

# Gabriel Seow - Project Portfolio

## 1. Overview

This project portfolio documents the key features I implemented for +Work, an app designed for project managers to manage their teams with less hassle. This app was designed as part of the a team project for the NUS module, *CS2103 Software Engineering*.

### 1.1. About The Team

The team consists of 5 students NUS Computer Science, currently in their 2nd year.

### 1.2. About The Project

+Work is a application designed with project managers in mind, and includes features directed towards making project management hassle-free. +Work allows users to easily view and manage tasks, team members and inventories, generate inventory reports and schedule team meetings, among other features.

+Work is also a Command Line Interface (CLI) application, and is best suited for users that are familiarised with the commands and can type reasonably fast.

The application is aimed at NUS project leaders, who would benefit from the inventory features, as their small to medium sized projects would require a student, typically the project leader, to take on the role of the treasurer. These student leaders would also benefit from the calendar feature, as their well-defined timetables will allow +Work to schedule the best meeting times according to their schedule.

## 2. Summary of Contributions

This section describes my contributions to the project, and showcases the variety of features implemented.

### 2.1. Enhancements

¥ Major enhancement: I added the Calendar features for +Work

- ! What it does: This feature allows users to import `.ics` files (Calendar format for Gmail and NUSmods) to +Work , where +Work implements the logic to find common available timings and generate timings when the most number of members are available.
- ! Justification: This feature greatly reduces the hassle of scheduling a team meeting, as project leaders no longer require the hassle of choosing sub-optimal meeting times and thereafter, gather responses from team members. +Work guarantees that the meeting times generated have the highest possible attendance rate.
- ! Highlights: It was very interesting working with calendar `.ics` files and observing how applications like 'Google Calendar' would store their calendars. However, implementing the

Calendar features required working with the `iCal4j` library, and required adding dependencies, configuring `ical4j.properties` and learning the API. Overall, the process was challenging, but rewarding as well.

¥ Major enhancement: I added the Undo/Redo feature for +Work

- ! What it does: This feature allows users to undo a command at any point of time, while using the application.
- ! Justification: This feature greatly improves the user experience, as any accidental deletions can be easily 'undone' with the short `undo` command. If the user realises he does not need to undo, he can also execute the `redo` command.
- ! Highlights: Learning how to implement the undo/redo feature required me to observe the project architecture, and allowed me to appreciate the well organised way in which the AB3 source code was structured. As +Work is based on the AB3 source code, it showed me the importance of a well-structured project.

¥ Major enhancement: I added support for mapping tasks, team members and inventories to one another

- ! What it does: This feature allows users to assign tasks to team members, or assign inventories under tasks. By creating 'associations' between the tasks, team members and inventories, users can view these 'associations' by seeing the tasks allocated to a team member, or the inventories allocated for a task.
- ! Justification: Assigning a task to a team member or allocating inventories under a task is commonplace in a project. Allowing users to capture these associations greatly improves the user experience when wanting to view additional information about the tasks or team members.
- ! Highlights: While implementing the mappings classes, I ran into problems deciding how to effectively store these mappings as Json objects. However, through discussions with the team, we came up with an efficient solution, to store index references to the task, team member or inventory, which would greatly reduce the storage needed.

¥ Code contributed: [[Functional code](#)] [[Test code](#)]

### 3. Contributions to the User Guide

The following section documents my contributions to the +Work User Guide for the `Undo/Redo` and `Calendar` features.

### 4. Calendar Commands

This section contains the commands for managing your team member's calendars and for scheduling a project meeting time.

Before you start using +Work's calendar commands, make sure

1. Your team members have exported their calendars as an .ics file

TIP

Not sure how to export a calendar? Refer to the short guide below for instructions

2. You have collected the calendar files from your team members
3. You take note of the file path where the files are stored, it will be used in the commands

How to export a calendars as an .ics file?

