Daniel Wong – Project Portfolio for treasurerPro (tP)

1. Introduction

This document serves as a project portfolio for treasurerPro, and outlines my contributions to the project, including the features that I have implemented.

1.1. About the team

My team of five consists of four Year 2 Computer Science students, including me, and a Year 4 Computer Engineering student, all taking the module CS2103T Software Engineering.

1.2. About the project

This project was developed as part of the module CS2103T Software Engineering. We were tasked to develop a desktop application (Windows/macOS/Linux) with a Command Line Interface (that is, the program operates via text input from the user, called commands). Additionally, we were required to use an existing application, called AddressBook Level 3, as the starting point for building our application.

My team decided to create an application that would help treasurers of Co-Curriculur Activities (CCAs) and student clubs track their expenses, reimbursements, sales and inventory. To do so, we incorporated the existing people management features of AddressBook and used it as a starting point to build treasurerPro. We also built a new User Interface (UI) to accommodate the new features that we had added.

In total, treasurerPro took a total of 10 weeks to complete.

1.3. Key to the icons and formatting used in the document

ℹ	This symbol indicates extra information or definition.
⚠	This symbol indicates important information to take note of.

`Model` : Text with grey highlight indicates a component, class or object in the architecture of the application.

`command` : Text with blue font and grey highlight indicates a command that can be input by the user.

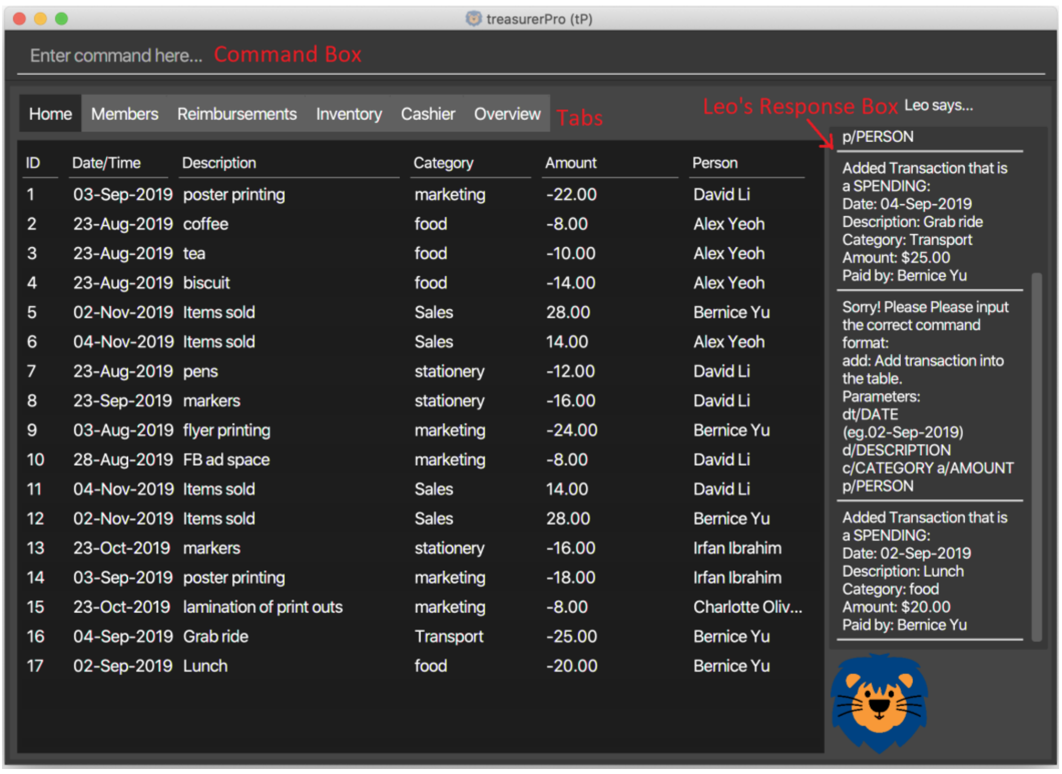
1.4. Introduction to treasurerPro

The User Interface of treasurerPro is primarily divided into 6 tabs - Home, Members, Reimbursements, Inventory, Cashier and Overview. Each tab serves a particular function:

1. The Home tab lists all transactions
2. The Members tab holds information of all the club's members
3. The Reimbursements tab keeps track of outstanding reimbursements
4. The Inventory tab keeps track of items owned by the club
5. The Cashier tab allows for easy sales management
6. The Overview tab shows summary statistics of the club's finances

Additionally, there are two common elements across all tabs. The command box, for users to input their commands, and Leo, the mascot of treasurerPro, who acts as a personal assistant and guides the user.

