

My role was to design and write the codes for the expenses, specialization, tag and category list features. The following sections illustrate these enhancements in more detail, as well as the relevant documentation I have added to the user and developer guides in relation to these enhancements.

Note the following symbols and formatting used in this document:



This symbol indicates important information.

`add`

A grey highlight (called a mark-up) indicates that this is a command that can be inputted into the command line and executed by the application.

`ExpenseCommandParser`

Blue text with grey highlight indicates a component, class or object in the architecture of the application.

Summary of Contributions

This section shows a summary of my coding, documentation, and other helpful contributions to the team project.

Enhancement added: I added the expenses feature

What it does: User can add his/her expenses using the `add` command. This adds a new expense into the existing list. The `find` command allows users to find all the expenses recorded on a specific date, and the user can also delete unwanted expenses using the `delete` command. To see all existing expenses, the user can use the `list` command.

If the user has made a mistake, the user can use the `undo` command to undo the previous command. In case the user forgets the commands, he can call `commands` to see the commands in expenses or `help` to see all the commands in the programme.

Justification: This expense feature enables users to record their finances. This is fitting for our target audience who are university students, as most university students do not have earnings and it is important that they can manage their tight finances well. Hence the `add` command allows them to record what they have spent, and `list` command allows them to see all their expenses. This can let them have an overview of their spending. In the event that users have made an error in recording an expense, the `delete` command allows them to remove that particular expense. If

the users want to track how much they spent on a particular date, they can use the `find` command to see the list of expenses on that day.

Code Contributed: Upon project phase 3, I contributed 3668 lines of code to the project. Please click the link to see a sample of my code contribution:

<https://nuscs2113-ay1920s1.github.io/dashboard/#search=e0323290&sort=groupTitle&sortWithin=title&since=2019-09-21&timeframe=commit&mergegroup=false&groupSelect=groupByRepos&breakdown=false&until=2019-10-30&tabOpen=true&tabType=zoom>

Other contributions:

Other enhancements and features:

Designed and wrote the code for Specialization feature (Pull requests [#150](#), [#162](#), [#301](#), [#303](#), [#306](#), [#317](#))

Designed and wrote the code for Tagging feature (Pull request [#84](#))

Designed and wrote the code for Category List feature (Pull request [#78](#))

Designed and wrote the code for DoWithinPeriod Task feature (Pull requests [#52](#), [#68](#))

Designed and wrote the code for enhancement of Delete function(delete more than 1 or all tasks at one go) (Pull request [#73](#))

Designed and wrote the code for toggling Do and Undo function (Pull request [#74](#))

Designed and wrote the code for Done and Undone Lists (Pull request [#83](#))

Documentations:

Made formatting improvements and vetted through the existing User Guide to make it more reader-friendly and consistent throughout.

Write-up for my own features and added images for easy references.

Community:

Reviewed Pull Requests ([#48](#), [#49](#), [#51](#), [#59](#), [#62](#), [#64](#), [#67](#), [#69](#), [#70](#), [#72](#), [#76](#), [#77](#), [#116](#), [#119](#), [#130](#), [#139](#), [#158](#), [#161](#), [#184](#), [#280](#), [#281](#), [#286](#), [#288](#), [#305](#))

Contributions to the User Guide

We had to update the original Duke User Guide with instructions for the enhancements that we had added. The following is an excerpt from our **Gazeebo User Guide**, showing additions that I have made for the expenses feature as an example.

Expenses Page:

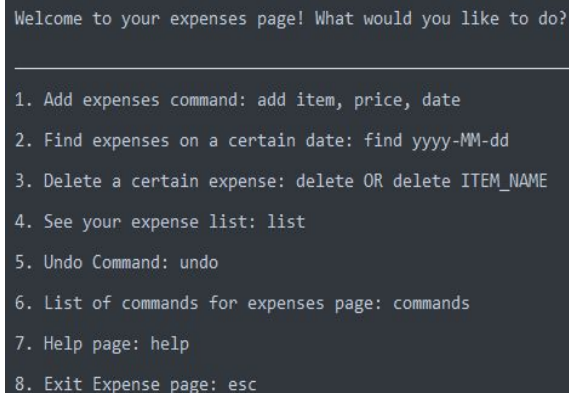
The expenses feature mainly helps you record your expenses to help you manage your finances.