1. On the NUSmods page, click on the `Download` icon and select download as an iCalendar File(.ics)
2. If you wish to include other commitments, simply open 'Google Calendar', click on `Settings`, select `Import & Export` and import the file from Step 1
3. Add any additional commitments through 'Google Calendar'
4. Export the calendar again by clicking `Settings`, followed by `Import & Export` and lastly `Export`
5. The .ics file will be downloaded, containing your timetable from NUSmods and 'Google Calendar'

#### 4.1. Adding a team member's calendar: `add-calendar`

You can add a calendar by inputting the file path of the team member's calendar as well as the name of the team member.

Format: `add-calendar [mn/MEMBER_NAME] [fp/PATH_TO_ICS_FILE]`

Example:

Adding *John Doe*'s calendar to +Work

```
¥ add-calendar mn/John Doe fp/C:\Users\gabriel\TeamCalendars\john_calendar.ics
```

#### 4.2. Removing a team member's calendar: `delete-calendar`

You can also remove a calendar by specifying the team member's name.

Format: `delete-calendar [mn/MEMBER_NAME]`

Example:

Removing *John Doe's* calendar from +Work

¥ `delete-calendar mn/John Doe`

### 4.3. Find possible meeting times: `find-meeting-time`

You can schedule a meeting by specifying the duration of the meeting in hour(s) and the time period to search for.

+Work will show a list of suitable meeting time between `START_DATE` and `END_DATE`.

Format: `find-meeting-time [start/START_DATE] [end/END_DATE] [hours/DURATION]`

NOTE | +Work recognises date and time in the format 'dd-mm-yyyy hh:mm`

TIP | +Work shows you the meeting times where the most number of people are available

Example:

¥ Let's say you want to schedule a 2 hour meeting in the upcoming week, between *11th Nov 8 a.m* and *15th Nov 5 p.m*. After entering the details in the correct format, as such

¥ Hit `Enter` and +Work will display a list of suitable timings as well as the team members that are available for that timing

NOTE | If there are no suitable timings, +Work will notify you as well

IMPORTANT | Because showing ALL possible meeting timings may not be appropriate, +Work helps by restricting the meeting timings to be between 8 a.m and 10 p.m

#### 4.4. Schedule a team meeting: `add-meeting`

After using the command `find-meeting-time`, you can schedule a meeting from the list of possible timings by referring to the `INDEX` of the meeting in the list.

Format: `add-meeting [meeting/INDEX]`

Example:

¥ Suppose you are looking to schedule a 2 hour meeting between *11th Nov 8 a.m* and *15th Nov 5 p.m.*

¥ After using the `find-meeting-time` command, you are given the following timings

¥ After looking through the suitable timings, you choose meeting #5 as your preferred timing

¥ Using the index of meeting #5, enter the command `add-meeting meeting/5`

¥ You can then view the recently added meeting at the `home` page

#### 4.5. Remove a team meeting: `delete-meeting`

You can remove a meeting by simply referring to the `INDEX` of the meeting in the 'Upcoming Meetings' list.

**TIP** | You can view your list of meetings by going to the `home` page

Format: `delete-meeting [meeting/INDEX]`

Example:

¥ To remove meeting #3, simply enter the command `delete-meeting meeting/3` and the meeting will be removed

#### 4.6. Undo a command: `undo`

You can undo your recent commands by using the `undo` command

Format: `undo`

Example:

¥ Suppose you accidentally deleted task #6 using the `delete-task` command

⌘ Entering the **undo** command will bring back the deleted task

#### WARNING

Once you restart +Work, you won't be able to **undo** commands from the previous session!

### 4.7. Redo a command: **redo**

You can redo a previously **undone** command by typing **redo**.

Format: **redo**



Example:

¥ Let's say you deleted a task and you **undo** the command. You can simply use the **redo** command to delete the task again

## 5. Contribution to the Developer Guide

The following section documents my contributions to the +Work Developer Guide for the **Calendar** and **Undo/Redo** features. My contributions would hopefully demonstrate my ability to write a concise and well-documented guide for software developers to easily refer to.

## 6. Calendar feature

### 6.1. Implementation

This feature is implemented to allow users to easily schedule a meeting time, without the hassle of having to obtain responses from team members.