The following screenshot shows the User Interface of the application on first launch:



ID	Date/Time	Description	Category	Amount	Person
1	03-Sep-2019	poster printing	marketing	-22.00	David Li
2	23-Aug-2019	coffee	food	-8.00	Alex Yeoh
3	23-Aug-2019	tea	food	-10.00	Alex Yeoh
4	23-Aug-2019	biscuit	food	-14.00	Alex Yeoh
5	02-Nov-2019	Items sold	Sales	28.00	Bernice Yu
6	04-Nov-2019	Items sold	Sales	14.00	Alex Yeoh
7	23-Aug-2019	pens	stationery	-12.00	David Li
8	23-Sep-2019	markers	stationery	-16.00	David Li
9	03-Aug-2019	flyer printing	marketing	-24.00	Bernice Yu
10	28-Aug-2019	FB ad space	marketing	-8.00	David Li
11	04-Nov-2019	Items sold	Sales	14.00	David Li
12	02-Nov-2019	Items sold	Sales	28.00	Bernice Yu
13	23-Oct-2019	markers	stationery	-16.00	Irfan Ibrahim
14	03-Sep-2019	poster printing	marketing	-18.00	Irfan Ibrahim
15	23-Oct-2019	lamination of print outs	marketing	-8.00	Charlotte Oliv...
16	04-Sep-2019	Grab ride	Transport	-25.00	Bernice Yu
17	02-Sep-2019	Lunch	food	-20.00	Bernice Yu

```
p/PERSON

Added Transaction that is
a SPENDING:
Date: 04-Sep-2019
Description: Grab ride
Category: Transport
Amount: $25.00
Paid by: Bernice Yu

Sorry! Please Please input
the correct command
format:
add: Add transaction into
the table.
Parameters:
dt/DATE
(eg.02-Sep-2019)
d/DESCRIPTION
c/CATEGORY a/AMOUNT
p/PERSON

Added Transaction that is
a SPENDING:
Date: 02-Sep-2019
Description: Lunch
Category: food
Amount: $20.00
Paid by: Bernice Yu
```

2. Summary of contributions

I served as Team Lead for the project, and my responsibilities included the design of the new User Interface, as well as the features within the Overview tab.

In the following sections, I will illustrate the above-mentioned enhancements in greater detail, along with the corresponding documentation that I have written for them within the user and developer guides.

2.1. Enhancements and new features added

The following describes the enhancements and new features that I added to the project.

1. New User Interface (UI)
    - What it does: The new UI allows for each tab to effectively display its relevant data and interact with the user.
    - Justification: The current AddressBook Level 3 (AB3) UI was unsuitable due to the multitude of new features within each tab that have been implemented. As such, we decided to design a brand new UI that could fully represent the features of treasurerPro.
    - Highlights: The UI was made with SceneBuilder, an application built specifically for designing UIs. It retains AB3's command box, but replaces the bulk of the application with a `TabPane` that houses the content for each tab.
      Additionally, a `GridPane` was used to create space for Leo, our mascot and personal assistant, to interact with the user.
2. Set and Notify Commands in Overview tab
    - What it does: The set and notify command allow the user to set goals for their expenditure, budget and sales revenue, and be notified when a percentage threshold of the goal has been met.
    - Justification: These commands work together in tandem to form the basis of treasurerPro's financial planning feature. They allow the treasurer to stay updated and continually keep track of the club's current financial health.
    - Highlights: Each notification displays only once per app run, unless the goals or percentage thresholds are modified. This prevents the user from being overwhelmed by excessive messages from treasurerPro.
3. Summary statistics view in Overview tab
    - What it does: The summary statistics view in the Overview tab allows the treasurer to monitor and take note of his club's financial health with just a quick glance.

- Justification: It is tedious to flip through tab by tab, and read each table record by record just to gather summary data about the current financial status of the club.
- Highlights: The calculations in this feature were implemented using Java Streams. This allows for future, similar statistics to easily be added, simply by changing the criteria used when processing of the stream.

2.2. Code contributed

Click on the following links to view the code that I have contributed:

- RepoSense
- Functional Code
- Test Code

The links above will bring you to our team's GitHub repository

2.3. Other contributions

The following describes the various other contributions that I have made to the project.