How to arrive at the Expenses page:

From main page, type `expenses` and press ENTER. The system will bring you to the start up Expenses page.

▲ You can only enter the Expenses page from the main page.

Following Figures 2 show the start-up page of Expenses.



```
Welcome to your expenses page! What would you like to do?  
  
1. Add expenses command: add item, price, date  
2. Find expenses on a certain date: find yyyy-MM-dd  
3. Delete a certain expense: delete OR delete ITEM_NAME  
4. See your expense list: list  
5. Undo Command: undo  
6. List of commands for expenses page: commands  
7. Help page: help  
8. Exit Expense page: esc
```

Figure 2. Expense page

Below are 2 function demonstrations of Expenses, add and delete.

Scenario 1: Add expenses to the existing list : `add`

This command adds an expense by recording the item bought, price and the date of purchase.

Example: Let's say you bought bread at \$4 on 2019-09-09, you can record it as seen in the figure below.

```
add bread, $4, 2019-09-09
Successfully added:
bread, $4, bought on 2019-09-09
|
```

Figure 3. Adding an expense

- ▲ You must key in the format: `add ITEM_NAME, PRICE, DATE`
- ▲ Date entered must be in the YYYY-MM-DD format.

Scenario 2: Delete a certain expense: `delete`

This command allows users to delete an existing expense from the list.

Example: Let's say you have added the bread expense by mistake, you can delete it by executing the `delete` command as seen in the figure below.

```
delete bread
Successfully deleted: bread, $4 | bought on 2019-09-09
```

Figure 4. Deleting an existing expense

- ▲ You must key in the format: `delete ITEM_NAME` OR `delete`, then the system will prompt for an `index number` of the item you wish to delete.
- ▲ If the index of the expense entered does not exist in the list, an `IndexOutOfBoundsException` will be caught and an error message will be displayed.

Contributions to the Developer Guide

The following section shows my addition to the **Gazeebo Developer Guide** for the expenses feature. This is just a small excerpt of the expenses feature.

Expenses Section

The expenses feature is facilitated by `ExpenseCommandParser`. It extends the `Command` class with the expenses feature, stores the `Expenses` internally as an `HashMap<LocalDate,ArrayList<String>>` expenses and also externally as `Expenses.txt`. Additionally, it implements the following operations:

`ExpenseCommandParser#add()` — Adds expenses to existing expenses list.

`ExpenseCommandParser#delete()` — Delete certain expenses from the existing list.

`ExpenseCommandParser#find()` — Find list of expenses recorded on a specific date.

`ExpenseCommandParser#list()` — Shows the list of expenses recorded.

The following sequence diagram shows how the `ExpenseCommandParser` works:

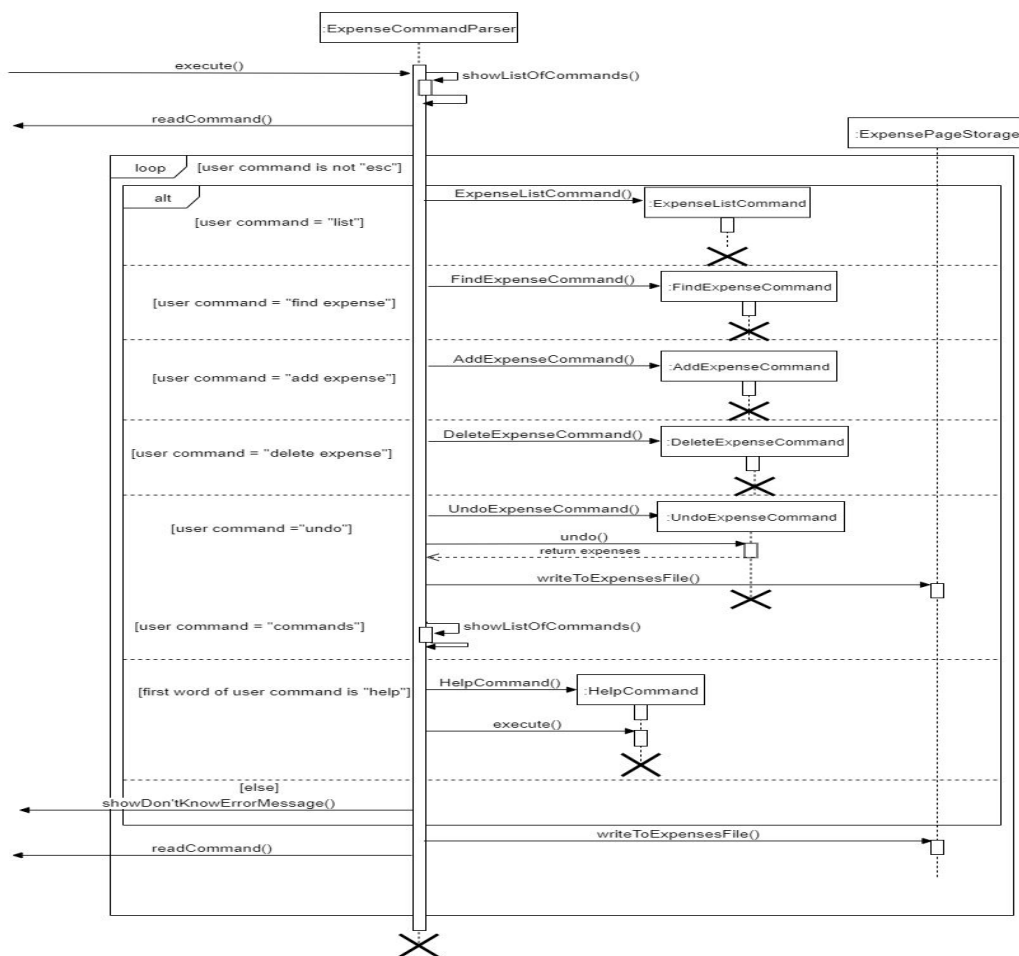


Figure 5. UML Sequence Diagram of `ExpenseCommandParser`

Given below is an example usage scenario and how the the add function of Expenses.

Step 1. The user enters the application. The `ExpenseCommandParser` is initialized when expenses is called in the main page.

Step 2. The user executes `add` to add a new expense by calling `AddExpenseCommand`. User must input in the sequence of the item, price and date of purchase (e.g. add bread, \$4, 2019-09-09). The added expense is added in to the internal `Map<LocalDate, ArrayList<String>> expenses` and it is saved in the external storage `Expenses.txt`.

The following activity diagram summarizes what happens when a user executes a new command:

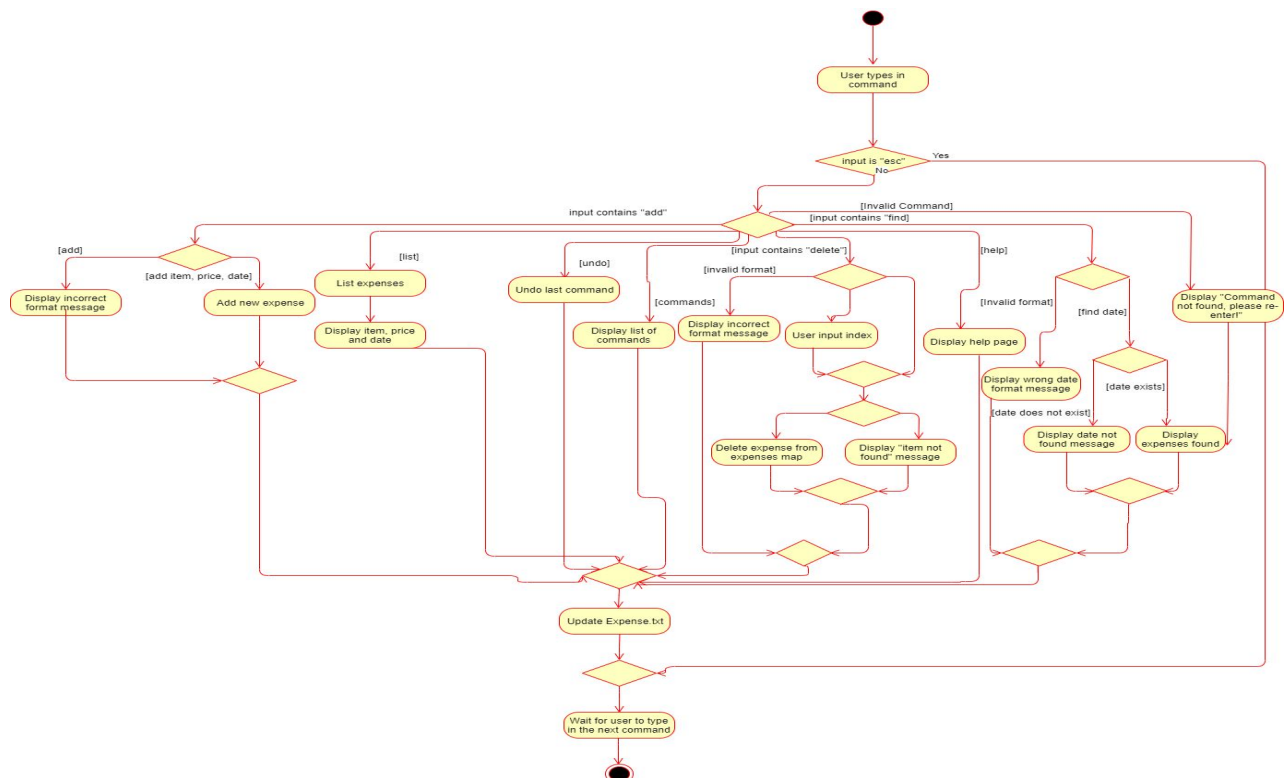


Figure 6. UML Activity Diagram of `ExpenseCommandParser`

Design Considerations

When designing the expenses feature functions, I had to make decisions on how best to execute the commands and what data structure to support the commands. The following is a brief summary of my analysis and decisions.

Aspect	Alternative 1	Alternative 2
Data structure of Expense feature	<p>Using TreeMap to store expenses with key as the LocalDate and values as the list of expenses recorded on that date (item name and price).</p> <p>Pros: TreeMap has a natural ordering of keys, hence there is no need to manually sort the map in order.</p> <p>Cons: TreeMap is slower than HashMap as it has a time complexity of $O(\log(n))$ for deletion and insertion.</p> <p>I decided to proceed with this option because for our feature, it is intuitive for users to have a date-sorted list of expenses to keep the expense list organized.</p>	<p>Using ArrayList to store LocalDate and string of expenses</p> <p>Pros: HashMap operates faster than Map as its time complexity for deletion and insertion is $O(1)$.</p> <p>Cons: HashMap allows the storing of null keys and values, allowing less optimal amount of allocation of memory to store items compared to TreeMap.</p>

[Proposed Feature] Uncheck function for technical electives in Specialization feature (coming in v2.0)

Implementing an Uncheck function:

An Uncheck function is to mark a technical elective as incomplete if it was once marked as completed. It can be integrated by using the boolean function. If a technical elective is completed, it will be marked as "true". Hence, to uncheck a completed technical elective, if a technical elective is "true", change the boolean to "false".