This feature includes basic commands for managing meetings and team member's calendars, i.e. **add-meeting**, **delete-meeting** and **add-calendar**, **delete-calendar** respectively. This feature also includes support to parse and import **.ics** calendar files, with help from the **net.fortuna.ical4j** library. The calendar feature also implements additional logic to compare member's calendars and generate possible meeting times where the most number of members are available.

#### NOTE

Team member's calendars in +Work are always handled and stored in a **CalendarWrapper** class, which also stores the name of the team member

Apart from the basic commands for managing calendars, the command for finding a meeting time is handled by **UniqueCalendarList**, while the logic for accessing external **.ics** files is handled by **DataAccess**. Finally, the logic for parsing **.ics** files is incorporated into **ParserUtil**

¥ **UniqueCalendarList#findMeetingTime(startDate, endDate, meetingDuration)** "Generates a list of possible meeting timings where the most number of members are available"

¥ **DataAccess#getCalendarStorageFormat(filePath)** "Converts an external **.ics** file into String format"

¥ **ParserUtil#parseCalendar(.ics String)** "Parses an **.ics** in String format to create a **Calendar** object"

Commands for generating meeting times and managing calendars or meetings are exposed in the **Model** interface in the following respective commands

¥ **Model#findMeetingTime(startDate, endDate, meetingDuration)**

¥ **Model#addCalendar(calendarToAdd)**

¥ **Model#deleteCalendar(calendarToRemove)**

¥ **Model#addMeeting(meetingToAdd)**

¥ **Model#deleteMeeting(meetingToRemove)**

Given below is an example usage scenario and how the more complex commands work.

Command: `Model #addCalendar (calendarToAdd)`

Step 1. The user calls the `add-calendar` command, which is handled by `AddCalendarParser`

Step 2. `DataAccess#getCalendarStorageFormat` accesses the file specified by the user and converts the `.ics` file into a `String` format

Step 3. The `.ics` file in `String` format is parsed using `ParserUtil#parseCalendar` and converted into a `net.fortuna.ical4j.Calendar` object

Step 4. The `Calendar` object is stored as a `CalendarWrapper` object together with the `MemberName` of the associated team member

Step 5. The `CalendarWrapper` object is passed to the `Model` and subsequently `ProjectDashboard`, where it is stored in `ProjectDashboard's UniqueCalendarList` instance variable.

Command: `Model #findMeetingTime(startDate, endDate, meetingDuration)`

The following sequence diagram shows how the `findMeetingTime` operation works:

*Figure 1. FindMeetingTimeCommand Sequence Diagram*

Step 1. The user calls the `find-meeting-time` command, which is handled by `FindMeetingTimeParser`.

Step 2. This creates a `FindMeetingTime` command that executes `Model #findMeetingTime`.

Step 3. The `findMeetingTime` command is passed from `Model` to `ProjectDashboard` and finally to `UniqueCalendarList`, where team member's calendars are stored.

Step 4. `UniqueCalendarList` handles the logic for comparing each calendar and generates a `MeetingQuery` object, which contains the list of possible meeting times and other essential information about the most recent `findMeetingTime` command.

## NOTE

Details on the logic for handling calendar 'events' and timings are excluded for the sake of simplicity.

Step 5. The `MeetingQuery` is then stored in `ProjectDashboard` where the user view can update and display the list of possible meeting times.

Step 6. Follow-up from user: The user can execute the `add-meeting` command to select a meeting from the list of timings, by referring to the `INDEX` of the meeting in the list.

## 6.2. Design Considerations

### 6.2.1. Aspect: Scheduling meetings based on tasks

¥ Alternative 1 (current choice): +Work assumes

! Pros: Easier to implement, files can be stored in application.

! Cons: User must enter file path, which is error prone.

¥ Alternative 2: Upon execution of `import-calendar` a file chooser pops up to allow user to browse and upload file.

! Pros: User can use UI to upload instead.

