Weng Kexin - Project Portfolio

PROJECT: Dolla, an expense manager for students staying on NUS

This project portfolio serves to document my involvement and contribution in the team-based project module CS2113T (Software Engineering & Object-Oriented Programming) at the National University of Singapore in Semester 1 of the 2019/2020 academic year.

Overview

My team and I were tasked with enhancing a basic command line based application, Duke for our project. We chose to morph it into a personal financial assistant designed for NUS students staying on campus, called Dolla. This enhanced application enables NUS residents to track and manage their expenses and incomes, budgeting and saving goals, as well as keeping track of debts.
My role was to design and implement the features - Budget and Saving tracking and the help feature. The following sections illustrate these implementations in greater detail, as well as the relevant documentations I have added to the user and developer guides in relation to these enhancements.

Summary of Contributions

This section shows a summary of my coding, documentation and other contributions to the project.
Code contributed: Please click this link to see an example of my code: [Project Code Dashboard]
Features implemented:
Biggest feature: added the ability for user to track budgets and savings (also called limits).

- What it does: the `set [LIMIT TYPE] [AMOUNT] [DURATION]` command allows the user to set a new budget or saving for a fixed duration (daily, weekly or monthly). The user may also remove or modify the limit. After setting a budget or saving limit, the user is able to view their remaining limit for a specified duration, by using the command `remaining [DURATION] [LIMIT TYPE]` (eg. remaining daily saving). There will be a message notifying the user if they have exceeded their budget or achieved their savings goal.
- Justification: users are now able to set specific financial goals they wish to achieve, within a specified duration and be more aware of how much they can spend or have to save in a given time span. Their financial overview are presented to them visually. This will thus reinforce better spending habits in the user.
- Highlights: this feature works with existing commands. Every time the user inputs a new expense or income, the remaining budget and saving will change accordingly. There is also a visual representation of the remaining budget or savings they have, for users to have a better

overview of their finances and hence, inducing them to make better spending and saving decisions.

Other feature implemented:

- Added the help feature where users are able to see the list of dolla commands available by typing “help” into the application. This enables users to see a quick summary of the available commands, without having to refer to the lengthy user guide.

Other contributions

- Project management
    - Managed release versions on Github and submission of group documents.
    - Managed deadlines, milestones and issues on Github.
- Documentation
    - Wrote the Developer Guide for v1.1 submission.
    - Made the Developer and User Guide more reader friendly with cosmetic tweaks.
- Community
    - Corrected checkstyle errors and reported commonly made mistakes (#148, #150)
    - Managed DollaException and fixed high severity bugs (#254, #275)
    - Refactored codes (#197, #264, #277)
- Tools
    - Integrated Codacy and Coveralls, to automate code quality analysis and coverage checks on the team repository and enforce code quality standard (#248)

Contributions to the User Guide

The following is an excerpt from our ***Dolla User Guide***, showing the features I have implemented for the project. It illustrates my ability in writing documentation targeting end-users to guide them in using the various features of our application.

4.6. Switch the mode to Limits: limit

You can add, edit, view and remove your budgets and savings in this mode.

Format:

`limit`

NOTE: [DURATION] can be daily, weekly or monthly.

Common Commands Available:

`remove` `search duration` `modify` `sort amount` `help` `bye`

4.6.1. View the list of limits you have added: limits

View the list of budget and saving goals that you have set for yourself.

Format:

`limits`

4.6.2. Check your remaining budgets: remaining

Check on how much budget you have left for the duration you chose!

Format:

`remaining [DURATION] budget`

Example:

- `remaining daily budget`

4.6.3. Check your saving goals: remaining

Check on how much money you have saved for the duration you chose!

Format:

`remaining [DURATION] saving`

Example:

- `remaining monthly saving`

4.6.4. Set duration-based expense budget: set budget

Set the budget on the amount of money you wish to spend within the time period you have input.

Format:

`set budget [AMOUNT] [DURATION]`

Example:

- `set budget 50 weekly`

4.6.5. Set duration-based saving goals: set saving

Set a target saving you wish to achieve for a specified duration.

Format:

`set saving [AMOUNT] [DURATION]`

Example:

- `set saving 300 monthly`

4.6.6. Remove a limit from the list: remove

Remove a limit (budget/saving) from the list.

Format:

`remove [LIST NO.]`

Example:

- remove 1

Contributions to the Developer Guide:

The following is an excerpt from our ***Dolla Developer Guide***. These sections showcase my ability in writing technical documentation to provide developers with insights on the design of our application and also showcase the technical debt of my contributions to the project.

