Tan Zhan Ming - Project Portfolio for FinSec

1. Overview

1.1 Introduction

This project portfolio serves to provide a sample of my technical software engineering skills, by showcasing the work I have done on FinSec - the financial secretary application. FinSec is developed by a group of 4 students, including myself, as a software engineering project for the module CS2103T as part of our studies in the National University of Singapore.

The aim of the project was for us, as students, to be able to contribute production quality work to a small to medium software project. This was done by letting us morph an existing product - the AddressBook Level 3, into a product of our choice under certain constraints to mimic a realistic development environment. The main constraint is as follows: The application should be a CLI-based desktop application. This meant the application should be based around working on a single computer, and should target users who can type fast or users who prefer typing over other means of input.

1.2 About the product

FinSec is a finance planning and management application that can keep track of various financial statistics such as claim amounts, revenue streams and overall budgets. FinSec is targeted at finance secretaries and accountants of small organisations, such as those belonging to school clubs or even Small-Medium Enterprises, and aims to replace Microsoft Excel as their preferred tool to keep track of finances by providing a core set of specialised features. With its main form of input being the Command Line Interface (CLI), FinSec also favors people who can type fast.

FinSec's main features are as follows.

- FinSec can store information about your expenditure, revenue streams and even point-of-contacts by processing them as Claims, Incomes and Contacts respectively.
- FinSec splits the Contact, Claim and Income tabs for improved readability and allows you to easily switch between them.
- FinSec can generate prospective Budget values and even provide you with a graph outlining the monetary amounts of Claims, Incomes and Budget for the current month.
- FinSec can map any command to a Shortcut, allowing you to shorten lengthy commands or map them to words that are easier for you to remember.

The following image shows how FinSec's user interface looks like:

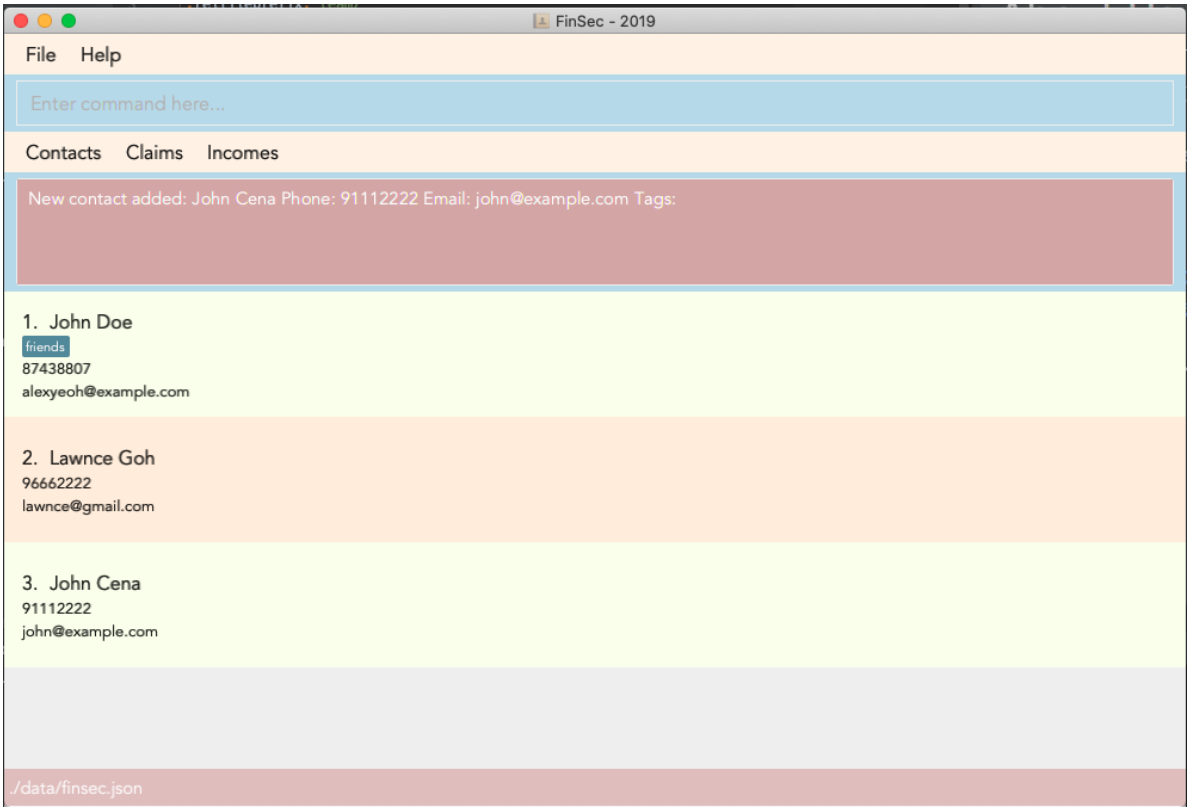

Figure 1. Main window of FinSec

1.3 Understanding this document

Note the following symbols and formatting used in this document:

`budget`	A grey highlight (called a mark-up) indicates that this is a command or parameter that can be input into the command line and executed by the application.
`HelpWindow`	Bold text with mark-up indicates that this is a class/package name or a method name.

2. Summary of contributions

My main role was to design and develop the `help` and `budget` commands. This section shows a summary of my coding and documentation efforts toward that end, and also highlights other helpful contributions that I had made towards FinSec.

2.1 Enhancements made

- **Major enhancement 1**: I added the **ability to calculate budget projections**
  - What it does: The `budget` command calculates your prospective budget - the balance of money you are expected to have after handling all your claims and incomes.
    Besides calculating the overall budget, the `budget` command also displays claim, income and budget statistics for the current month via a graphical representation.
  - Justification: The main point of a finance management application like FinSec is to keep track of the cashflow in your organisation. The `budget` command does exactly that by calculating and showing you your overall budget. By providing a clear graphical representation at the same time, it allows you to easily identify certain problems that might arise in the current month - such as the need to raise finance or cash flow difficulties.
  - Highlights: This graphical portion of the `budget` command is specifically designed to be incredibly visual and easy to understand. This is done by varying the colour and width of the lines used to plot the graph and by implementing tooltips on each data-point.
    The implementation was quite challenging as the library I used for the graphing was not a native one. Thus, I had to improvise a lot, such as by creating my own custom tooltips for the graph as the default ones provided did not provide as much clarity as I had wanted.
  - Credits: I would like to thank [JFreeChart] for their amazing open source library. I had a much easier time utilising their API to create a graph rather than trying to do so with JavaFx.

- **Major enhancement 2**: I updated the existing `help` command to **provide 3 types of help for each command**.
  - What it does: The `help` command by default now opens up a `HelpWindow` containing a list of all the FinSec commands, while the advanced version of `help` with parameters can now provide 3 different types of assistance.
    `brief` - Gives you a short summary of what the command does, how to use it and lists an example.
    `guide` - Opens up your default web browser and brings you to the command in question in the FinSec User Guide
    `api` - Is for advanced users who wish to know more about the inner workings of the commands
  - Justification: The previous implementation of `help` was sorely lacking in that it relied solely on the user guide and thus was dependant on internet connectivity. This has been remedied with both the `brief` and `api` types of help, which cater to offline users.
    The `guide` functionality has also been improved to bring you to the exact part of the user guide pertaining to the requested command for added convenience.
  - Highlights: The `api` form of help was designed from the ground up to be as unobtrusive as possible. It will only generate a single html file onto your computer, and replaces it for every command thereafter to prevent excess file cluttering.

2.2 Code contributed

Please follow these links to see my code: [Commits] [Pull requests] [RepoSense Code Contribution Dashboard]

2.3 Other contributions

- Project management
  - Managed product releases v1.3 and v1.3.2 on GitHub.
- Enhancements to existing features
  - Updated the original `HelpWindow` to display the command list and to take the user to the FinSec user guide directly instead of simply providing them with a link. (Pull request #78)
- Documentation:
  - Contributed multiple diagrams in the user guide and developer guide of FinSec to explain the execution and implementation of the `budget` and `help` commands. (Pull requests #189, #193)
- Community:
  - Helped pace pull requests and ensure adherence to deadlines (Pull requests #52, #142)
- Tools:
  - Integrated a third party library (JFreeChart) to the project (Pull request #176)

3. Contributions to the User Guide

> The following section shows a small extract of what I have written in the FinSec user guide. This is to illustrate my ability to write clear and concise end-user documentation by guiding users on how to use the `help` command.

{Start of extract for `help` command}

Viewing help : `help`

Provides 3 different types of help for all commands in FinSec.

