Wong Ren-Wei, Ryan – Project Portfolio

PROJECT: MisterMusik

Overview

MisterMusik is a Command line interface (CLI) based application that allows for music students studying at the National University of Singapore(NUS) to have a centralized scheduling platform for all their music based activities. The user interface is entirely command line based and is written in java.

My roles were the following:

- Regarding back-end code:
    - To create proper exception handling and string validation to tackle cases of invalid user input.
    - To create a sorting function so that the list of events in the scheduler is always sorted in increasing order by date.
    - To create a budgeting system so the user can keep track of monthly concert costs and set a budget to ensure that he/she does not overspend.
    - To create a proper system that automatically prevents schedule clashes whenever changes are made to the scheduler.
- Regarding project management
    - To set a clear goal and vision of an achievable and desirable end-product.
    - To coordinate teammates on the integration of different implementations.
    - Setting up of Gradle project necessary for build automation and jar packaging.
    - To ensure that the jar file was functional at each point of development

Summary of contributions

Overall code contributions: https://nuscs2113-ay1920s1.github.io/dashboard/#search=ryan-wong-ren-wei

Major enhancement: Added a concert budgeting system to the application.

- What it does:
    - Allows the user to keep track of how much he/she is spending on concerts each month, and to set a stipulated monthly budget that the system

automatically helps the user to adhere to. If the user attempts to add a concert to the scheduler that exceeds the stipulated budget for the month, he/she is informed that the concert exceeds the budget and the operation is cancelled.

    - Allows the user to view the total amount of money he/she intends to spend on concerts for any specified month, based on the existing concerts previously entered into the scheduler.

- Justification: This feature improves the product significantly because a user can freely plan their schedule without having to worry about constantly checking if they have exceeded their budget. This allows for more freedom for the user to focus on more important issues within schedule planning.
- Highlights: One of the problems I faced initially was that I needed a solution that could store the concerts in a way that was easily accessible via a specific month and year (so that the user could view his concert costs for a specific month and year). In order to do this, I implemented a map where the key was a string in the format "MM-yyyy" and the value was a object MonthlyBudget that stored all the pertaining information for that month regarding concerts and concert costs.

[code contributed]

[test code contributed]

Major enhancement: Added an automatic clash detection system to the application.

- What it does: Automatically notifies the user of any clash in the schedule when he/she attempts to add a new event to the scheduler. If a clash is detected, the user is informed of the event that caused the clash. This automatic clash handling is also triggered when the user attempts to reschedule an existing event.
- Justification: This feature improves the product significantly because a user can freely plan their schedule without having to worry about constantly checking if there are any schedule clashes with his/her intended plan.
- Highlights: One of the initial problems faced was that there wasn't a proper way to manually pass data detected in a schedule clash back to the Command and then the UI class so that the information could be disseminated to the user. I ended up using exceptions to handle this problem. Each time a clash is detected, an exception is thrown up the chain. When caught, the exception contains the event that caused the

clash, and this can be accessed and passed along to the user.

In addition, because of the way that clashes are handled, any incremental extension to the application that involves the addition of a new event to the list, where the event is new or pre-existing with changes made, can be easily coded to include checking for a clash via exception handling. This was due to the clash handling being in a way that was easily scalable in terms of adding new features that required this functionality.

This can be seen in the following implementation (done by YuanJiayi) who implemented a rescheduling function that made use of the clash handling system to ensure the user did not inadvertently create an illegal schedule clash within the list when changing the date/time of a pre-existing event. [code here] As seen in the code, implementation of this feature was very smooth as all that was required was to remove the Event object, create a duplicate with a different date (reschedule) and then add it back into the list. This automatically triggers the clash handling system, throwing an exception if a clash is detected.

[code contributed]

[test code contributed]

Minor enhancement: Added an automatic sorting system to the application.

- What it does: Automatically sorts the schedule according to date each time the user makes any changes to the schedule.
- Justification: This feature allows for the user to prioritize his/her schedule properly especially in a time-sensitive environment.
- Highlights: Having a sorting functionality required that each of the instantiated Event objects in the list was comparable in terms of date, which had to be done manually so that java's Collections api could be exploited to efficiently sort the list when changes were made.

[code contributed]

Other contributions

For the most part, throughout the course of the development phase, I played an involved role in detecting bugs that occurred when new implementations were pushed to the repository, as well as preventing coding standard violations with code reviews whenever possible.

