task description, date and time, or task type, one at a time.
    - Pros: Easy to implement.
    - Cons: Can be very inefficient if the user wants to execute two or more types of update of an existing task.
- Alternative 2: Updates up to all types of update in one command.
    - Pros: Very efficient for the user when updating many types of update of an existing task.
    - Cons: Difficult to implement, requires complex algorithm.

3.5. Reminders Feature

The reminders feature will allow the user to retrieve the existing tasks with a due date and time specified in the task list of Duke Manager and display it as a reminder. This will allow the user to be reminded of the task which have a due date in 3 days from today’s date at the home page.

3.5.1. Implementation

The following diagram shows the activity diagram of getting reminders in Duke Manager.



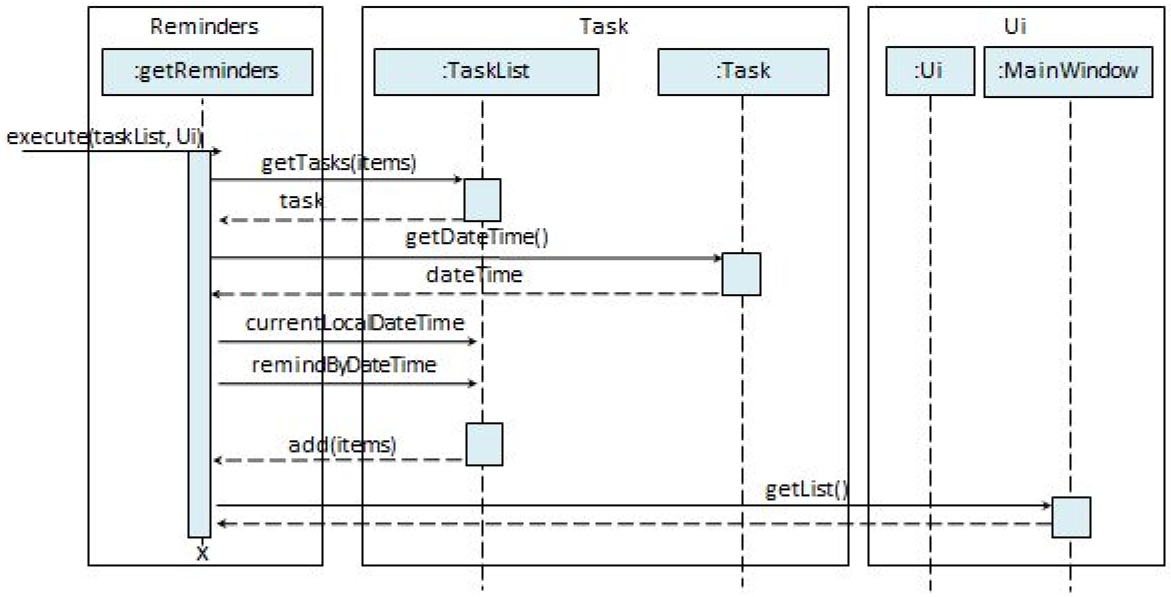
User exits Duke Manager once and launches again
System checks if it matches with any existing tasks
[Matching]
[Not matching]
Retrieves existing tasks with deadlines
System continues as per normal
View the reminders on the home page

The feature is activated upon properly exiting Duke Manager the first time previously and launched again for the second time. Upon entering the class Reminders, the following steps are taken by the application:

The getReminders() method does not require any additional input. However, ensure that there is an existing Priority-Related Task Input (task with a time and date) added previously and this same existing task <date and time> is set to 1-3 days after <todaysDate>.

1. Duke Manager is launched for the second time. It attempts to load for each existing tasks in storing in TaskList() array, which also refer to the directory of the saved data files in file explorer.
2. After successfully retrieving existing tasks with a due date and time, it gets the current date and time where the LocalDateTime is tracked.
3. To set a reminder for the task due in 3 days, it compares the current date and time with the existing task due in that <date and time>.
4. If the task is due after 3 days from the current LocalDateTime and the task is due before the remindByDateTime, the add() method implemented in TaskList() will be invoked.
5. Therefore, at the home page, there will be a section on upcoming reminders where existing tasks in the TaskList() will call the getReminders() method which is set to remind only existing tasks between 1-3 days after today's date.

Below is the sequence diagram of retrieving the list of tasks due in 3 days as reminders.



3.5.2. Design Considerations

Aspect: Importance of being able to remind user of upcoming tasks

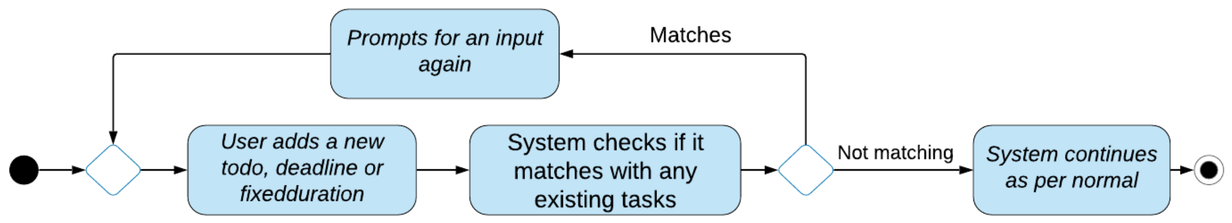
Due to our natural daily lifestyle which consists of tasks and personal goals to finish these tasks by a certain date and time. A reminder feature is useful in this case after user inputs all his tasks and returns back to the program after exiting for the first time. Therefore, it was implemented so that upon proper exiting of Duke Manager through the exit command, the updated progress of all the data will always be saved where it will retrieve tasks which have a due date in 3 days from today's date.