Keyword: `help`

Format: `help cmd/COMMAND type/TYPE`

Refer to [Command Summary] for all available commands

Types include:
`brief` (gives a brief description)
`guide` (opens a web browser and bring you to our user guide)
`api` (for advanced users who want to know the inner workings of the command)

Cant remember all of FinSec's commands or the command format for `help`? No worries! Even if you mistakenly type `help` without the other parameters or get the parameters confused, a default help window such as the one below will appear and display a command list with similar instructions to this page!

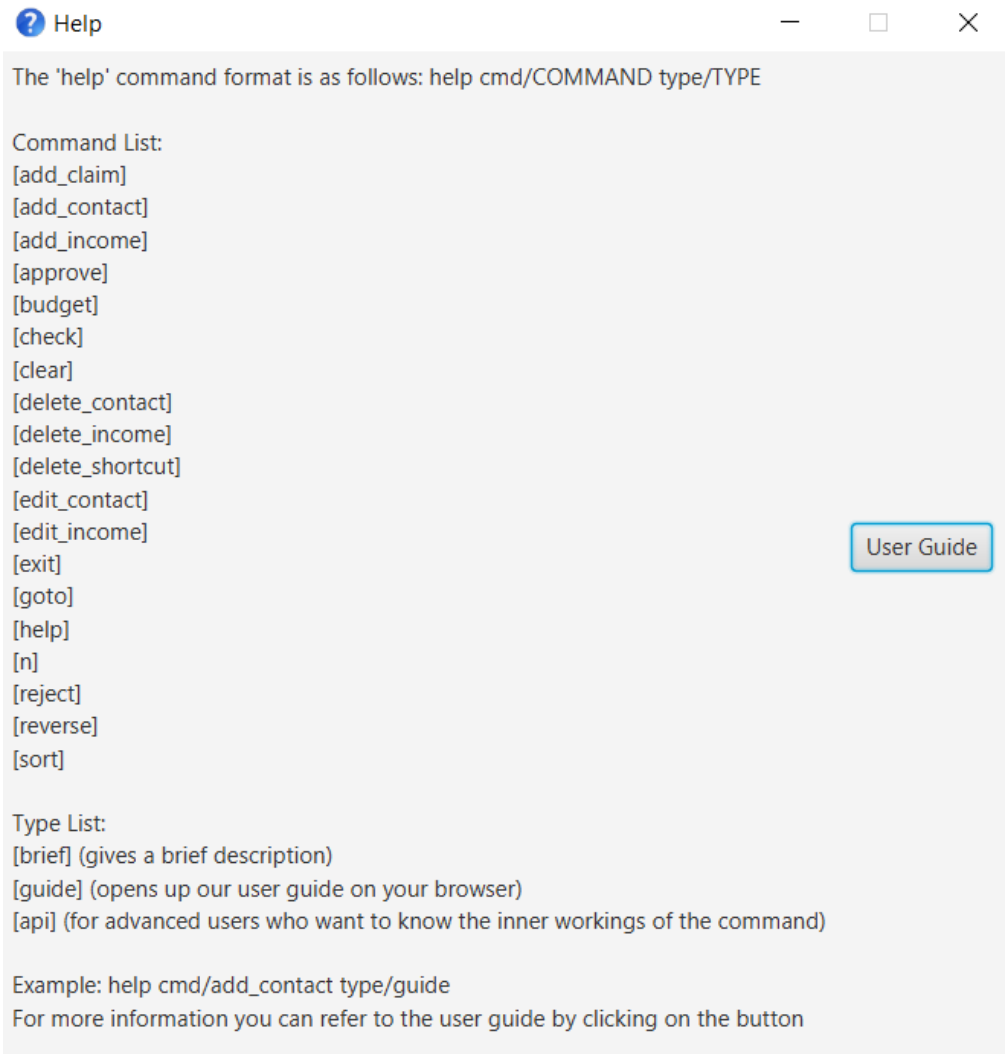

Figure 3.1.1: The help window that appears when the user asks for the default form of `help`

Instead of typing in `help`, you can also access help by clicking on the `help` button on the top left of the FinSec application GUI or just press `F1` on your keyboard as shown below!