I also led my teammates in pursuing a clearly defined vision of what our product was to look like in the end. This allowed us to focus on achieving a common goal and greatly reduced the number of arguments within the group regarding what features were good or not good for our product.

I also contributed much to project management by setting up the project as a Gradle project, and ensuring that the jar file functioned correctly and was packaged with the necessary resources for each release.

Contributions to Developer Guide and User Guide

USER GUIDE:

3.17. viewing monthly costs of concerts : `budget`

The total cost of concerts each month can be viewed by the user using the `budget` command.

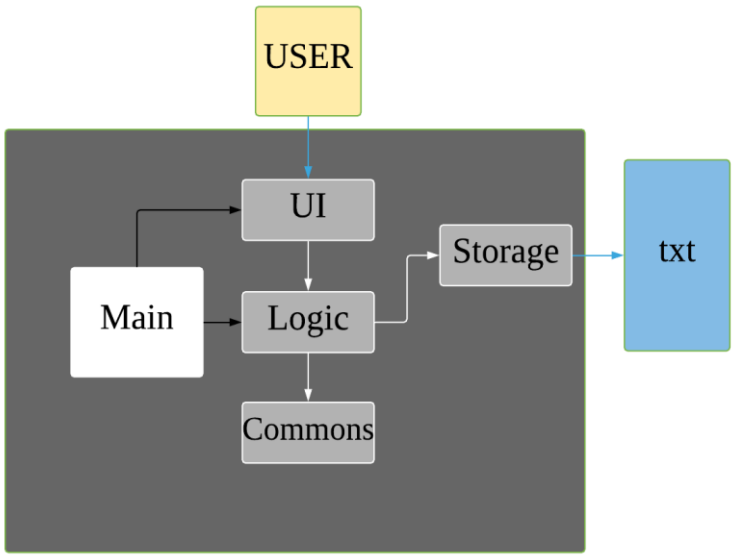
Format: `budget MM-yyyy` symbolising the month and year the user wishes to take a look at.

Note: MM must be a two digit value. For example, the month of may, 2019 must be entered as `05-2019`instead of `5-2019`.

DEVELOPER GUIDE:

2.1. Overall Architecture

This image shows the overall architecture of our program



2.2. Main component

Main component has one class called `Duke`. It is responsible for the following:

1. On startup: Initializes all components and sets up the correct file path so that the program correctly interacts with the external txt file.
2. During runtime: Acts as an intermediary between the `Parser`class and the `Command` so that user input can be parsed and then executed accordingly.
3. On shutdown: Interacts with the UI class to communicate the shutdown message to the user.

2.3. UI component

UI component contains all classes necessary to interact successfully with the user.
The `Parser` parses input commands from the user whilst the `UI` class handles all necessary dissemination of information to the user through `System.out`.

2.4. Logic component

This component contains all necessary classes that :

1. Are in charge of handling how all necessary information is internally stored within the program's runtime.
2. Alter the internally stored information whenever necessary (i.e when changes are made by the user).
3. Extract information requested by the user from the stored information and pass them back to the user through the UI component
4. This is achieved through the two classes `EventList` and `Command`. `EventList` decides how information is stored

internally as well as how internal stored information is altered and/or extracted. `Command` commands the `EventList` class on what needs to be done at any point in time.

2.5. Storage component

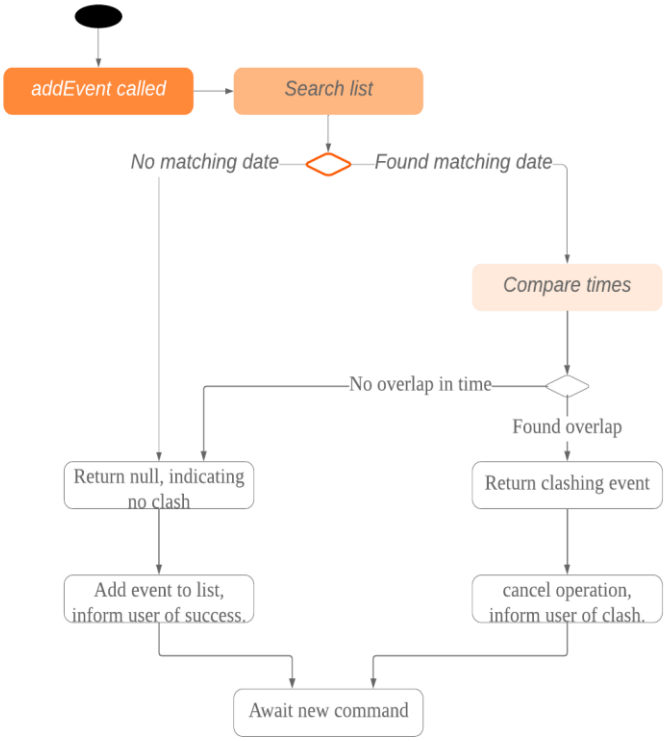
This component contains all necessary classes that read and write external txt files. This is where all information is stored while the program is not running.