3.6. Detect Duplicates

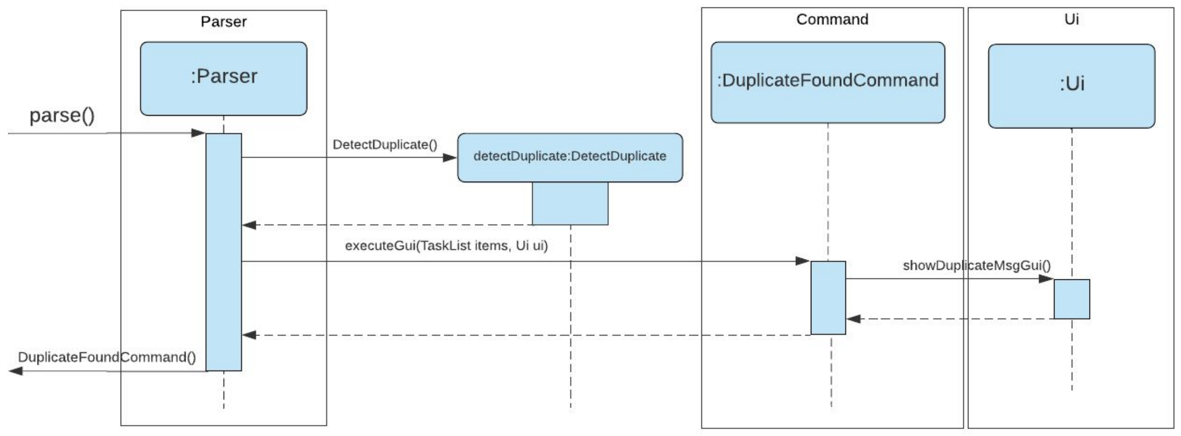
DetectDuplicate() automatically checks if the todo, deadline or fixedduration task added by the user matches with any existing tasks that have already been stored.

3.6.1. Implementation of Detect Duplicates

The following is an activity diagram to show the events when a duplicated task is found.


Prompts for an input again
Matches
User adds a new todo, deadline or fixedduration
System checks if it matches with any existing tasks
Not matching
System continues as per normal

Below is the sequence diagram of Detect Duplicates:



Step 1. When the user enters a command to add a todo, deadline or fixedduration task, DetectDuplicate() is created in the system and TaskList is passed into it.

Step 2. isDuplicate() checks to see if the input matches with any tasks in the TaskList and returns a true value if there is, if not, it will return a false value.

Step 3. If true is sent back to the Parser, DuplicateFoundCommand() is called and displays an alert to show the user, preventing them from adding the same task again. If false is sent back, the respective commands proceed as per normal.

3.6.2. Design Considerations for Detect Duplicates

Aspect: Detecting duplicates across the different types of tasks (Types like todo, deadline and fixedduration)

- Alternative 1 (Current Choice): It will detect the description and alert the user even if the type of task differs.
    - Pros: Able to detect if the same task is added to a different type of tasks.
    - Cons: Users are not able to add the same task across the different types of input.
- Alternative 2: Detect duplicate to not activate if the same description is found in the different types of input.
    - Pros: Allows users to store the same description in for example todo and deadline.
    - Cons: Calling the detect duplicate method will look lengthier and users may get confused with their own input as for example, the task in deadline has a time but the same task in todo does not. Making hard to spot the difference in the tasks.

3.7. Detect Anomalies

When the user adds a new Deadline task or updates an existing task, the DUKE manager checks if there is any task that has the same date and time in the task list. If found, it will tell the user to re-enter the command with a new date and time, or mark the existing task as done first.

3.7.1. Implementation

Upon entering the class Parser, the following steps will be done:

1. If the command is either adding Deadline task or updating task's date and time, Detect Anomalies will be activated.
2. Then, it proceeds to do a linear search in task list. It compares the dateDesc, where dateDesc refers to date and time description, with taskList.get(i).getDateTime(), where get(i) refers to the task from the task list based on index i, and getDateTime() refers to the task's date and time.

3. Furthermore, Detect Anomalies also checks if the task is marked as done. It returns an exception if there is an existing task with the same date and time and is not marked as done.

3.8. Notes

The user is able to have notes attached to existing tasks in the task list of Duke Manager. The user can either add or update notes, remove, or to display notes.

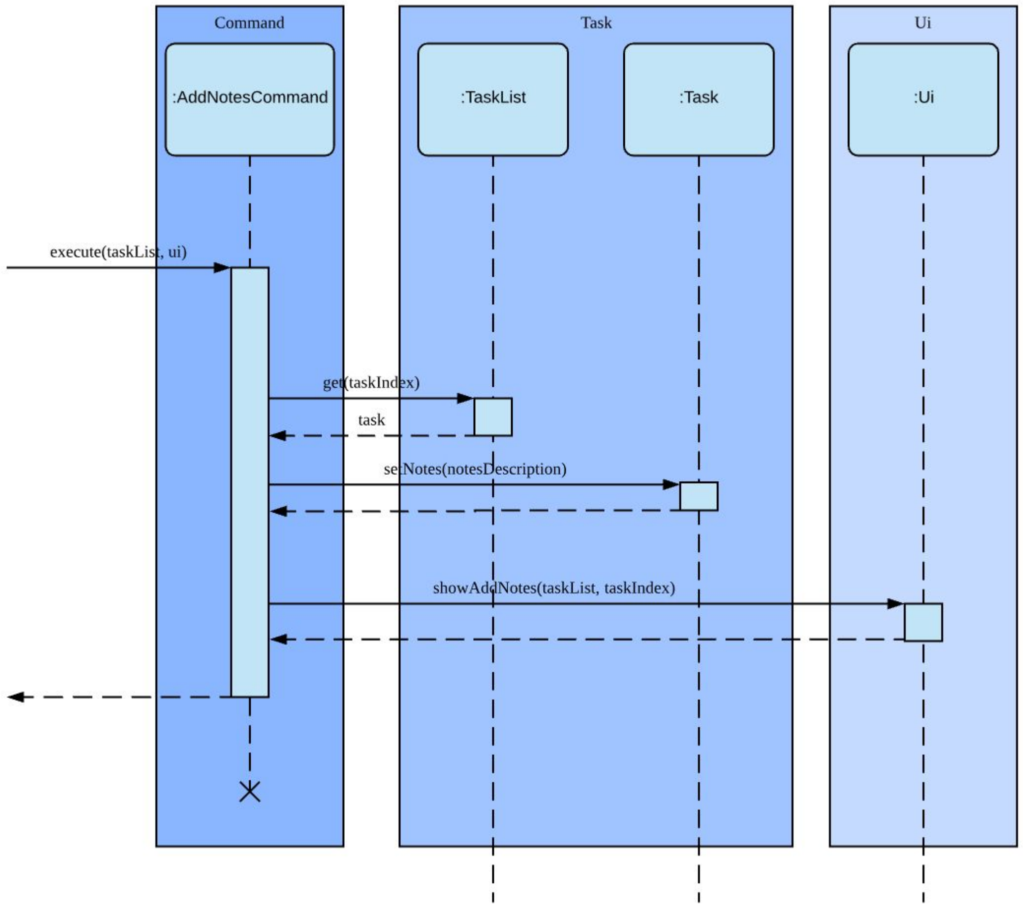
3.8.1. Implementation

There are three commands related to Notes: adding (or updating) notes, deleting notes, and showing notes. The reason why adding notes is treated as updating, is that the notes description is integrated to each and every task. When notes is added to a task, it is actually updating the notes description of the task, instead of creating a new note, hence the ambiguity.