1. Project management:
   - I managed all major releases, from version 1.1 to 1.4. In total, 8 releases were made by me (including interim releases).
   - I took charge of 4 team meetings out of a total of 11 meetings that we held.
2. Enhancement to existing features:
   - Updated JavaFX CSS Stylesheet to include new UI elements.
3. Documentation:
   - Updated diagrams and write-ups in the Developer Guide for the design of the Architecture, UI and Logic components.
   - Proofread User Guide for spelling and grammatical errors.
   - Reformatted User Guide to ensure consistency between sections.
4. Community:
   - Mediated disagreements on workflow process between teammates as Team Lead.
   - Cleaned up code style errors in teammates' packages (PRs: #95 #124)
   - Reviewed Pull Requests (with non-trivial review comments) (PRs: #49 #98 #133 #134 #149 #202 #221)
   - Contributed to forum discussions (links: 1 2 3 4 5 6 7 8 9 10 11 12)
5. Tools:
   - Added Coveralls' code coverage service to the team repository
   - Added Codacy's code quality service to the team repository

3. Contributions to the User Guide

{Start of extract from User Guide}

5.6. Overview Tab

Summary of features in the Overview Tab

- The overview tab displays a variety of statistics for the user.
- These include:
  - Total expenses made thus far
  - Total inventory value
  - Total sales revenue

    - Remaining budget

- The user may also set financial goals and set up percentage thresholds to receive notifications.
- Leo will notify the user any time their financial goals have been reached.

5.6.1. Statistics Information

The following describes how the various statistics are calculated:

- Expense Summary: Total spent represents the sum of all negative transactions (cash outflows) made by the club.
- Inventory Summary: Inventory value represents the total cost of all goods currently in the inventory.
- Sales Summary: Total sales represents the sum of all positive transactions (cash inflow) from the Sales category.
- Budget Overview: Amount remaining represents the budget goal + total sales - expenses.

It is possible for your remaining budget to exceed your budget goal if your sales revenue is larger than your expenses.

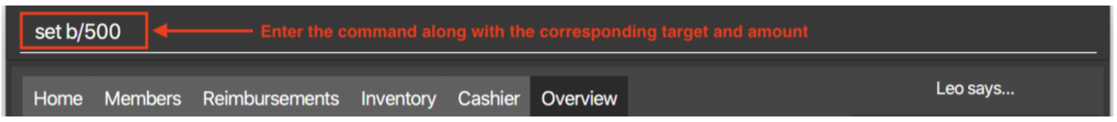
5.6.2. Set a Financial Goal

This command enables you to set a financial goal.

- Command:
    - To set budget goal: `set b/AMOUNT`
    - To set expense goal: `set e/AMOUNT`
    - To set sales goal: `set s/AMOUNT`
- Examples:
    - To set budget goal: `set b/500`
    - To set expense goal: `set e/500`
    - To set sales goal: `set s/500`

The amount should be non-negative, and the maximum amount you may set is 10,000,000 (10 million). If more than 2 decimal places are specified, the amount will be rounded to 2 decimal places. To reset the goal, simply set it to 0.

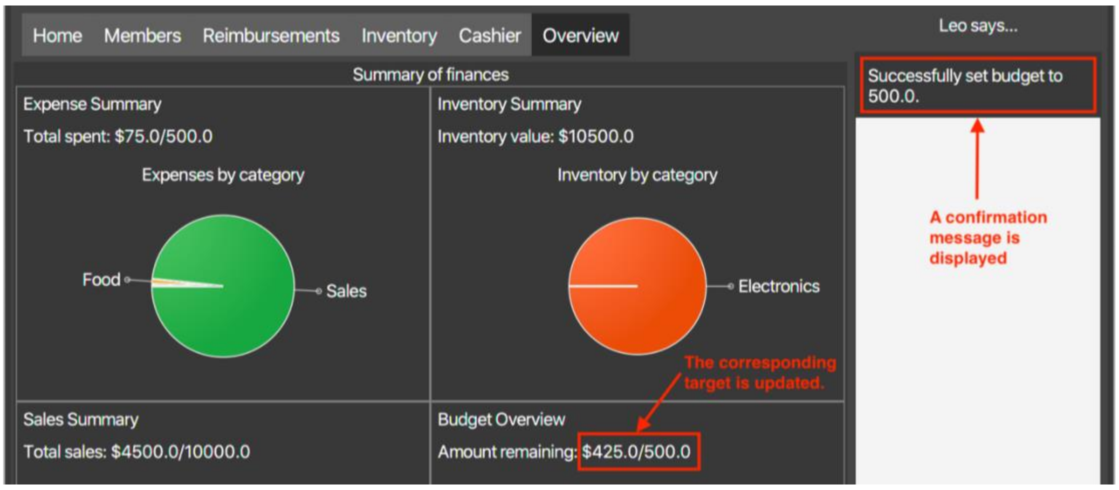
- Steps:
    1. Type the command with the corresponding target and amount.



```
set b/500
```