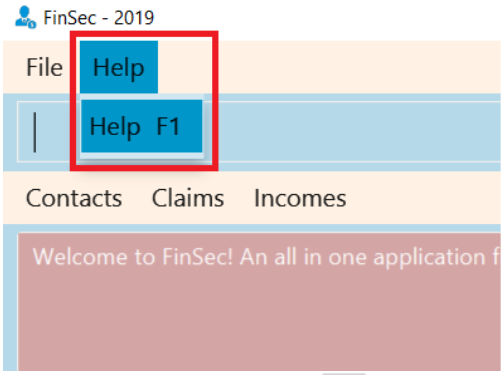

Figure 3.1.2: The location of the `help` button on FinSec's GUI

Example commands:

`help cmd/add_contact type/brief`

This shows you a brief description of what the `add_contact` command does and how to use it

`help cmd/goto type/api`

This generates an 'API.html' file containing our API for the `goto` command, and opens it up

`help cmd/help type/guide`

This opens up a page in your browser and brings you right here to this section!

Figure 3.1.3 shows what you can expect to see when typing in the first example: `help cmd/add_contact type/brief`.

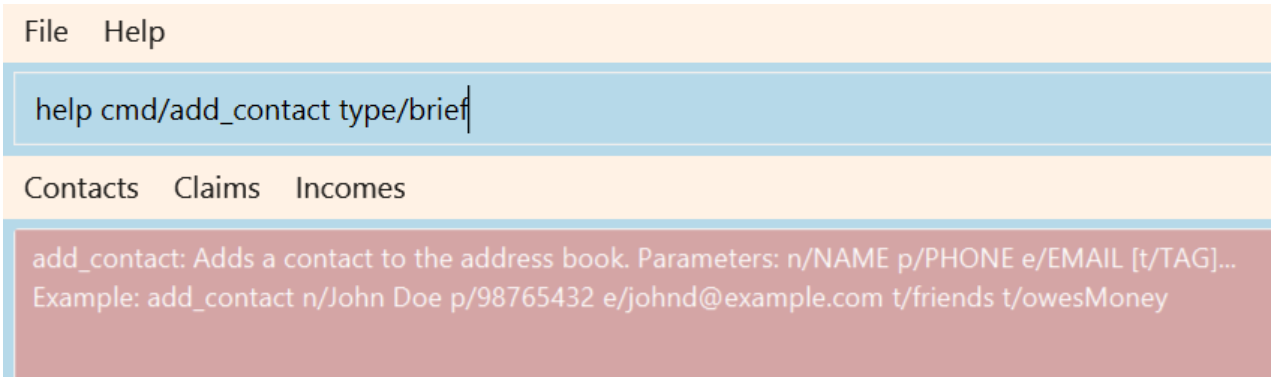

Figure 3.1.3: FinSec giving a brief description of the `add_contact` command

{End of extract for `help` command}

If you are interested to find out more about my end-user documentation skills, you can click on the link below to view more of my handiwork; this time detailing the `budget` command in the FinSec user guide. (UG-budget)

4. Contributions to the Developer Guide

> The following section shows a small extract of what I have written in the FinSec developer guide. This is to illustrate my ability in writing comprehensive technical documentation, and how I am able to provide developer insight on the design and development of the `budget` command.

{Start of extract for `budget` *command}*

Budget feature

The `budget` command allows for users to Generate a **`Budget`** object in FinSec. It also creates a **`Budget Graph`** object and displays it via the User Interface.

Overview

The **`Budget`** feature relies primarily on the **`Claim`** and **`Income`** features, and serves as an extension to calculate their difference. The **`Budget`** object calculates the cash amount values of all existing **`Income`** objects and all cash amount values of **`Claims`** that have a status of 'approved'. It then returns the difference in values as the budget value and creates a graph detailing the statistics for the month.

Current Implementation

Figure 2.6.2.1 is a sequence of steps that illustrates the interaction between various classes when the `budget` command is entered.

