

Abhinav Ramnath - Project Portfolio

Introduction

This project portfolio documents my contributions to a software engineering group project taken in my sophomore year in the National University of Singapore. For this project, my team and I worked together to build a desktop application, +Work, targeted at NUS students who are project leaders.

About the Project

+Work is a project management tool with a graphical user interface (GUI) that helps project leaders to manage their projects details. +Work uses a command line interface (CLI) to operate, in other words, users mainly interact with +Work by typing specific commands and pressing the enter key. The main features of +Work include; managing tasks, managing members, managing inventory, generating statistics, generating PDF reports and finding optimal meeting times for members.

Important Symbols

Shown below are a few symbols that will appear in this document, along with their significance.

TIP	Information listed here can help users of +Work save time when using the application.
NOTE	Information listed here should be noted by users to help them improve their experience.
IMPORTANT	Information listed here is essential to users. They need to understand this in order for +Work to work optimally.

Summary of Contributions

This section serves to summarise my major contributions to the project as well other noteworthy contributions such as project management.

Enhancements

¥ Major enhancement one: I added the ability for users to customise the appearance of +Work to suit their viewing preferences.

! What it does: This feature allows the user to toggle the theme of the entire application between light and dark. Also, it allows the user to toggle the display of task deadlines format of the app between 24hr and 12hr.

! Justification: This feature improves the current product as +Work can adapt to users viewing preferences. Furthermore, as +Work scales to v2.0 and beyond, it would be easier for users and developers alike to handle the increasing complexity additional features may

bring.

! Highlights: As of the latest release, this enhancement affects any command that deals with time sensitive data. The implementation was moderately difficult as I was able to adapt the integration of the [UserSettings](#) component based on existing components. The more difficult aspect was to design the component in such a way where it would be easy to extend its functionality in the event more user settings are to be added.

! Credits: I adapted some code that does responsive highlighting for the settings panel in +Work from a course mate.

¥ Major enhancement two: I added a project dashboard to +Work that helps users see important information when the application opens.

! What it does: This feature allows the user to view their tasks based on their current completion status. Furthermore, users can view their upcoming deadlines as well as any upcoming project meetings.

! Justification: This feature improves the current product as students decide which tasks demand their attention so that they can remain on track for their project. Furthermore, they can prepare for upcoming meetings more efficiently.

! Highlights: The dashboard is responsive to changes in task status, deadlines and any project meetings. The implementation was relatively simple as I was also responsible for integrating task management into the project.

¥ Minor enhancement: I added cosmetic improvements to +Work's dashboard so as to make it more appealing to students.

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

Other Contributions

¥ Project management:

! Managed releases [v1.1](#), [v1.2.1](#) and [v1.4](#) (3 releases) on GitHub

! Tracked and helped to fix bugs found during developer testing using GitHub issues (Issue [#82](#))

! Added additional tests to the repository to bump coverage up from 37% to 39% (Pull request [#166](#))

¥ Enhancements to the application:

! Created a modular framework in [Ui](#) to make it easier for teammates to add different user views for their respective features (Pull request [#51](#))

¥ Documentation:

! Did cosmetic tweaks to existing contents of the User Guide: [#152](#)

¥ Community:

! Reviewed pull requests made by teammates to the repository on GitHub (with non-trivial review comments): [#55](#)

! Reported bugs and suggestions for other teams in the module (examples: [1](#), [2](#), [3](#))

¥ Tools:

! Added the test coverage tool Coveralls to the team repository (Pull request [#149](#))

Contributions to the User Guide

The following is an excerpt taken from the User Guide of +Work. I wrote instructions to help users customise their preferences using the settings feature.

Settings Commands

+Work helps you view your current settings by highlighting your current option! To see your current settings navigate to the settings panel by entering `settings` as described in section Section 3.1.3.

Switching the theme of +Work: `theme`

This command helps you toggle the theme of +Work between `light` and `dark` to suit your viewing preferences.

TIP | By default, the theme is set to `dark`.

Format: `theme light`

Example:

¥ Type `theme light` into the command box as shown below.

¥ Hit the `Enter` key and +Work switches to the `light` theme! As seen below, your choice of `light` is highlighted as well.