! Cons: Due to constraints of application, a ui based upload may not be feasible (Possibly in v2.0)

### 6.2.2. Aspect: Storing calendar data on +Work

¥ Alternative 1 (current choice): +Work preserves and stores the original calendar `.ics` file in `String` format

! Pros: Less error prone when converting from storage format to `Calendar` object format

! Pros: Captures more details about calendar 'events'. More compatible with additional v2.0 features, such as meeting location suggestions

! Cons: Requires more storage space when storing calendars

¥ Alternative 2: +Work only stores essential calendar information (i.e. duration and time of calendar 'events')

! Pros: Takes lesser time to retrieve `Calendar` objects from storage

! Cons: Harder to implement and requires manipulating `Property` and `Component` objects stored in `net.fortuna.ical4j.Calendar` objects

## 7. Undo/Redo feature

### 7.1. Implementation

This feature is implemented to allow the user to undo/redo a command, while improving the overall user experience.

This feature does not implement many additional functions. Rather, it takes advantage of the

existing project architecture, to achieve the according **undo** or **redo** outcome. This feature includes the basic commands **undo** and **redo**. The feature introduces the ability for the **Model** to store previous instances of the **ProjectDashboard**, essentially saving the 'state' of **ProjectDashboard**, similar to a commit on GitHub. The user then navigates between these 'states' when using the **undo** and **redo** commands.

NOTE

Each **ProjectDashboard** instance stores all information in **+Work**, which is why reverting to a previous **ProjectDashboard** instance does not result in any loss of data

The **undo**, **redo** mechanism is facilitated within **Model**, by including two variables **previousSaveState** and **redoSaveState** to store **ProjectDashboard** 'states' and the addition of the **Model #saveDashboardState** function. The **undo** and **redo** commands also make use of the **ProjectDashboard#resetDate** to revert the **ProjectDashboard** displayed by **+Work** to a previous 'state'.

¥ **previousSaveState**"~"Stores **ProjectDashboard** states from previous non-**undo** commands

¥ **redoSaveState**"~"Stores **ProjectDashboard** states from previous **undo** commands

TIP

**+Work** can only **redo** and **undo** command, if no command was executed after the **undo** command

¥ **Model #saveDashboardState()**"~"Saves the current **ProjectDashboard** state

¥ **ProjectDashboard#resetData(previousState)**"~"Resets the data of the current **ProjectDashboard** using data from the **previousState**

NOTE

Only the **undo** and **redo** commands are exposed in the **Model** interface. Other commands are used internally as part of the logic to manage **ProjectDashboard** states

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

Step 1. When the user starts up **+Work**, the **Model** does not store **ProjectDashboard** states from the previous session.

Step 2. When the user executes the **delete-meeting meeting/2** command, **Model #saveDashboardState** is called to save a copy of the original **ProjectDashboard** state, **pd0: ProjectDashboard** before executing the command. As shown below, the original **ProjectDashboard** state has been saved.

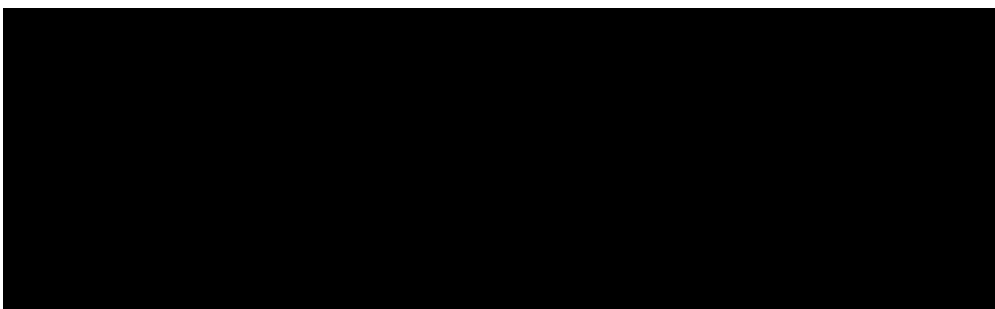


Figure 2. Storing previous **ProjectDashboard** states

Step 3. After executing another command i.e. **add-task tn/Complete DG**, a copy of the current **ProjectDashboard** state, **pd1: ProjectDashboard** is also saved and added to the list. The command continues execution on **pd2: ProjectDashboard**.



Figure 3. Executing more commands

NOTE

If an invalid command is entered by the user, `Model #saveDashboardState` is not called and the `ProjectDashboard` state is not saved.