2.3. Ui component

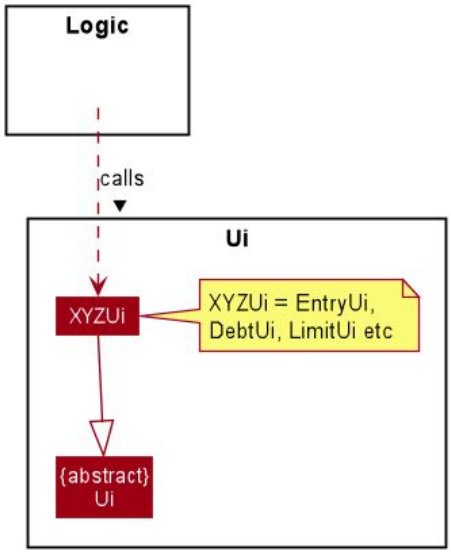

Figure 5: Structure of the `Ui` component

The `Ui` package handles the printing of output in a user-friendly manner. It consists of an abstract class `Ui` and other corresponding ui classes, that inherits from the abstract class (such as EntryUi, DebtUi and LimitUi), as seen in figure 5 above.

1. Depending on the commands input by the user, the `Logic` executes a command and calls the respective `Ui` class to notify user of a successful or failed command execution. These `Ui` classes all inherit from the abstract class `Ui.java`, as seen in figure 5 above.
2. The `Ui` component listens for instructions from `Logic`, so that the `Ui` can be updated with the new data to echo.
3. For example, when the user executes a command `set budget 50 weekly`, the corresponding command `AddLimitCommand` is executed. After successful execution of the command, `Logic` calls the method `echoAddRecord(limit)` in the `LimitUi` class, to echo details of the newly added limit and notify the user.

3.1. Budget and saving tracking

3.1.1. Implementation

The budget and saving tracking mechanism is facilitated by the `LimitParser` in Logic, which extends from `MainParser`. Limit here refers to both budget and saving. `MainParser` parses the user input and determines the `commandToRun`, before calling `LimitParser` to handle the user input (when the user is in limit mode). There are various commands available in the limit mode and the command to be called is determined by the `commandToRun`. This was explained in further detail in section 2.2 above, on the logic component.

The main commands that facilitates the limit tracking process are `AddLimitCommand` and `ShowRemainingLimitCommand`. `AddLimitCommand` allows users to set a budget or saving for a given duration. `ShowRemainingLimitCommand` shows users the amount of money they have already spent (expenses), against their set budget, or the amount of money they have saved (incomes), against their target savings goal. There is also a visual representation of the remaining budget or savings they have, for users to have a better overview of their finances.

In `LimitParser`, after `commandToRun` has been determined, the logic first verifies that the input command is valid. If so, it calls the corresponding command (eg.`ShowRemainingLimitCommand` or `AddLimitCommand`), with different components depending on the input command (eg. [TYPE] [AMOUNT] [DURATION]). `Model` is then used to access the limit data, which is stored internally in `LimitList`. The data is then modified in the `Storage` component and the corresponding `LimitUi` is called, to notify the user of a successful execution of the command.

Given below is an example usage scenario of how the set limit mechanism behaves at each step.

Step 1. The user executes the `limit` command to switch mode to limit.

Step 2. The user executes `set budget 10 daily` to add a new budget (or `set saving...` to add a new saving) to the system. `AddLimitCommand` is called.

i	If the input fails its execution, it will not call `AddLimitCommand` and the limit will not be saved into the LimitList.

Step 3. The user is now able to view his current limits by executing the `limits` command. The `limits` command will call `ShowListCommand(limit)`, which will list out all the existing limits of the user.

ℹ	If a limit type of a specified duration already exists, the user is not able to add the limit type of the same duration to Dolla. The system will return an error message to the user and suggest that they modify the limit instead.