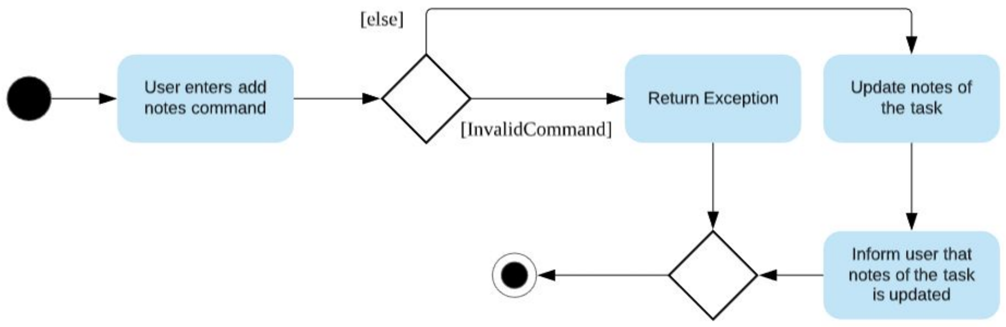
1. Adding/Updating notes: Upon entering the class AddNotesCommand, the following steps will be done:

4. It proceeds to call get(taskIndex), where taskIndex refers to index of task, to retrieve the task. Next, it calls setNotes(notesDescription), where notesDescription refers to the notes description, to overwrite the notes of the retrieved task.
5. Then, it calls showAddNotes(taskList, taskIndex), where taskList refers to the task list and taskIndex refers to index of task, to tell the user that the adding or updating of notes has been completed.

Below is the sequence diagram of adding (or updating) notes.



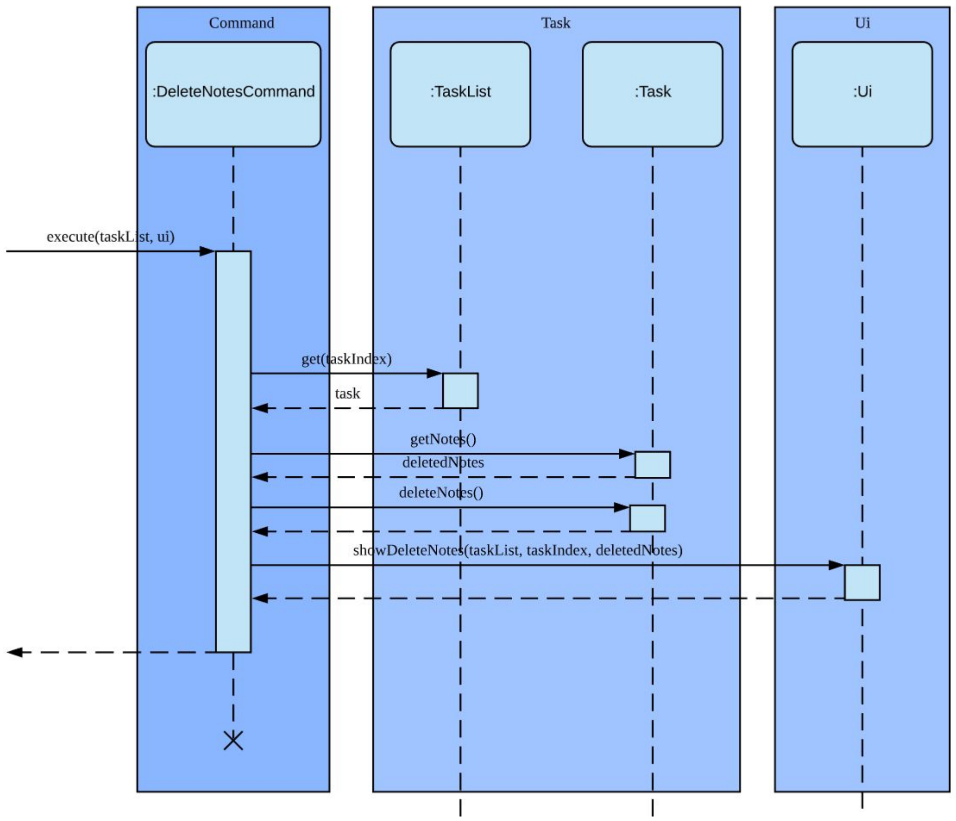
The following diagram shows the activity diagram of the process of adding (or updating) notes Duke Manager.


User enters add notes command
[else]
[InvalidCommand]
Return Exception
Update notes of the task
Inform user that notes of the task is updated

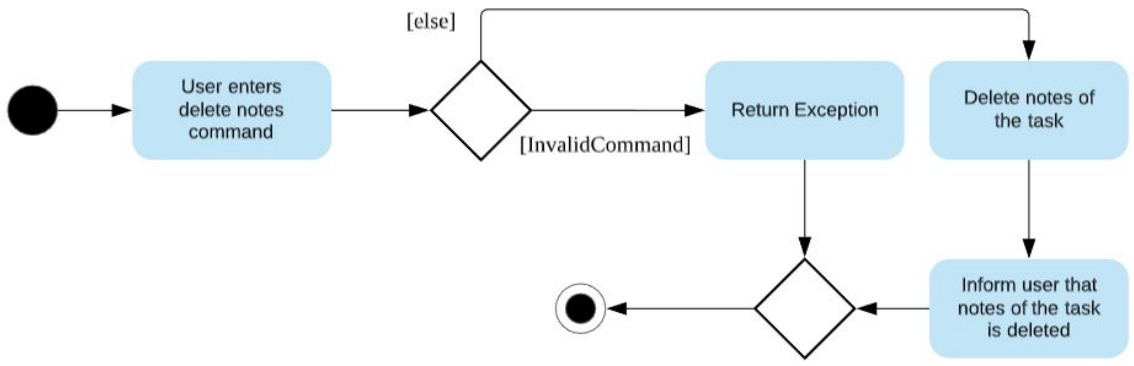
2. Deleting notes: Upon entering the class DeleteNotesCommand, the following steps will be done:

1. It proceeds to call get(taskIndex), where taskIndex refers to index of task, to retrieve the task. Next, it calls getNotes(), to retrieve the notes of the task to be deleted.
2. Then, it calls deleteNotes(), to delete the notes by setting it to “empty” on notes description of the task.
3. Next, it calls showDeleteNotes(taskList, taskIndex, deletedNotes), where taskList refers to the task list, taskIndex refers to index of task, and deletedNotes refers to the previous notes that have been deleted from the task, to tell the user that the deleting of notes has been completed.