Step 4. When the user realises he does not need to complete the DG, the user executes the `undo` command, reverting to the most recent `ProjectDashboard` state, `pd1:ProjectDashboard`. `ProjectDashboard#resetData` is called with the previous state. Previously `Current State` would have been pointing to the `pd2:ProjectDashboard`, where the 'Complete DG' task was added.

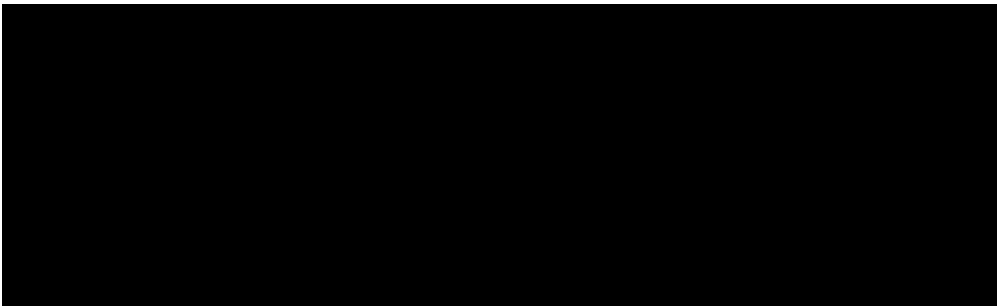


Figure 4. Retrieving a previous state

Step 5. The user can also execute the `undo` command again to revert to the original state, `pd0:ProjectDashboard`