¥ Similarly, `theme dark` switches to the `dark` theme.

Switching the time format of +Work: `clock`

This command helps you toggle the time format of +Work between 24 hour and 12 hour clock.

TIP | The time format is set to 24 hour by default.

NOTE | This does not affect the input format of deadlines for tasks, you still need to be enter them in the 24 hour format!

Format: `clock twenty_four`

Example:

¥ Type `clock twelve` into the command box as shown below.

¥ Hit the `Enter` key and +Work switches the time format to the 12 hour clock! As seen below, the new format is highlighted as well.



¥ Now when you navigate to any view which has time sensitive data, you will see that the format has been switched. For example, if you were to switch back to the dashboard by entering `home` you should see a window similar the one shown below.

¥ Similarly, `clock twenty_four` switches to the 24 hour clock.

Contributions to the Developer Guide

The following is an excerpt taken from the Developer Guide of +Work, showing additions I have made to explain the technical aspects of the settings and dashboard feature.

Settings feature

Implementation

This feature was implemented to allow users to customise their experience when using +Work.

The commands introduced by this feature include; `theme light`, `theme dark`, `clock twenty_four` and `clock twelve`. The commands are facilitated by `UserSettings`. This component resides in `Model` and contains the customisable settings available to the user, which are currently the `theme` and

clockFormat.

¥ `UserSettings#getTheme()` "Ñ"Retrieves the current theme applied to +Work.

¥ `UserSettings#getClockFormat()` "Ñ"Retrieves the current clock format applied to +Work.

¥ `UserSettings#setTheme(Theme newTheme)` "Ñ"Sets the default theme of +Work to be `newTheme`

¥ `UserSettings#setClockFormat(ClockFormat newClockFormat)` "Ñ"Sets the default clock format of +Work to be `newClockFormat`

These operations are exposed in the `Model` interface as `Model#getCurrentTheme()`, `Model#getCurrentClockFormat()`, `Model#setCurrentTheme(Theme newTheme)`, `Model#setClockFormat(ClockFormat newClockFormat)` respectively.

NOTE	To allow <code>Ui</code> to be responsive to updates in the settings, two of the operations are similarly exposed in the <code>Logic</code> interface as <code>Logic#getTheme()</code> and <code>Logic#getClockFormat()</code> .
------	--

The activity diagram below summarises the process of executing a settings command.

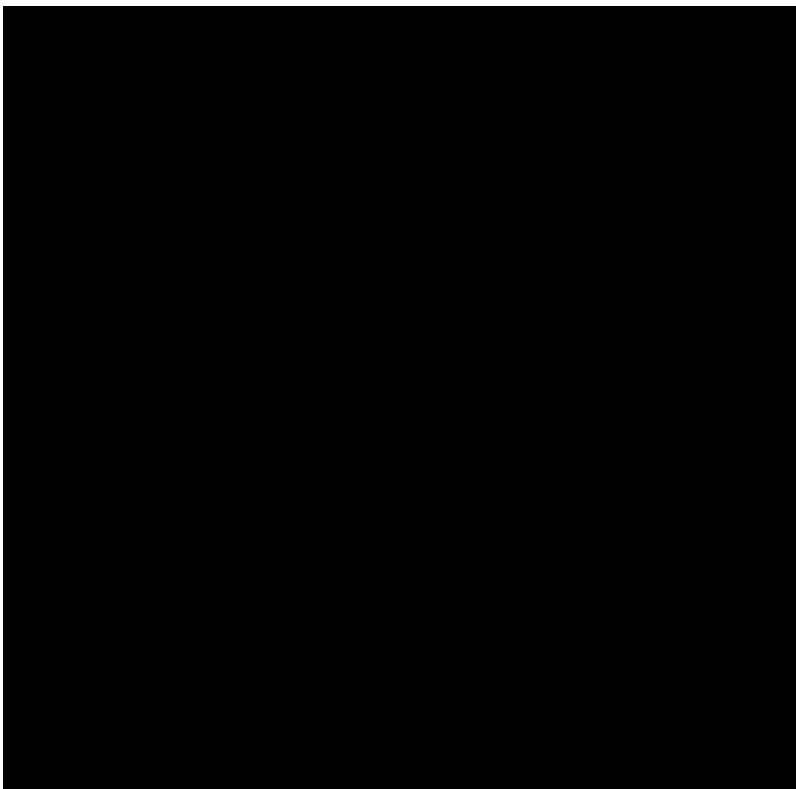


Figure 1. Activity diagram of settings command execution.