The following sequence diagram shows how the set limit operation works:

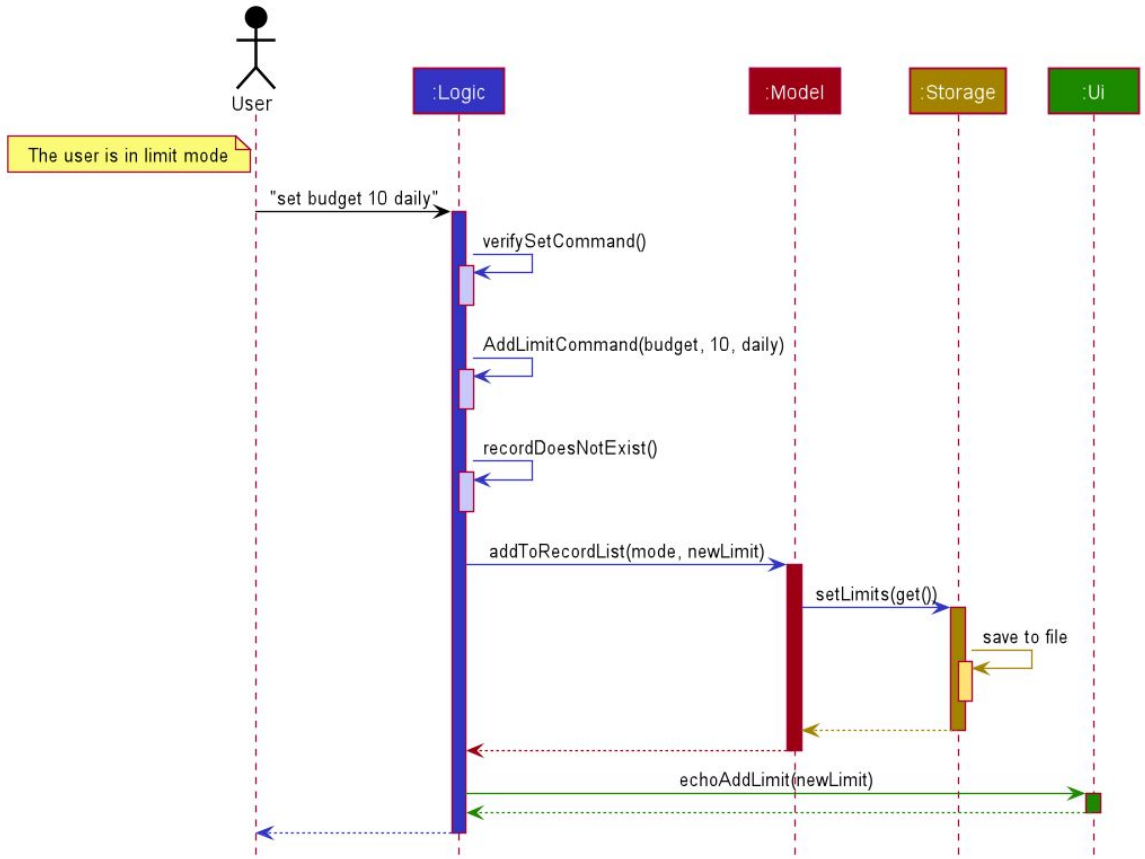

Figure 8. AddLimitCommand sequence diagram

Given below is an example usage scenario of how the show remaining limit mechanism behaves.

Step 1. The user is currently in `limit` mode and types `remaining daily saving` to view his current saving goal for the day.

Step 2. The system verifies the `commandToRun` and calls `ShowRemainingLimitCommand()` with parameters `daily` and `saving`.

Step 3. With the given parameters, `findLimit()` and `findTotalEntries()` in `execute()` are called, to find the amount of `saving` the user has set for the duration `daily`, as well as the total `income` the user has input for that duration.

Step 4. The system then calls `processRemainingLimit()` to output the remaining `saving` the user has for duration `daily`.

ℹ	If no limit is found, a `DollaException.noExistingLimit` is thrown in the try block of `execute()` and the user will be informed by `LimitUi` that they have not set a limit for the specified duration.

Step 5. The system calls `LimitUi` to print out a message regarding the user's daily saving, for him to review. The user is also able to remove or edit their limit if they wish to.

The following sequence diagram shows how the show remaining limit operation works:

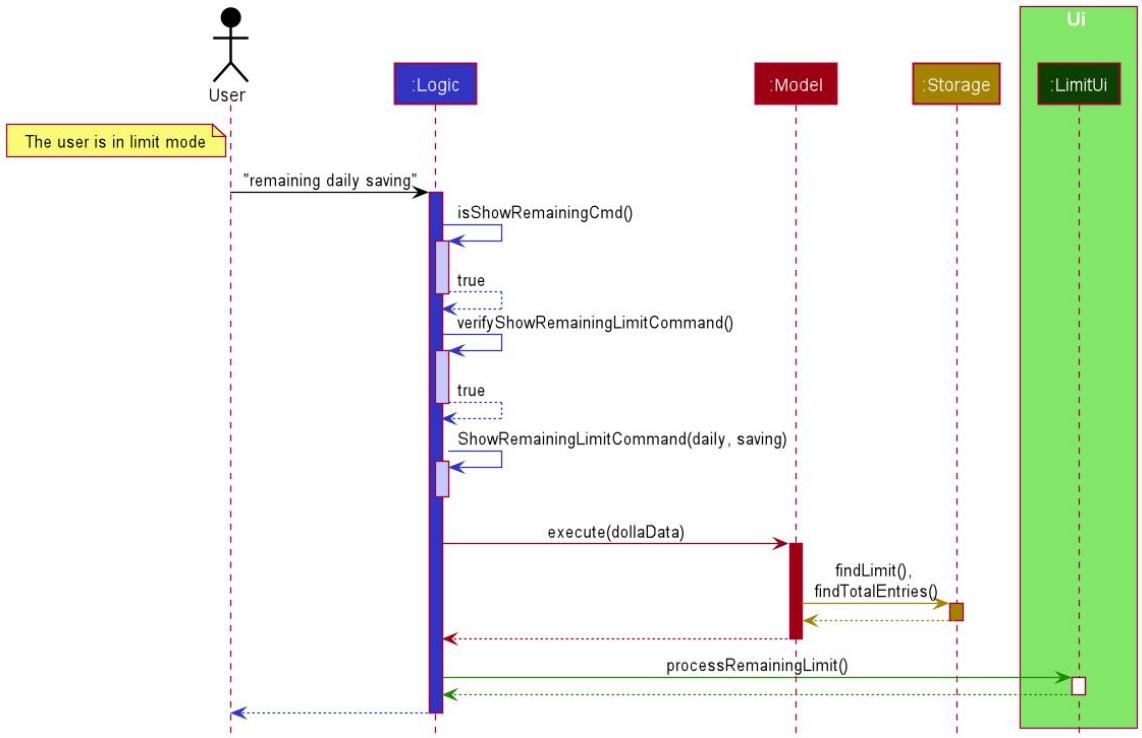

Figure 9. ShowRemainingLimit sequence diagram

3.1.2. Design Considerations

Aspect: How the tracking executes

- Alternative 1 (current choice): Saves the limits in a LimitList and calls the LimitList to get the corresponding limit amount whenever the user wishes to track their budget / saving goal.
    - Pros: Easy to implement.
    - Cons: May have performance issues in terms of memory usage.
- Alternative 2: Storing a reference of limits to be dynamically changed whenever the user inputs new saving or budget data.
    - Pros: More efficient.
    - Cons: Increases coupling and hence dependency of classes, which is not recommended.

Aspect: Data structure to support limit tracking

- Alternative 1 (current choice): Use an ArrayList to store the limits set by the user.
    - Pros: Easy to implement.
    - Cons: May have performance issues if the number of limits are high (issue not valid as of now as there can only be one limit type of each duration.
- Alternative 2: Use a hashmap to store limits instead.
    - Pros: Increased efficiency and performance due to O(1) access.
    - Cons: More complex and hence difficult for new computer science students (who are likely to be the new incoming developers of our project) to understand.

3.5. Help feature

3.5.1. Implementation

The help feature is facilitated by `HelpCommand()`, which extends Command. This feature is available regardless of the mode the user is on. When the user types `help`, `MainParser` in `Logic` will parse the input and determine the commandToRun, and then call `HelpCommand()`. `HelpCommand()` then calls `helpCommandPrinter()` in the `HelpUi` class, to print out a summary of the available commands. Given below is an example usage scenario.

Step 1. The user is using Dolla and has difficulty executing a particular command as he has forgotten the command format. He enters the input `help` into the system.

Step 2. After parsing the input, the system executes `execute()` in `HelpCommand()`, which calls `helpCommandPrinter()` in the `HelpUi` class, to print out the list of available commands for the user to process and read.

3.5.2. Design Considerations

Aspect: How the printing of commands execute

- Alternative 1 (current choice): There is a predefined list of commands with its description in the HelpUi class. The helpCommandPrinter prints out the predefined list after being called.
    - Pros: Easy to implement.
    - Cons: reduced readability since HelpUi class is cluttered with command strings.
- Alternative 2: Have a manual.txt and read and print from it when the user executes the help command.
    - Pros: Reduces clutteredness of HelpUi class and higher portability of help commands.
    - Cons: Increased complexity of the implementation. New computer science students, who are likely to be the new incoming developers of our project, might have a hard time understanding the implementation, as it involves reading and printing from a text file.