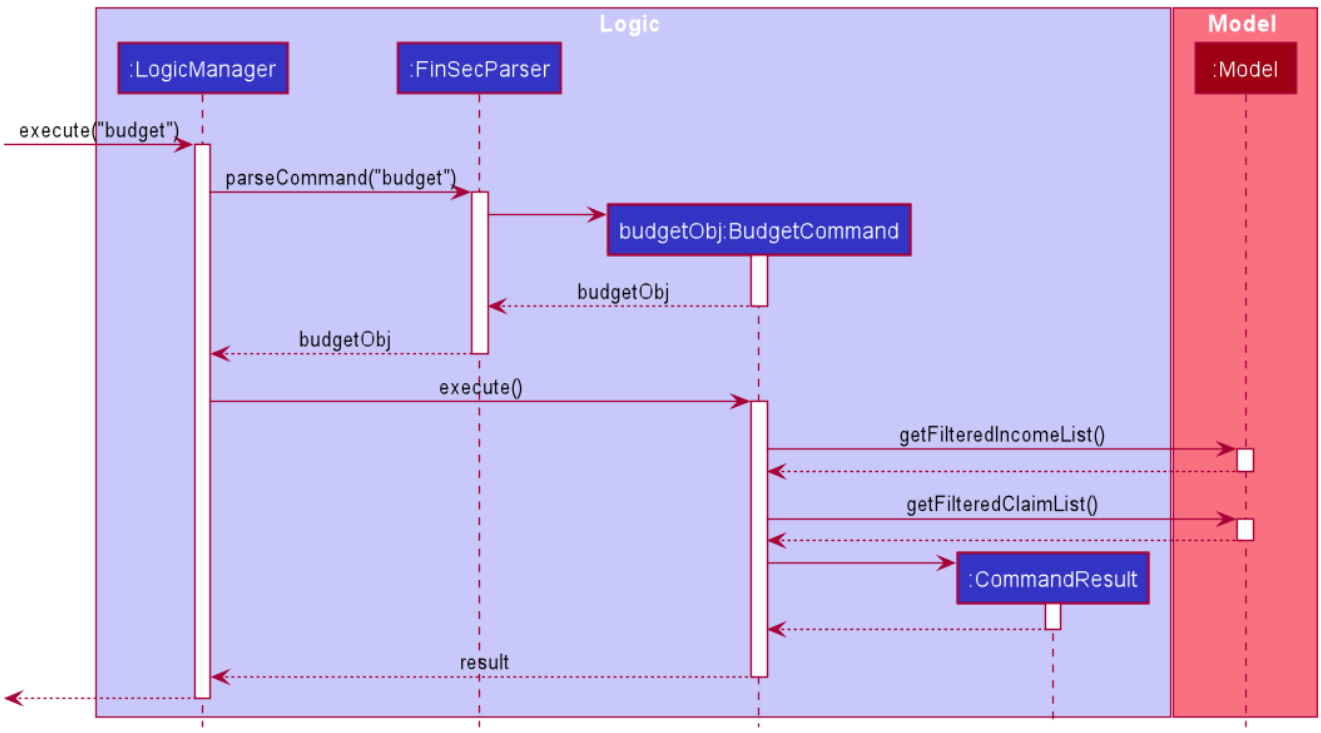

Figure 2.6.2.1: Execution sequence of the `budget` *command*

Step 1 : The `budget` command is passed on to the `LogicManager` as commandText

Step 2 : The `LogicManager::execute` method then calls `FinSecParser::parseCommand` which receives the user input (`budget`) as a parameter.

Step 3 : FinSecParser then references the various command words and identifies the command to be a `budget` command . It then calls the `BudgetCommand` class.

Step 4 : This newly created `BudgetCommand` object is returned to the `LogicManager` class, which then calls the `BudgetCommand::execute` method.

Step 5 : The `BudgetCommand` then interacts with the model component of our software architecture to create a filteredList of all `Income` and `Claim` objects using the `model.getFilteredClaimList()` and `model.getFilteredIncomeList()` methods.

Step 6 : It instantiates a `Budget` object which contains methods such as `calculateTotalExpenses()` and `calculateBudget()` to calculate the amount values of all the `Claims`, `Incomes` and thus use them to find the budget amount.
A `BudgetGraph` object is also created in parallel (Details expanded upon below)

Step 7: The `BudgetCommand::execute` finally completes by constructing a message string containing all these values and returning a new `CommandResult` with the specific message string to its calling method which is `LogicManager::execute`.

Step 8 : `LogicManager::execute` method returns a `CommandResult` to the calling method which is `MainWindow::executeCommand`.

Step 9 : The specific feedback is then retrieved through `CommandResult::getFeedbackToUser` and set in the result display of the `MainWindow`.

While creating the `Budget` object, a `BudgetGraph` object is also created in parallel. The activity diagram below shows how it would look like from a user's point of view.

Figure 2.6.2.2 describes the workflow of FinSec when the `budget` command is entered.