Assume that the current `theme` is `LIGHT` and `clockFormat` is `TWENTY_FOUR`.

Given below is an example usage scenario and how the various commands work:

Step 1. The user launches the application. The `UserSettings` will be initialised by `Model` based on the saved `UserSettings`.

Step 2. The user executes `theme dark` command.

Step 3. `Logic#execute()` calls `Model#setDarkTheme()`, which calls `UserSettings#setDarkTheme()`. This changes the `theme` attribute in `UserSettings` to `DARK`.

Step 4. `DARK` theme has been applied to `+Work` and `Ui` is updated.

Step 5. The settings have been updated and stored in `plusworksettings.json`.

The following sequence diagram shows how the `theme dark` operation works with reference to steps 2 and 3 above.

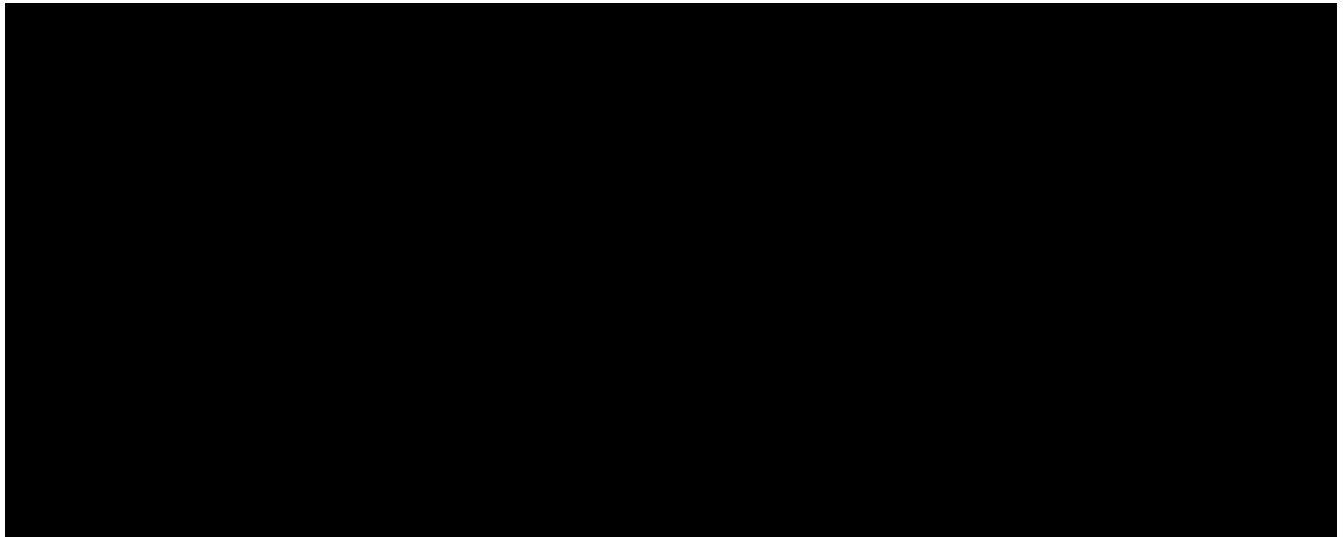


Figure 2. Operational flow of `ThemeCommand`

NOTE	The lifeline for <code>ThemeCommand</code> should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.
------	---

The `theme light` operation is similar to the one shown in figure above. However, the method called is `UserSettings#setLightTheme()`.

IMPORTANT	The <code>clock twelve</code> and <code>clock twenty_four</code> have a similar operation to <code>theme dark</code> as well. There are two differences, <code>ClockCommand</code> replaces <code>ThemeCommand</code> and the associated methods called in <code>Model</code> are different.
-----------	--

Design Considerations

This section explores how the design can affect the level of customisation available to the user through the settings feature in `+Work`.