Below is the sequence diagram of deleting notes.



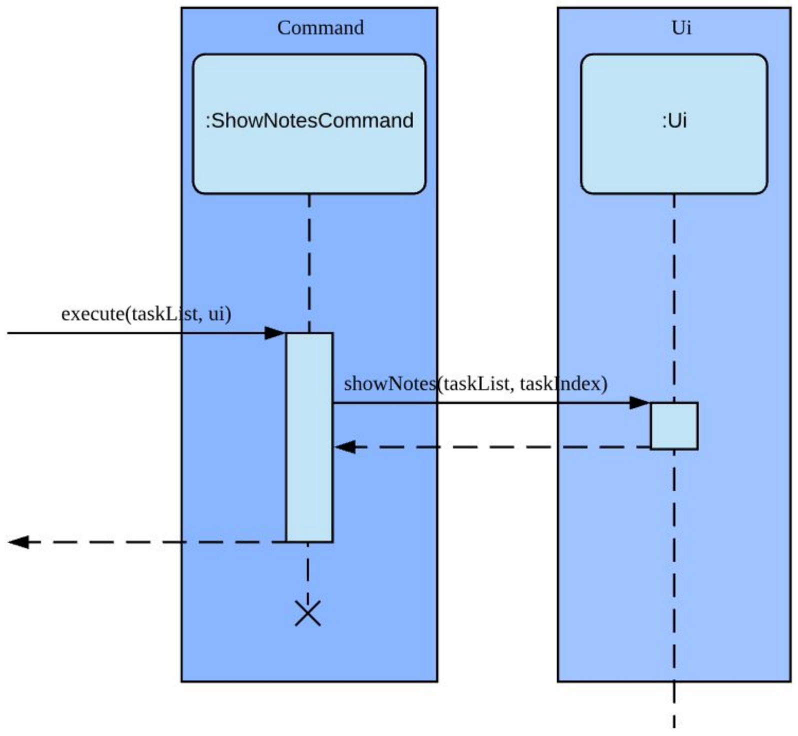
The following diagram shows the activity diagram of the process of deleting notes Duke Manager.


[else]
User enters delete notes command
[InvalidCommand]
Return Exception
Delete notes of the task
Inform user that notes of the task is deleted

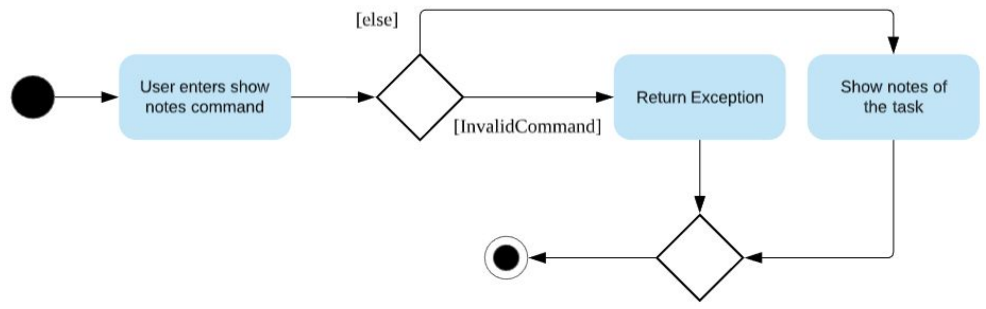
3. Showing notes: Upon entering the class ShowNotesCommand, the following steps will be done:

1. It calls showNotes(taskList, taskIndex), where taskList refers to the task list, taskIndex refers to index of task, to display the notes of the task to the user.

Below is the sequence diagram of showing notes.



The following diagram shows the activity diagram of the process of showing notes Duke Manager.



3.8.2. Design Considerations

Aspect: Types of notes

- Alternative 1: Make notes as a separate entity. There is no relation between tasks and notes.
    - Pros: Can have as many notes as the user wants.
    - Cons: As the user is unable to tag notes to the tasks, it can be very messy if there are many tasks and notes.
- Alternative 2 (current choice): Make notes to be linked with tasks.
    - Pros: Very simple for the user when adding, deleting, or showing notes by using the task's index.
    - Cons: Can only have one note per task.

3.9. History [V2.0]

The history function will allow the user to retrieve a list of commands that were entered previously. By pressing up or down arrow keys, the user can traverse through the previous commands. When history command is entered, it passes a string that contains "history" to HistoryCommand. When up or down arrow key is pressed instead, it passes a string that contains "up" or "down" to HistoryCommand.

3.9.1. Implementation

Upon entering the class HistoryCommand, the following steps will be done:

1. If the string contains "history", this means that it is calling the list of previous commands. It proceeds to call showHistoryList(), to display the list of previous commands to the user.
2. If the string contains "up", this means that it is traversing backwards, where the first command is the oldest, and the last command is the latest. It proceeds to deduct previousCommandIndex by one, where previousCommandIndex refers to the index of the list of previous commands.
3. If the string contains "down", this means that it is traversing forward, where the first command is the oldest, and the last command is the latest. It proceeds to add previousCommandIndex by one, where previousCommandIndex refers to the index of the list of previous commands.

4. After either Step 2 or 3 is executed, it proceeds to call showpreviousCommand(previousCommandIndex), to display the command to the user.

If the previousCommandIndex has reached the index of either oldest or latest command, it would not deduct to a negative number nor add to a number that exceeds the size of the history list.

3.10. Logging

We are using java.util.logging package for logging. The DukeLogger class is used to manage the logging levels and logging destinations.

- The Logger can be obtained using DukeLogger.getLogger(Logger.GLOBAL_LOGGER_NAME) which will log messages according to the specified logging level.
- Currently log messages are output through: Console and to a .log file.

Logging Levels

- SEVERE : Critical problem detected which may cause the application to not work as intended. There is a chance of crashing the application.
- WARNING : Take note of the error, the application is still working properly.
- INFO : Information showing some actions done by the application to the user, but unable to display through GUI.