Enter the command along with the corresponding target and amount
Home
Members
Reimbursements
Inventory
Cashier
Overview
Leo says...

2. Hit `Enter`.



Leo displays a confirmation showing that the goal was successfully set.
The user interface reflects this under the respective section.

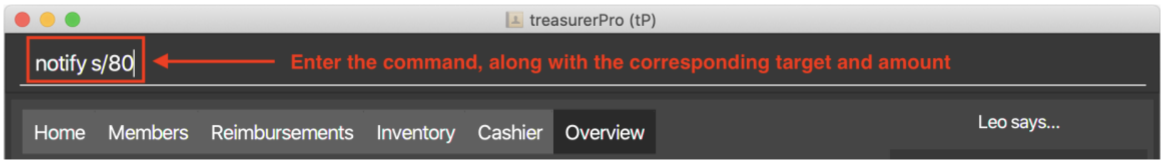
5.6.3. Set a Notification Threshold

This command allows you to set a percentage threshold for notifications.
For example, an 80 percent threshold will mean that you will receive a notification once you have reached 80% of the goal set for that particular financial goal.

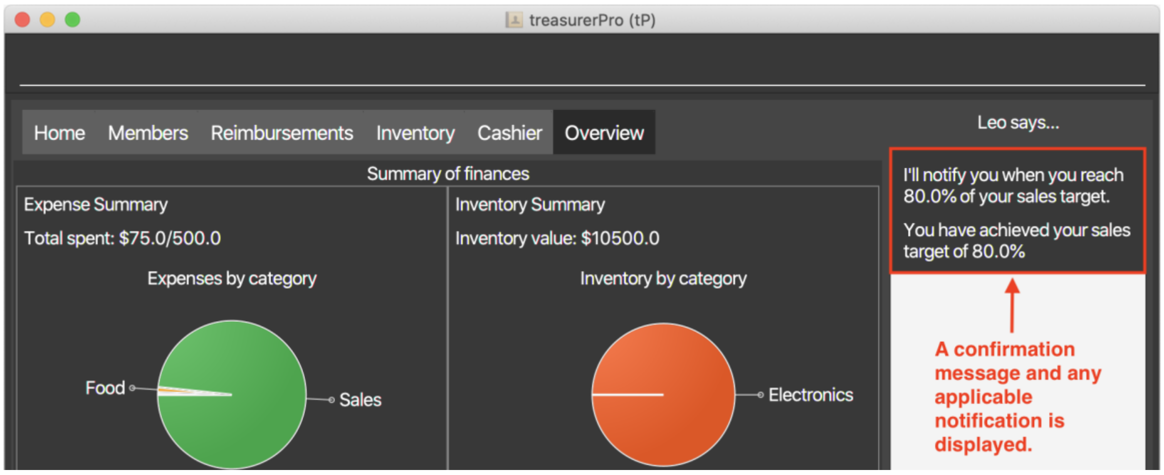
- Command:
  - To set budget goal notification: `notify b/PERCENTAGE`
  - To set expense goal notification: `notify e/PERCENTAGE`
  - To set sales goal notification: `notify s/PERCENTAGE`
- Examples:
  - To set budget goal notification: `notify b/80`
  - To set expense goal notification: `notify e/80`
  - To set sales goal notification: `notify s/80`

The percentage value should be a value from 0 - 100.
This feature will not work if no goals have previously been set with the set command, or if the goal is currently set to 0.
To disable notifications for a particular financial goal, simply set its notifications threshold to 0.

- Steps:
  1. Type the command with the corresponding target and amount.



2. Hit `Enter`.



I'll notify you when you reach 80.0% of your sales target.