Aspect: Storage of the various options in settings data

Within a specific setting stored in `Model`, each option has data that helps yield a specific behaviour. Currently the available settings are represented as `Enum`.

¥ Alternative 1 (current choice): The relevant data is stored within the class itself.

For example `ClockFormat` has two constants `TWENTY_FOUR` and `TWELVE` that contain `DateTimeFormatters` which are retrieved when the user wishes to toggle between them.

! Pros: Better design as it is more modular. The data can be stored as attributes of the enum

constants and retrieved via the default setting from `Model`. Furthermore if data is to be changed, it only needs to be changed in one component for the expected behaviour to be achieved.

! Cons: User cannot customise the data directly due to the nature of `Enum` classes.

¥ Alternative 2: The data is stored in the `UserSettings` component as `static` fields.

! Pros: This exposes the data of each option for each settings to the `Model` component. If the user requests to customise that data, it would be possible in this design.

! Cons: `UserSettings` would change whenever the data related to a particular settings option is updated. Ideally, `UserSettings` should only be aware of the various settings the user is able to customise.

We decided to opt for design option one, so as to be in line with the Single Responsibility principle. This would make it easier for future developers to extend the functionality of `UserSettings` in a more modular manner.

Dashboard feature

Implementation

This feature was implemented to allow users to view the status of the tasks in their project, upcoming deadlines and upcoming meetings at a glance.

The command introduced by this feature is `home` and displays data affected by `Task` and `Meeting` commands such as `add-task`, `edit-task` and `add-meeting`. The commands are facilitated by `ProjectDashboard`. This component resides in `Model` and contains the in-memory data of the application which is retrieved when the user switches to `Home`.

¥ `ProjectDashboard#getTasksNotStarted()` ~Retrieves the current list of tasks with status `unbegun` in `+Work`.

¥ `ProjectDashboard#getTasksDoing()` ~Retrieves the current list of tasks with status `doing` in `+Work`.

¥ `ProjectDashboard#getTasksDone()` ~Retrieves the current list of tasks with status `done` in `+Work`.

¥ `ProjectDashboard#getTasksByDeadline()` ~Retrieves the current list of tasks with nearing deadlines in `+Work`.

¥ `ProjectDashboard#getMeetingList()` ~Retrieves the current list of meetings in `+Work`.

¥ `ProjectDashboard#splitTasksByStatus()` ~Processes the current list of tasks and stores the tasks by status.

¥ `ProjectDashboard#splitTasksByDeadline()` ~Processes the current list of tasks and stores the tasks based on nearing deadlines.

These operations are exposed in the `Model` interface as `Model#getFilteredTasksNotStarted()`, `Model#getFilteredTasksDoing()`, `Model#getFilteredTasksDone()`, `Model#getFilteredTasksByDeadline()` and `Model#getFilteredMeetingList()`.

NOTE

To allow **Ui** to be responsive to updates in the settings, all of the operations are similarly exposed in the **Logic** interface `Logic#getFilteredTasksNotStarted()`, `Logic#getFilteredTasksDoing()`, `Logic#getFilteredTasksDone()`, `Logic#getFilteredTasksByDeadline()` and `Logic#getFilteredMeetingList()`.

Step 1. The user executes the **home** command.

Step 2. **Logic** executes `Logic#getFilteredTasksNotStarted()`, `Logic#getFilteredTasksDoing()`, `Logic#getFilteredTasksDone()`, `Logic#getFilteredTasksByDeadline()` and `Logic#getFilteredMeetingList()`.

Step 3. This calls `Model#getFilteredTasksNotStarted()`, `Model#getFilteredTasksDoing()`, `Model#getFilteredTasksDone()`, `Model#getFilteredTasksByDeadline()`.

Step 4. This executes `ProjectDashboard#splitTasksByStatus()`, to populate `tasksNotStarted`, `tasksDoing` and `tasksDone`.

Similarly, `ProjectDashboard#splitTasksByDeadline()` is called to populate `tasksByDeadline`.

Step 5. The various **FilteredList** objects are updated, since their backing lists are stored in **ProjectDashboard**. (refer to Figure 9)

The object diagram below shows a snapshot of the various objects involved when the user views the dashboard.