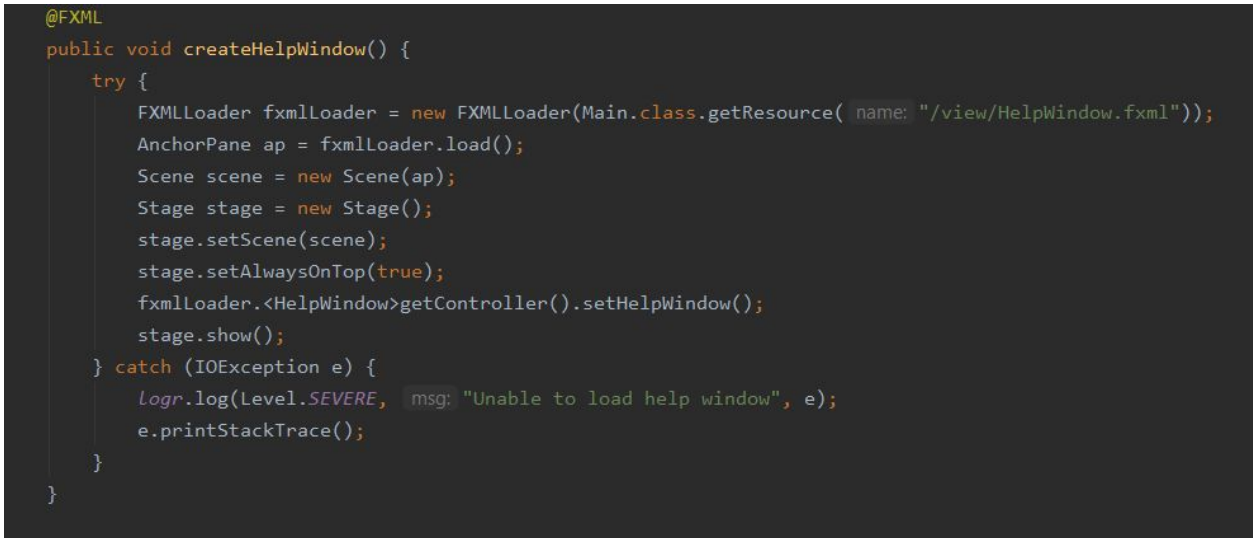
3.11. Help

This feature allows users to view all available commands in Duke Manager.
It comprises of the following functions: Selecting a command, followed by a short description, format and an example of a particular command.

3.11.1. Implementation

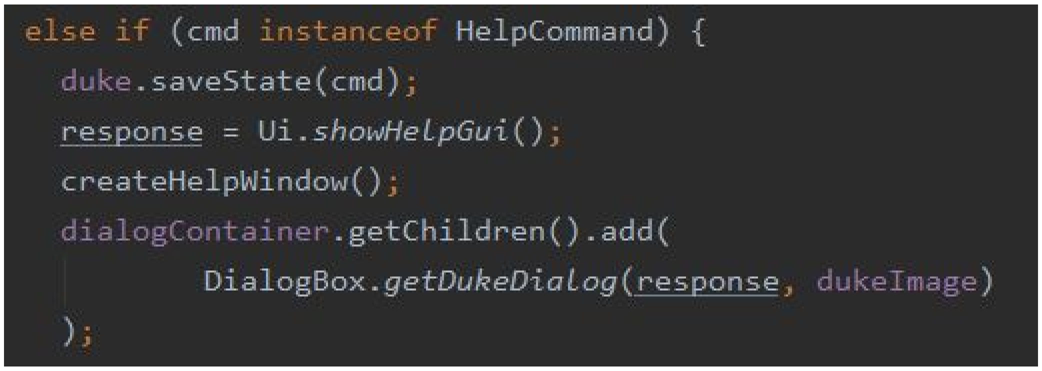
The help Function is facilitated by the MainWindow from the UI Component. A drop-down list to select a command which requires help is created in this process.

The exact implementation is shown below:



```
@FXML
public void createHelpWindow() {
    try {
        FXMLLoader fxmlLoader = new FXMLLoader(Main.class.getResource( name: "/view/HelpWindow.fxml"));
        AnchorPane ap = fxmlLoader.load();
        Scene scene = new Scene(ap);
        Stage stage = new Stage();
        stage.setScene(scene);
        stage.setAlwaysOnTop(true);
        fxmlLoader.<HelpWindow>getController().setHelpWindow();
        stage.show();
    } catch (IOException e) {
        logr.log(Level.SEVERE,  msg: "Unable to load help window", e);
        e.printStackTrace();
    }
}
```

When the help command is entered, the MainWindow from the UI Component prompts the user-friendly interface of help window so that the user can navigate freely to be aware of the functions available in Duke Manager and what each command specifically does.



```
else if (cmd instanceof HelpCommand) {
  duke.saveState(cmd);
  response = Ui.showHelpGui();
  createHelpWindow();
  dialogContainer.getChildren().add(
          DialogBox.getDukeDialog(response, dukeImage)
  );
```

3.11.2. Design Considerations

Aspect: Types of help

- Alternative 1: Dynamic help command to suggest format when help followed by a space is being entered.
    - Pros: Can dynamically retrieve help instantly when unsure of a command format
    - Cons: As the user is unsure of the system, this method might not allow user to be aware of the functions available in Duke Manager and what each command specifically does.
- Alternative 2 (current choice): Help window pops up with a friendly user interface for user to fuss-free navigation and usage of Duke Manager.
    - Pros: Very simple for the user to receive help for a command format he/she is unsure of

- Cons: View the entire description of one single command one at a time.

4. Documentation

4.1. Document Creation

We used Google Drive (Google Docs) to create the Developer Guide and User Guide as it was relatively user friendly and coupled with the sharing and editing capabilities, our team preferred it over .adoc and .md (Mark down).

Conversion to PDF is done by downloading the guide as .pdf format which can be found under File > Download > PDF Document (.pdf).

4.2. Diagram Creation

Editing of the diagrams is done by mainly using LucidChart. LucidChart provides a free alternative to creating diagrams with templates that can then be copied and pasted onto our developer guide. In addition, we referred to the topics covered during lecture to understand how the diagrams are created.