In particular, the `Storage` class directly reads from and writes to and external txt file in the data directory.

3.4. Clash handling

3.4.1. Activity diagram

The following activity diagram represents a typical clash handling scenario



3.4.2. How is it implemented

The program is able to detect clashes when creating new events. When the user enters the command to add a new Event entry to the list, the method EventList.addEvent is called from the Command class object upon execution.

The addEvent method will then call the EventList.clashEvent method to check the existing entries for any clash in schedule. This is done by first searching the list for an event that has a matching date with the new event.

If no such event is found, the method returns a null value, indicating that there is no schedule clash. If an event is found with a matching date, the clashEvent then calls the EventList.timeClash method to check whether the two events have overlapping time periods.

If there is any overlap, the timeClash method will return true as a boolean, indicating there is a schedule clash. The clashEvent method then throws an exception ClashException, indicating that there was indeed a schedule clash between the desired new entry and some pre-existing Event.

The details of the clashing Event are passed back to Command object via the exception so that it can be used to inform the user about the clashing event. The user is then required to fix the conflict before continuing, either by rescheduling or deleting the pre-existing event, or by choosing a different date/time for the new Event entry.

3.4.3. Why it is implemented this way

The process of checking for a clash was implemented as small, simple components so as to ensure scalability, reliability, and to reduce dependencies.

The choice to use exception handling to deal with an event clash was done so that it could be easily re-purposed for any incremental extension that required checking for a schedule clash. Catching of the ClashException should be performed in the Command class, and the info regarding the schedule clash can be easily obtained for further action.

By having the clashEvent method return a null value or a reference to a clashing event in the schedule, the clashEvent method can now be used for any further increments to the code requiring addition of events.

It was thus easy to implement this clash detection as a part of adding recurrent events (to check for clashes when recurrent events were automatically entered) as well as the rescheduling function (to check for clashes when the user attempts to reschedule an existing event, so that he/she does not inadvertently create a new schedule conflict).

This implementation is reliable because it can always be expected to work whenever it is necessary to add new events to the list, provided the unit tests for this functionality under

MainTest work. It is also not dependent on any other functionality, allowing for developers to change the implementation of other parts of the application without affecting the clash handling

3.4.4. Expected behaviour of functionality

When a user attempts to add an event(recurring or otherwise) and the program detects a clash with an existing event in the pre-existing list, the following output should be printed: "That event clashes with another in the schedule! Please resolve the conflict and try again!" This is followed by the following line indicating the details of the detected clash: "Clashes with: [E][X] YST Final project review START: Tue, 03 Dec 2019, 15:00 END: Tue, 03 Dec 2019, 18:003.4.5. Design considerations

While designing the clash handling system, i had to decide how best to: 1. Detect a clash. 2. Pass relevant information back to the caller class for further usage.

Aspect	**Alternative 1**	**Alternative 2**
How clashes were detected	Simple if-else statements instead of using exception handling. Pros: Much easier to implement and simpler to work with. Cons: Less scalability as it would be harder to integrate the functionality into new features.	Have a specific command/method so that the current list can be checked for a clash at will. Pros: Much more flexible, and even more scalable than the current implementation as it would be possible to do anything regarding changes to the list, and then later check for clashes. Cons: Much more room for error, since clashes are allowed to exist within the list normally, and are not automatically detected. This could lead to major bugs related to events that overlap each other.
How relevant information is passed up the chain	Simply returning the relevant event that caused the clash as a part of the method call. Pros: Extremely simple to understand. Easy to implement. No need to deal with exception handling, just code for specific case in the event of a clash. Cons: Less scalable. If a developer wants to add more functionality to the clash handling(e.g return more data), he/she would need to return a new object containing the relevant data.	