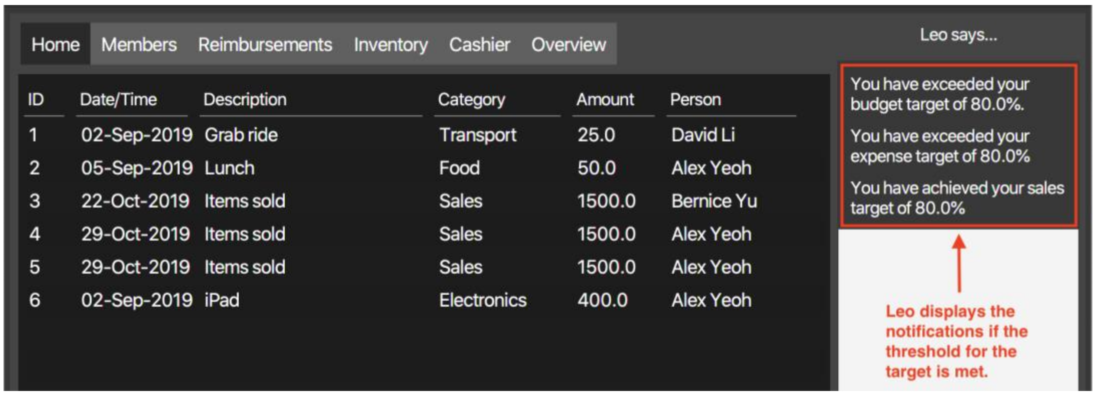
You have achieved your sales target of 80.0%

Leo displays a confirmation showing that the notification was successfully set.
Additionally, if the goal is already reached, it will display the notification immediately.

5.6.4. Viewing Notifications

Notifications in treasurerPro are automatically displayed upon app launch by Leo as long as the threshold is met. Notifications show only once per app run, or whenever a target or threshold is modified.

The following screenshot shows an example of the notifications:


Home
Members
Reimbursements
Inventory
Cashier
Overview
Leo says...
Leo displays the notifications if the threshold for the target is met.

ID	Date/Time	Description	Category	Amount	Person
1	02-Sep-2019	Grab ride	Transport	25.0	David Li
2	05-Sep-2019	Lunch	Food	50.0	Alex Yeoh
3	22-Oct-2019	Items sold	Sales	1500.0	Bernice Yu
4	29-Oct-2019	Items sold	Sales	1500.0	Alex Yeoh
5	29-Oct-2019	Items sold	Sales	1500.0	Alex Yeoh
6	02-Sep-2019	iPad	Electronics	400.0	Alex Yeoh

You have exceeded your budget target of 80.0%.

You have exceeded your expense target of 80.0%

You have achieved your sales target of 80.0%

{End of extract from User Guide}

4. Contributions to the Developer Guide

The following section shows my additions to the treasurerPro Developer Guide.

{Start of first extract from Developer Guide}

2.3.5. Details on `Logic` Implementation for the Overview Tab

This section will show further details of the `Logic` Component of the Overview tab. Given below is a Class Diagram showing the structure of `Parser` within the `Logic` Component:

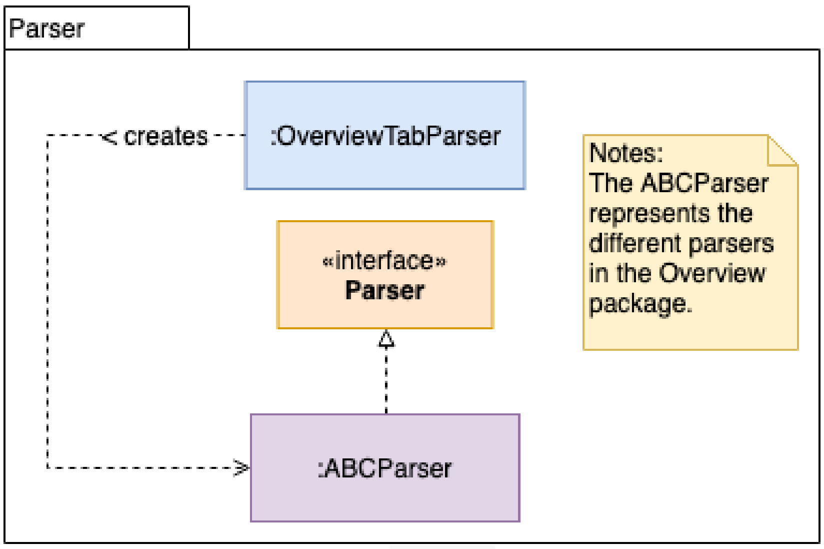

Parser
< creates
:OverviewTabParser
«interface»
Parser
:ABCParser
Notes:
The ABCParser represents the different parsers in the Overview package.