5. Testing

5.1. Running Tests

Tests can be conducted by utilising IntelliJ's Junit test runner. Right click on src/test/java and click on "Run tests in duke.test". To run an individual test or group of tests, just right click on the individual tests/test packages and choose "Run test in ______"

5.2. Running Checkstyle test

Basic format tests can be checked by using the checkstyle command in gradle. To run the checkstyle test, open the terminal in intellij and input the command "gradlew checkstyleMain checkstyleTest". To test only the Main or Test package's checkstyle, you can do "gradlew checkstyleMain" or "gradlew checkstyleTest" respectively. If this is not done in the terminal of intellij, navigate to the root folder in your respective terminal. As long as there is one error, the code will report a "build failed". To see the report, navigate to build > reports > checkstyle to check and fix the error stated before re-entering the command.

6. Dev Ops

6.1. Build Automation

Using gradle, a build automation tool, we used it to automatically build tasks such as:

- JUnit Tests
- Style compliance analysis
- JavaFX

The configurations for gradle can be found under `build.gradle`.

To run a Gradle command, open the terminal in IntelliJ and enter the Gradle command. Gradle commands look like this:

- On windows: `gradlew run`
- On MAC/Linux: `./gradlew run`

Gradle instructions that can be used in our project

- `gradlew run`
  This runs the program.
- `gradlew test`
  This runs all the test.
- `gradlew checkstyleMain checkstyleTest`
  This checks the code for any inconsistency in the styles of code writing.

6.2. Continuous Integration

We used Travis CI to perform *Continuous Integration* on our projects. For setting up, please refer to section 1.4.

6.3. Making a Release

Appendix A : Project Scope

Introduction

<u>User profile:</u>

Our target user are Engineering Undergraduates in NUS, especially those who:

- has a need to manage a large number of school-related tasks such as todo and deadline
- prefer desktop apps over other types
- can type fast
- is reasonably comfortable using CLI and GUI apps

- is currently studying a course related to engineering
- likes to be efficient

Problem addressed: As an engineering student, there can be many different commitments to handle and things may be confusing at times. The academic handbook given by Student Union (NUSSU) might not be handy for us as most of us use laptops. Therefore our team is aiming to create a digital academic handbook to address the problem of having multiple tasks and deadlines in general.

Societal impact: The impact of our handbook is targeted mainly to ease the workload of engineering undergraduates like our team.

Value Proposition: Our application aims to allow students to manage school-related tasks better than an average task scheduler app by optimizing our initial program created in Phase 1 of this module.

Appendix B : User Stories

Our team has categorized the user stories to their priorities:

★★★ High: Essential to have

★★ Medium: Good to have

★ Low: Unlikely to have

Priority	As a ...	I want to ...	So that I can...
★ ★ ★	engineering undergraduate	store multiple tasks	remind myself to complete them any time.
★ ★ ★	engineering undergraduate	store multiple deadlines	complete them before deadline.
★ ★ ★	engineering undergraduate	store multiple events	keep track and attend the events accordingly without forgetting about them.
★ ★ ★	engineering undergraduate	set a list of goals for myself	be motivated to strive.
★ ★ ★	user	have an undo function	undo in case i typed in something erroneous.
★ ★ ★	user	be able to mark a task as done	keep track of my progress.
★ ★ ★	user	be able to delete a task	erase unnecessary tasks
★ ★ ★	user	have a backup for my handbook	recover it in case I delete this program

★ ★ ★	user	be able to sort the input entries by priority	know which tasks to complete first.
★ ★ ★	user	update the type of existing tasks	not have to delete an old task and create a new task with a different type.
★ ★ ★	user	update date and time of my existing tasks	not have to recreate the same task with date and time being different only.
★ ★ ★	user	update task description time of my existing tasks	not have to delete an old task and create a new task.
★ ★ ★	engineering undergraduate	be able to record contact details of my professor	find them for consultation.
★ ★ ★	user	set the priority of existing tasks	can remind myself of the importance of these tasks.
★ ★ ★	user	be able to set a reminder for a specific task	can create a reminder of a task which is die in how many days specified.
★ ★ ★	user	be able to find tasks by priority	can selectively complete all tasks.
★ ★ ★	user	be able to sort tasks by priority	can clearly know which task to finish first.
★ ★ ★	user	be able to find tasks by date	can know what tasks I have on a certain date.
★ ★ ★	user	update task description of my existing tasks	choose not to delete an old task and create a new task

★ ★ ★	User	find tasks by date	know what tasks I have on a certain date
★ ★ ★	Engineering undergraduate	be able to record contact details of my professor	find them for consultation.
★ ★ ★	user	delete contacts stored	remove outdated contacts.
★ ★ ★	engineering undergraduate	keep track of how much I spent in school.	keep track of how much I spent in school.
★ ★	user	find tasks with a certain priority	selectively finish my tasks
★ ★	user	find my contact by their name	quickly access their contact information.
★ ★	engineering undergraduate	receive notifications 3 days before deadline is due,	know the highest priority or urgent tasks to be completed.
★ ★	user	have access to backup data	I can retrieve it for import/export purposes.
★ ★	user	be able to delete multiple tasks	not have to repeat the delete action multiple times.
★ ★	user	be able to undo multiple tasks	not have to repeat the undo actions many times.
★ ★	user	insert notes to my existing task	remind myself on what I need to do for the task.

★ ★	user	have an interactive tutorial of the features of Duke Manager	better understand the product
★ ★	user	I want to see the list of commands	refer to them in case I forget them due to long time no use
★ ★	user	have shorter commands	can type in my tasks faster and more efficiently.
★ ★	user	be able to keep track of my examinations	remember to revise them
★ ★	user	assign command shortcuts	type in my tasks faster
★ ★	user	create a list of ideas	implement them progress
★ ★	user	create appointments	be aware of what appointments (medical, interview etc) I have planned.
★ ★	user	categorize the tasks to different categories	have an overview on a specific category on what is to be done.
★ ★	engineering undergraduate	ensure that I do not enter the same task twice	would not waste time recalling if I had actually done the task.
★	engineering undergraduate	have a birthday reminder at least 2 weeks before	remember important dates about people I care.
★	engineering undergraduate	include emergency contacts	know that people will know who to call during an emergency.

★	engineering undergraduate	keep track of my module grades	keep track of my current university progress
★	engineering undergraduate	keep track of my module credits	keep track of my current university progress
★	user	be able to use a command to enter the current date and time	choose not to refer to my clock and write out the time
★	user	be either in a viewer mode or admin mode	choose not to accidentally edit my stored tasks/notes in the handbook
★	user	be able to copy tasks in my handbook	paste them in my calendar/notes/other parallel todo trackers.
★	user	have a password system	ensure that only those who knows the password can have access to my handbook

Appendix C : Use Cases

For all use cases, the System is Duke Manager (DM) and the Actor is Student unless otherwise specified.