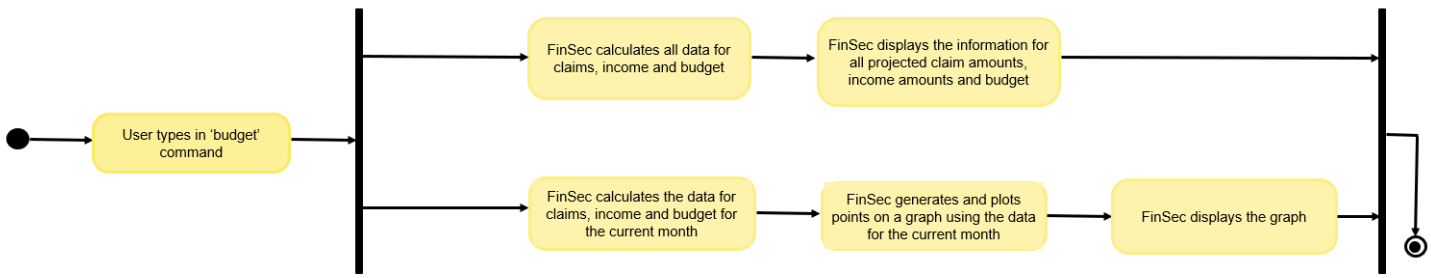

Figure 2.6.2.2: Activity diagram of the `budget` command

The series of steps below demonstrates what the `BudgetGraph` object does in parallel to Step 6 above.

Step 6a : At the same time the `Budget` object is created, the `BudgetGraph` object is also instantiated, which is basically an XY-graph.

Step 6b : The `BudgetGraph` object creates a dataset by taking in the list of `Claims` and `Incomes` and parsing them to the `ClaimPlotter`, `IncomePlotter` and `BudgetPlotter` classes.

Step 6c : The 3 plotter classes then filter their respective lists to create new lists for the current month and start adding the points to the series.

The code snippet below shows the `ClaimPlotter::plotClaims` method

JAVA

```
    XYSeries plotClaims() {
         Double amountToAdd;
         findClaimValueAtStartOfMonth();
         claimSeries.add(1, startingExpenses);
         double currentExpenses = startingExpenses;
         List<Claim> approvedClaimsInCurrentMonthList = findApprovedClaimsInCurrentMonth();
         for (int day = 2; day <= 30; day++) {
             for (Claim claim : approvedClaimsInCurrentMonthList) {
                 if (claim.getDate().date.getDayOfMonth() == day) {
                     amountToAdd = Double.parseDouble(claim.getAmount().value);
                     assert amountToAdd >= 0 : "A negative claim value managed to get into the
claim list";
                     currentExpenses += amountToAdd;
                     currentExpenses = Math.round(currentExpenses * 100) / 100.0;
                 }
             }
             claimSeries.add(day, currentExpenses);
         }
         return claimSeries;
     }
```

Step 6d : Once the 3 series have been returned, the plotter classes then return the completed dataset to the `BudgetGraph` class which then renders the image.

Step 6e : The `BudgetCommand::execute` method then calls the `BudgetGraph::displayBudgetGraph` method to display the graph image.

Why was it implemented this way?

With so many claims and incomes, all having differing dates, it can be hard to keep track of how much money one should have on hand at any one time.

- We felt that while knowing how much our prospective budget would be is good, knowing it over a range of time (such as a month in the case of `BudgetGraph`) would help with better planning
- We also wanted to keep track of the history of said `Claims` and `Incomes` and doing it over a 1 month period ensures there will not be too visual data cluttering the screen.

Design Considerations

We have considered between two differing graph designs.

Table 1. Graph Designs

Graph Design Considerations	**Pros and Cons**
Single Graph (Current Choice)	**Pros** : Clean and clutter-free display. **Cons** : It does not display as much data
Separate Graphs based on Organisation Tags	**Pros** : Displays all relevant data that the user can possibly ask for **Cons** : Opening a multitude of graphs will visually clutter the screen with data unless more parsing is done to sort out which graphs are required

We have settled on adopting a single-graph approach as having multiple graphs open can lead to the user being overwhelmed by unnecessary data, and the code needed to achieve this result satisfactorily would be too convoluted.

{End of extract for `budget` command}

If you are interested to find out more about my technical documentation skills, you can click on the link below to view more of my handiwork; this time detailing the `help` command in the FinSec developer guide. (DG-help)

5. End of Project Portfolio Page

Thank you for reading my Project Portfolio Page. If you have any questions on FinSec or about any of my documentation work, you can contact me at e0349703@u.nus.edu.

Special thanks to my team for making this project a success!