Figure 10. Structure of `Parser` for Overview Tab.

The parsers represented by the `ABCParser` are:

- `SetCommandParser`
- `NotifyCommandParser`

{End of first extract from Developer Guide}

{Start of second extract from Developer Guide}

3.6. Overview Tab

This tab displays various summary statistics for the data within treasurerPro. There are four main statistics shown:

1. Expense Summary: Pie chart of expenditure by category.
2. Inventory Summary: Pie chart of inventory by category.
3. Sales Summary: Bar chart of sales by months.
4. Budget Overview: Line chart of budget remaining by months.

The above summaries are automatically updated whenever new data is entered from any of the other tabs.

There are two main user features within this tab: a feature allowing the user to set goals, and a feature for the user to set percentage thresholds for notifications.

3.6.1. Set Command Feature

This feature allows the user to set a goal for their budget, expense or sales targets.

The following Sequence Diagram depicts how the Set Command operates, and is an extension of the general Sequence Diagram found in [2.3. Logic component: Figure 5](#):

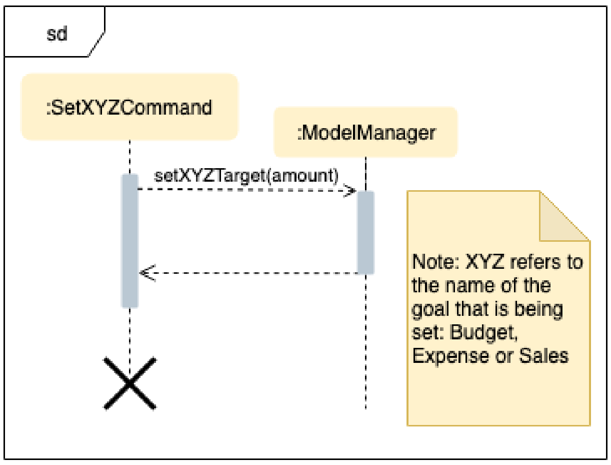

sd
:SetXYZCommand
:ModelManager
setXYZTarget(amount)
Note: XYZ refers to the name of the goal that is being set: Budget, Expense or Sales

Figure 52. Sequence Diagram of Set Command in the Overview package.

After execution of the command, the `LogicManager` also instructs the `StorageManager` to save the new information to the data file.

3.6.2. Notify Command Feature

This feature allows the user to set a percentage threshold for notifications. Upon hitting that percentage for a particular financial goal, the user will automatically be notified of it with a message from Leo.

The following Sequence Diagram depicts how the Notify Command operates, and is an extension of the general Sequence Diagram found in 2.3 Logic component: Figure 5:

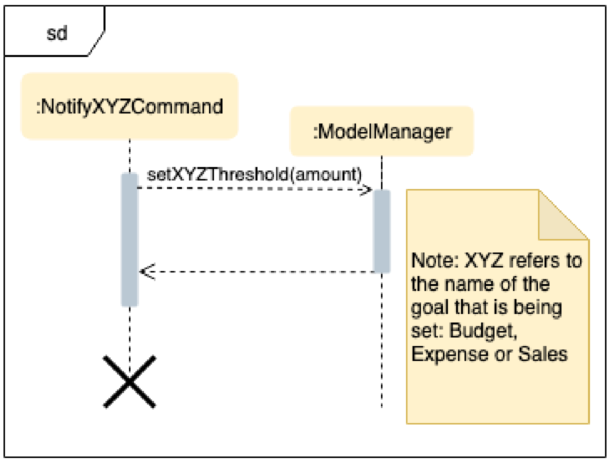

sd
:NotifyXYZCommand
:ModelManager
setXYZThreshold(amount)
Note: XYZ refers to the name of the goal that is being set: Budget, Expense or Sales

Figure 53. Sequence Diagram of Notify Command in the Overview Package

After execution of the command, the `LogicManager` also instructs the `StorageManager` to save the new information to the data file. The full execution of the command is shown in the activity diagram below:

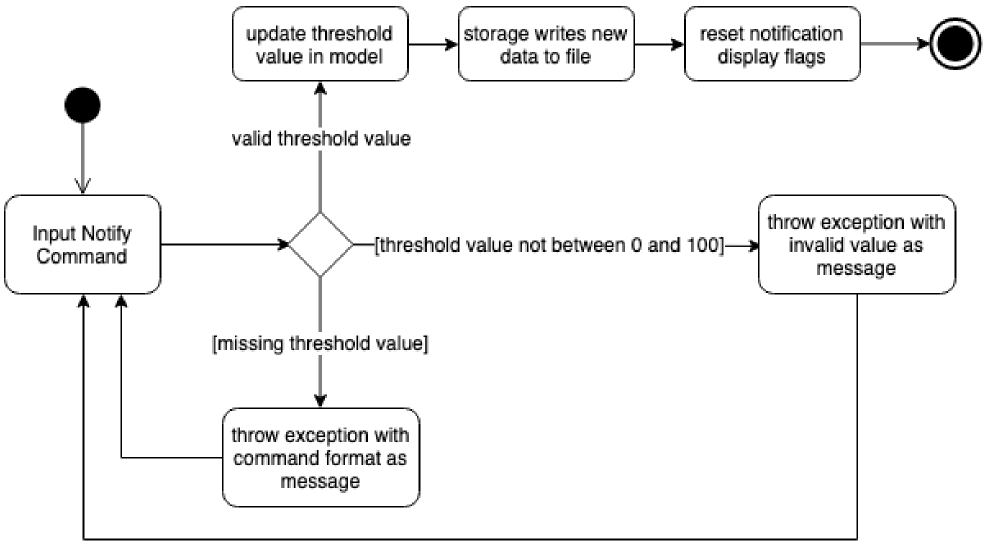

update threshold value in model
storage writes new data to file
reset notification display flags
valid threshold value
Input Notify Command
[threshold value not between 0 and 100]
throw exception with invalid value as message
[missing threshold value]
throw exception with command format as message

Figure 54. Activity Diagram of Notify Command in the Overview Package

3.6.3 Design Considerations

In order to display the summary statistics shown to the user within the Overview tab, they must first be calculated. Several design considerations were made as to how these calculations would be made, as shown in the table below:

Alternative 1	Alternative 2 (Current Choice)	Conclusion and Explanation
The summary statistics are calculated by retrieving the transaction and item lists. Each transaction or item is then read individually and their totals added up. **Pros**: This is a much simpler, straightforward implementation. **Cons**: It is difficult to filter the list by a given criteria, which is required for certain summary statistics.	The Java Streams library is used to calculate the summary statistics, by retrieving the transaction list and item list as streams instead. **Pros**: It is much easier to filter the list from a given criteria with the built in `.filter()` method, and additional criteria can easily be added simply by adding additional `.filter()` methods. **Cons**: Java Streams run slower than their iterative counterparts when the list is small, and are more complex to implement.	Alternative 2 is selected as the performance difference is negligible for smaller lists, and will benefit the user in the long run as their lists of transactions and items become larger and larger. Additionally, it allows for future extensibility of summary statistics, as new statistics can be created simply by modifying or adding on new criteria.

The following is a code snippet of the chosen implementation above:

```
public double getTotalExpenses() {
    Stream<Transaction> transactionStream =
transactionLogic.getTransactionList().stream();
    return transactionStream
        .filter(transaction -> !transaction.getCategory().equals("Sales"))
        .filter(transaction -> transaction.isNegative())
        .flatMapToDouble(transaction ->
DoubleStream.of(transaction.getAmount()))
        .sum() * -1;
}
```

A design consideration was also made for the implementation of the notifications that are to be displayed to the user upon hitting the notification threshold. These are shown in the table below:

Alternative 1	Alternative 2 (Current Choice)	Conclusion and Explanation
A new class is created to act as a notifier, and is called after the execution of every command to check if any notifications need to be displayed to the user. **Pros**: All tabs can utilize this notifications feature and display messages to the user when needed. **Cons**: Extra program resources are needed to create such a class.	A method within the Overview tab's `Logic` is called to check if any of the notifications thresholds have been met. **Pros**: Easy to implement with minimal new resources required. **Cons**: It will be difficult to extend this functionality to other tabs if needed.	Alternative 2 was implemented after a discussion held with the team revealed that this functionality was and would not be needed for any of the other tabs. Thus, it made more sense to stick with the simpler, less resource intensive implementation of this function,