Use Case: Show command tutorial

MSS

1. Student launches Duke Manager.
2. Student requests to show tutorial on commands.
3. DM shows the command tutorial as well as the format of the commands.
4. Student enters command to exit.
5. DM bids student farewell.

Use case ends.

Use Case: Adds a task

MSS

1. Student enters the task to be done.
2. DM stores the task, mark it as undone and updates list of student task.

   Use case ends.

Extensions

1a. DM detects an error in the commands used.

   1a1. DM alerts student of mistake.

   1a2. DM requests for correct data.

   1a3. Student enters new data.

Use Case: List Available Tasks

1. Student Requests for list access.
2. DM loads and shows list of completed and uncompleted tasks.

   Use case ends.

Use Case: Delete a Task

1. Student requests to list tasks.
2. DM shows a list of tasks.
3. Student requests to delete a specific task in the list.
4. DM finds the task and delete it from the list.

   Use case ends.

Use Case: Find a Task

1. Student requests to find tasks with a specific keyword.
2. DM finds the tasks with the keyword and shows student.

   Use case ends.

Use Case: Create a Task with Minimal Required Duration

1. Student requests to input task with fixed duration but no start or end time.
2. DM stores the task, mark it as undone and updates list of student task.

   Use case ends.

Use Case: Create a Repetitive Task

1. Student requests to input repetitive task.
2. DM stores the task and updates the deadline accordingly.
3. DM mark them all as undone and updates list of student task.

   Use case ends.

Use Case: Adding a Contact Information

MSS

1. Student requests to input contact information.
2. DM stores the input accordingly even if certain information is not included.

   Use case ends.

Extensions

1a. DM detects an error in the commands used.

1a1. DM alerts student of mistake.

1a2. DM requests for correct data.

1a3. Student enters new data.

Use Case: Resetting or creating a new Budget

MSS

1. Student requests to input a new budget.
2. DM clears existing budget, stores the budget and show the student the current budget

   Use case ends

Use Case: Adding a Budget Expense

MSS

1. Student requests to add a new budget with an optional description
2. DM adds the budget and description to the list and shows the student that the list has been updated.

   Use case ends

Use Case: Undoing a budget expense

MSS

1. Student requests to undo a new budget with an optional description
2. DM finds the last entered budget, updates the current budget with the previous budget, removes the last entered budget from the list and informs the student that budget has been undone.

   Use case ends

Appendix D : Non-Functional Requirements

1. Should be able to respond to requests in less than 2 seconds.
2. A user should be able to accomplish the majority of the tasks faster than writing on a physical handbook using commands.
3. Should be able to be used on every windows computer that supports Java 11 or above.
4. Should be less than 500MB.

Appendix E : Instructions for Manual Testing

Given below are instructions to test the app manually. The testing list serves as a guideline and testers should do more exploratory testing.

E.1. Launch and Shutdown

1. First Launch
   a. Download the jar file and copy it into an empty folder.
   b. Double click on the jar file. [Expected: Launches with sample data]
2. Ways to exit

a. Enter either “bye” or “exit” on the CLI [Expected: Exit with saving]
b. Select Exit Tab in the Menu bar, then click on Close [Expected: Exit with saving]
c. Click on ‘X’ button at the top right corner. [Expected: Exit with saving]
d. Select Exit Tab in the Menu bar, then click on Close without Saving [Expected: Exit without saving]
e. Alt-F4 to exit [Expected: Exit with saving]
f. Using Task Manager to end the program task. [Expected: Exit without saving]

E.2. Creating task entries such as To-Do, Deadline, and FixedDuration.

1. Todo Task
    a. Test case: todo read a book [Expected: Create a todo task successfully if there are no duplicates]
    b. Test case: todo [Expected: Returns an exception]
    c. Test case: todo <a string of whitespaces> [Expected: Returns an exception]
2. Deadline Task
    a. Test case: deadline exams /by 5/12/2019 1600 [Expected: Create a deadline task successfully if there are no duplicates]
    b. Test case: deadline exams /by 12/2019 1600 [Expected: Returns an exception]
    c. Test case: deadline exams /by <any other strings that are not in correct format> [Expected: Returns an exception]
    d. Test case: deadline exams /by [Expected: Returns an exception]
    e. Test case: deadline exams [Expected: Returns an exception]
    f. Test case: deadline [Expected: Returns an exception]
3. FixedDuration Task
    a. Test case:
    b. Test case:
    c. Test case:
    d. ...

E.3. Receiving Reminders

1. Deadline Task
    a. Test case: deadline <task description> /by 03/11/2019 18:00 [Expected: Returns reminders for tasks due in upcoming 3 days before the deadline]
    b. Test case: deadline <task description> /by 03/10/2019 18:00 [Expected: Returns no existing reminders if the task deadline is over]

E.4. Creating Priorities

E.5. Creating budget entries

1. Creating a new budget
    a. budget new 1000.00 [Expected: budget will now be $1000]
    b. budget new 1000.105 [Expected: budget will store only $1000.1]
    c. budget new ten [Expected: budget will raise an error]
    d. budget new -100 [Expected: budget will be now $-100 (debt)]
2. Deducting expenditures from budget
    a. budget minus 10 [Expected: budget deducted, added to budget list]
    b. budget minus ten [Expected: Error raised]
    c. budget minus 10.15678 [Expected: 10.15 deducted from budget, added to budget list]
3. Adding earnings/payments to budget
    a. budget add 10 [Expected: budget will be deducted and added to budget list]
    b. budget add ten [Expected: Error raised]
    c. budget minus 10.15678 [Expected: 10.15 will be deducted from the current budget and added to budget list]
    d. budget minus 1 kopi peng [Expected: 1 will be deducted from budget, added to budget list]
4. Undoing latest entry in budget
    a. budget undo [Expected: Last entry in budget will be undone, and current budget will be modified accordingly.]