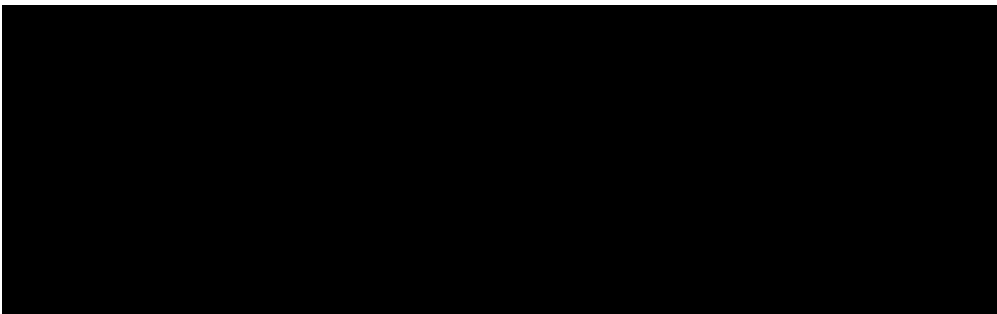


Figure 5. Retrieving previous states

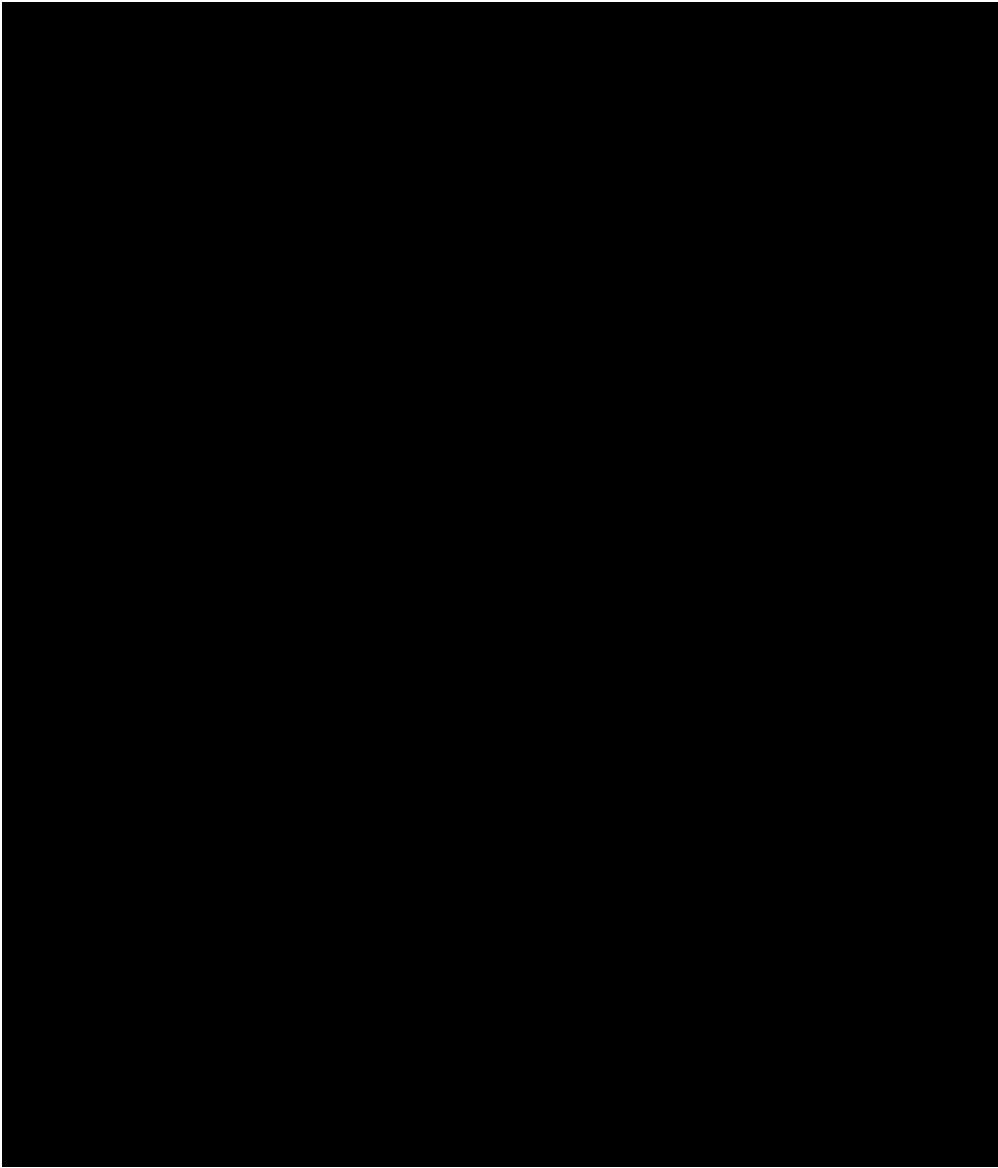
NOTE

If there are no more states to revert to, `+Work` will notify the user that there is no command to `undo`. The `undo` command uses `Model #canUndo()` to check if this is the case.

NOTE

The `redo` command works similar to the `undo` command, except it can only access `ProjectDashboard` states created by the `undo` command. In other words, `redo` can only be executed after an `undo` command.

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



*Figure 6. Executing a new command*

The following sequence diagram shows how the undo operation works:

*Figure 7. Interactions between Logic and Model*

## 7.2. Design Considerations

### 7.2.1. Aspect: Different implementations for undo & redo

¥ Alternative 1 (current choice): Saves the entire **ProjectDashbaord** object.

! Pros: Very easy to implement.

! Cons: May result in performance issues, when saving numerous instances of **ProjectDashboard**.

¥ Alternative 2: Each individual command has a **undo** counterpart.

! Pros: Uses much less memory, since the **Model** only has to keep track of which commands need to be undone.

! Cons: Prone to error, since +Work allows tasks, team members and inventories to be associated with one another. E.g. Trying to **undo** a deleted task may be unsuccessful in retrieving the original task.

### 7.2.2. Aspect: 'History' of **ProjectDashboard** and number of times **undo** can be executed

¥ Alternative 1 (current choice): Keep track of all past **ProjectDashboard** states

! Pros: Gives users the freedom 'undoing' any previous command.

! Cons: Uses a lot of memory to store previous instances of **ProjectDashboard**.

¥ Alternative 2: Clear redundant 'history' of previous **ProjectDashboard** states after exceeding a chosen quota. (E.g. 5 commands executed)

! Pros: Uses memory efficiently, while giving users some freedom to **undo** multiple commands.

! Cons: User would be unable to **undo** certain 'Older' commands.