The following is a code snippet of the chosen implementation above:

```
private void checkIfNotify() {
    List<OverallCommandResult> notifications =
overviewLogic.checkNotifications();
```

```
    for (OverallCommandResult notif: notifications) {
        lion.setResponse(notif.getFeedbackToUser());
    }
}
```

3.7 [Proposed] Undo/Redo feature

3.7.1 Proposed Implementation

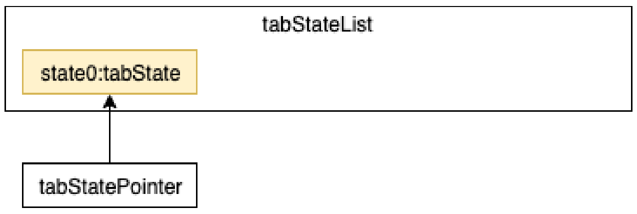
The undo/redo mechanism is facilitated by `VersionedtreasurerPro`. It extends each tab's `Model` with an undo/redo history, stored internally as a `tabStateList` and `tabStatePointer`. Additionally, it implements the following operations in each tab's `ModelManager`:

- `ModelManager#commit()` — Saves the current tab's state in its history.
- `ModelManager#undo()` — Restores the previous tab's state from its history.
- `ModelManager#redo()` — Restores a previously undone tab's state from its history.

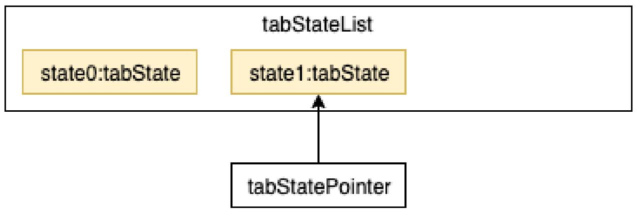
These operations are exposed in the `Model` interface of the tab as `Model#commit()`, `Model#undo()` and `Model#redo()` respectively.

Given below is an example usage scenario and how the undo/redo mechanism behaves at each step.

Step 1. The user launches the application for the first time. Each package's model will be initialized with their initial default state, and the `tabStatePointer` of each package pointing to that state.

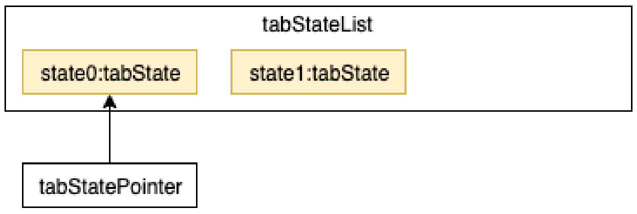


Step 2. The user executes a command that modifies the application state. The command calls `Model#commit()`. The state after executing the command is saved in the `tabStateList`, and the `tabStatePointer` is shifted to that new state.



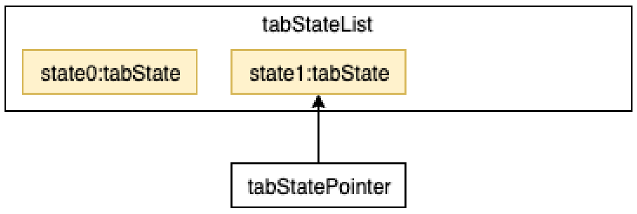
Commands that do not modify the application state will not call `Model#commit()`, and thus no new state will be created in `tabStateList`. The `tabStatePointer` will continue to point at the same state.

Step 3. The user executes `undo` to undo the last command. The `undo` command calls `Model#undo`, which shifts the currentStatePointer to the previous state, restoring it to that state.



If the `tabStatePointer` is already pointing to the first item in the list, an error will be returned to the user and nothing will be done.

Step 4. The user executes `redo` to redo the last `undo` command. The `redo` command calls `Model#redo`, which shifts the currentStatePointer to the next state, restoring it to that state. In this case, it is identical to after Step 2.



3.7.2 Design Considerations

In order to implement the undo/redo command, several alternatives in design were considered before settling on the current implementation:

Alternative 1 (Current Choice)	Alternative 2	Conclusion and Explanation
The state of the tab is saved after every command. **Pros**: This is much easier to implement. **Cons**: Extra memory is used to store each state after every command.	Each command is capable of undoing/redoing itself. **Pros**: Less memory is required. **Cons**: The implementation of each command becomes more complex, as it must be able to correctly undo/redo itself.	Alternative 2 is selected due to its ease of implementation. Most modern computers also have sufficient memory space to store extra states.

{End of second extract from Developer Guide}