E.6. Contacts Management

1. Adding a new contact
    a. Test case: addcontact Prof Tan, 91234567, tancc@nus.edu.sg, E1-08-11 [Expected: Show total contacts in the list. Show name, number, email and office added.]
    b. Test case: ac Alexa, 91238765, alexa@gmail.com, <an empty space> [Expected: Show total contacts in list and marks the empty field as “Nil”]
    c. Test case: ac Thanos, +21 2671231231, notValid, Nil [Expected: Returns an exception]
    d. Test case: addcontact [Expected: Returns an exception]
    e. Test case: addcontact <a string of whitespaces> [Expected: Returns an exception]

2. Listing all contacts
    a. listcontact [Expected: Shows the full list of contacts saved]
    b. listcontacts [Expected: Shows the full list of contacts saved]
    c. lc [Expected: Shows the full list of contacts saved]
    d. lc <a string of whitespaces> [Expected: Returns an exception]
3. Finding a saved contact
    a. findcontact Tan [Expected: Shows the full detail of the first contact as it found "Tan" in the details of the contact]
    b. fc gmail [Expected: Shows the second contact as the keyword matches]
    c. fc Alcatroz [Expected: Shows no matching tasks found]
4. Delete a contact
    a. deletecontact 2 [Expected: Shows the number of contacts left and the full details of the contact that was recently removed]
    b. dc 2 [Expected: Returns an exception as now there is only 1 contact left]
    c. dc 0 [Expected: Returns an exception]
    d. dc 1 [Expected: Shows the number of contacts left and the full details of the contact that was recently removed]

E.7. Detecting Duplicates (Deadline, Todo, FixedDuration)

1. Detect duplicates in deadline
    a. deadline fintech registration /by 10/11/2019 2359 [Expected: Create a deadline task successfully]
    b. deadline fintech registration /by 10/11/2019 2359 [Expected: Returns an error message and prevents the addition of the deadline task with same description]
2. Detect duplicates in todo
    a. todo find my cat [Expected: Create a todo task successfully]
    b. todo find my cat [Expected: Returns an error message and prevents the addition of the todo task]
3. Detect duplicates in fixedDuration
    a. fixedduration build a robot /for 3 hour [Expected: Create a fixedDuration task successfully]
    b. fixedduration build a robot /for 3 hour [Expected: Returns an error message and prevents the addition of the fixxedduration task as the description is the same]

E.8. Detecting Anomalies (Deadline tasks only)

1. Detecting date and time clashes
    a. The following requires two commands:
    - Test case, command 1: deadline cs2113 mcq /by 5/12/2019 2300

- Command 2: deadline ma1508e quiz /by 5/12/2019 2300 [Expected: Returns an exception]

b. The following requires three commands:

- Test case, command 1: deadline cs2040 mcq /by 4/12/2019 2100 [E.g. Task is saved as the first task]
- Command 2: done 1 [Mark Deadline task as done]
- Command 3: deadline cs1010 quiz /by 4/12/2019 2100 [Expected: Create a deadline task successfully if there are no duplicates]

E.9 Miscellaneous

1. Show a list of tasks
   a. Test case: list [Expected: Display a list of tasks successfully]
   b. Test case: list <any other string or whitespaces> [Expected: Returns an exception]
2. Find tasks
   a. Test case: find read [Expected: Returns matching tasks that contains "read", case-sensitive]
   b. Test case: find [Expected: Returns an exception]
3. Mark task as done
   a. Test case: done 1 [Expected: First task is marked done successfully]
   b. Test case: done 0 [Expected: Returns an exception]
   c. Test case: done 999 [Expected: Returns an exception if size of task list is less than 999]
   d. Test case: done <any other strings that are not valid> [Expected: Returns an exception]

E.10. Update

1. Update task description (1b, c, d, f, g apply to 2 and 3 too)
   a. Test case: update 1 /desc last assignment [Expected: First task's description is updated successfully if there are no duplicates]
   b. Test case: update 0 /desc last assignment [Expected: Returns an exception]
   c. Test case: update 999 /desc last assignment [Expected: Returns an exception if size of task list is less than 999]
   d. Test case: update <any other values or strings that are not within the range of the task list's size> /desc last assignment [Expected: Returns an exception]
   e. Test case: update 1 /desc [Expected: Returns an exception]
   f. Test case: update 1 [Expected: Returns an exception]

    g. Test case: update [Expected: Returns an exception]

2. Update date description
    a. Test case: update 1 /date 27/11/2019 2359 [Expected: First task’s date and time is updated successfully if a. There are no tasks that have the same date and time, b. The task is a Deadline task]
    b. Test case: update 1 /date 27/11/2019 [Expected: Returns an exception]
    c. Test case: update 1 /date <any other strings that are not in correct format> [Expected: Returns an exception]
    d. Test case: update 1 /date [Expected: Returns an exception]
3. Update type of task
    a. Test case: update 1 /type deadline [Expected: First task’s type is updated successfully]
    b. Test case: update 1 /type todo [Expected: First task’s type is updated successfully]
    c. Test case: update 1 /type fixedduration [Expected: First task’s type is updated successfully]
    d. Test case: update 1 /type <any other strings that are not valid> [Expected: Returns an exception]
    e. Test case: update 1 /type [Expected: Returns an exception]

E.11. Notes

1. Add/Update notes (1b, c, d, f, g apply to 2 and 3 too)
    a. Test case: notes 1 /add read up bubble sort [Expected: First task’s notes is updated successfully]
    b. Test case: notes 0 /add read up bubble sort[Expected: Returns an exception]
    c. Test case: notes 999 /add read up bubble sort [Expected: Returns an exception if size of task list is less than 999]
    d. Test case: notes <any other values or strings that are not within the range of the task list’s size> /add read up bubble sort [Expected: Returns an exception]
    e. Test case: notes 1 /add [Expected: Returns an exception]
    f. Test case: notes 1 [Expected: Returns an exception]
    g. Test case: notes [Expected: Returns an exception]
2. Delete notes

    a. Test case: notes 1 /delete [Expected: First task’s notes is deleted successfully, it is set to “empty”]

3. Show notes
    a. Test case: notes 1 /show [Expected: First task’s notes is displayed, if notes is empty, it will display “empty”]

E.12. Filter

1. Filter tasks
    a. Test case: filter todo [Expected: Task list is filtered successfully]
    b. Test case: filter deadline [Expected: Task list is filtered successfully]
    c. Test case: filter fixedduration [Expected: Task list is filtered successfully]
    d. Test case: filter <any other strings or whitespaces> [Expected: Returns an exception]
    e. Test case: filter [Expected: Returns an exception]

E.13. Backing up Duke Manager

1. Saves the backup and launches the file explorer directory where the data files are contained.
    a. backup [Expected: file directory containing backup folder opens with the saved data files]