NOTE

The diagram omits objects involving the **Ui** component as well as specific **Task** objects for brevity.

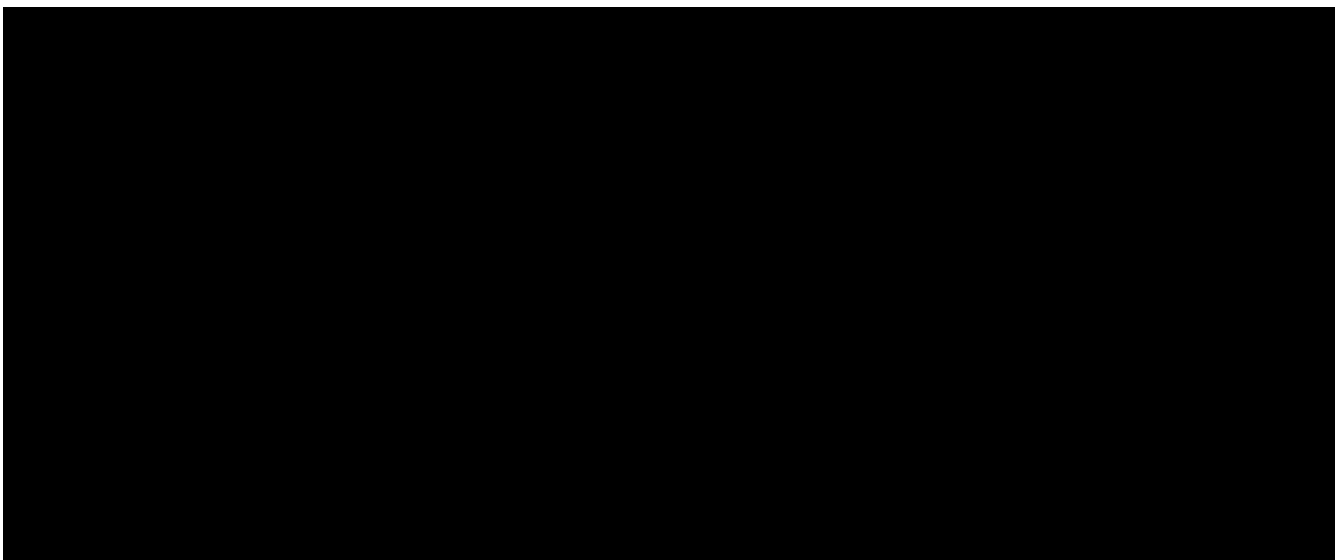


Figure 3. Snapshot of objects involved in populating the dashboard with data

Design Considerations

This section explores how the design of the dashboard can affect its responsiveness and integration with other data in the application.

Aspect: Data structure used to store **Task** objects.

¥ Alternative 1 (current choice): All tasks are stored in a single **UniqueTaskList** in **ProjectDashboard** with **TaskStatus** attribute. When the user enters **home** to view the dashboard, the tasks are split by **TaskStatus** and deadline in **ProjectDashboard** and dispatched to the **Ui**.

! Pros: Easier to implement in terms of storage and retrieval. By storing only one list and splitting the tasks in memory there is less data saved.

! Cons: The constant processing of task data may tax the memory of the application, as it is storing the same tasks in multiple data structures. This may affect performance for large number of tasks.

¥ Alternative 2: The **Task** objects will only contain attributes which are not filtered in the dashboard. They can be stored in a **HashMap** as values and the keys are filtered attributes such as **TaskStatus** and deadline.

! Pros: Memory usage of **_Work** is more efficient, as **ProjectDashboard** does not have to store multiple references of the same **Task** objects in memory. Also, due to the mappings between **TaskStatus** and the **Task** assigned those statuses, they can be retrieved and displayed more efficiently.

! Cons: Due to the requirements of **+Work**, **Task** objects are coupled to **Member** and **Inventory**. The method of storing these tasks, other components would have to iterate through all keys to obtain all the **Task** objects and manipulate their mappings. This would render the **HashMap** useless.

We decided to opt for design option one so as to enable **Task** to integrate with other components of **+Work** in the most efficient way possible. Although design option two would benefit the dashboard greatly it would cause almost all other components and views to become inefficient.