+Work - Developer Guide

By: `T12-1` Since: `Sep 2019` Licence: `MIT` == Setting up

Refer to the guide here.

1. Design

1.1. Architecture

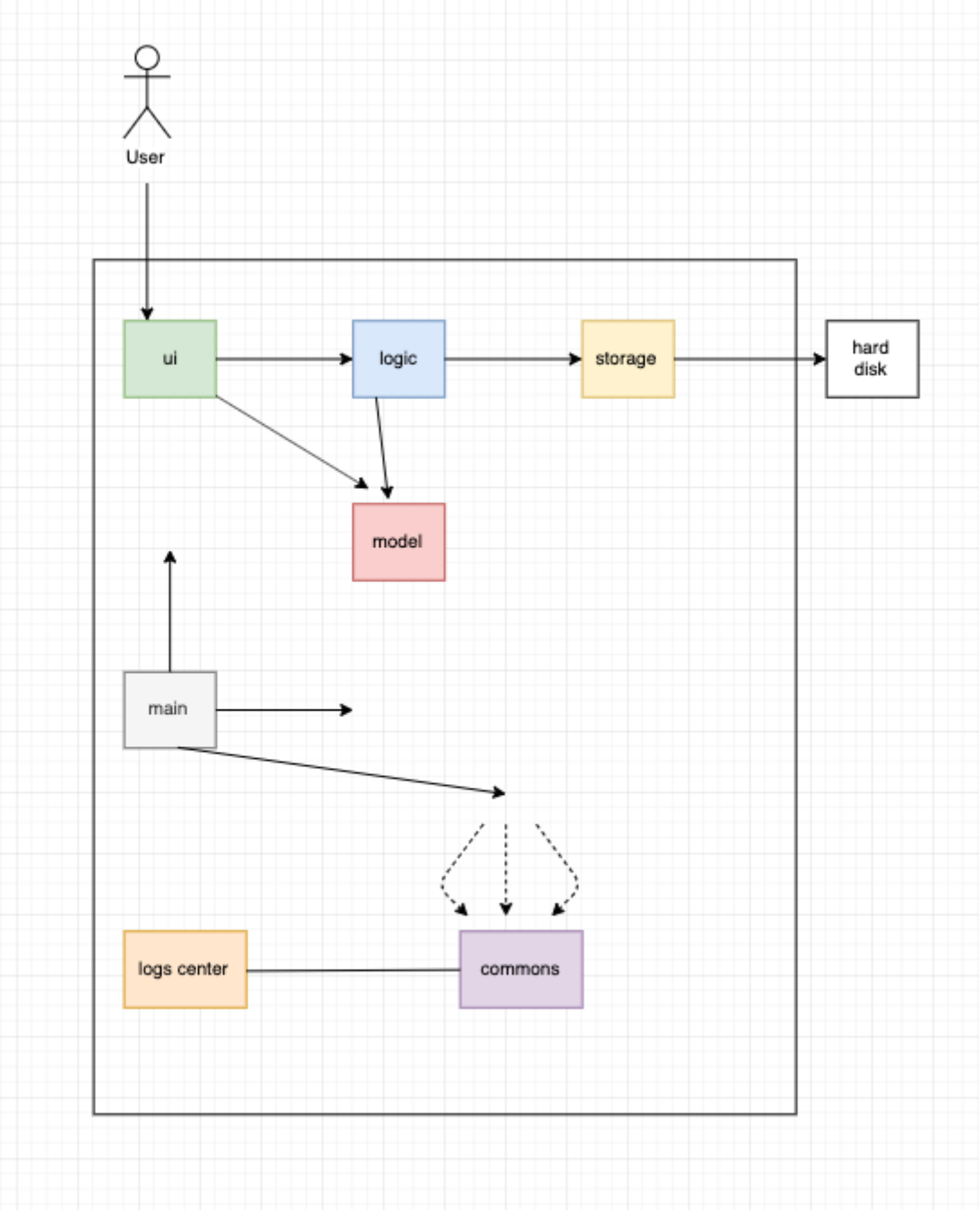


Figure 1. Architecture Diagram

The ***Architecture Diagram*** given above explains the high-level design of the App. Given below is a quick overview of each component.

TIP	The `.puml` files used to create diagrams in this document can be found in the diagrams folder. Refer to the Using PlantUML guide to learn how to create and edit diagrams.

`Main` has two classes called `Main` and `MainApp`. It is responsible for,

- At app launch: Initializes the components in the correct sequence, and connects them up with each other.
- At shut down: Shuts down the components and invokes cleanup method where necessary.

`Commons` represents a collection of classes used by multiple other components. The following class plays an important role at the architecture level:

- `LogsCenter` : Used by many classes to write log messages to the App's log file.

The rest of the App consists of four components.

- **`UI`**: The UI of the App.
- **`Logic`**: The command executor, parses user input.
- **`Model`**: Holds the data of the App in-memory.
- **`Storage`**: Reads data from, and writes data to, the hard disk.

Each of the four components

- Defines its *API* in an `interface` with the same name as the Component.
- Exposes its functionality using a `{Component Name}Manager` class.

For example, the `Logic` component (see the class diagram given below) defines it's API in the `Logic.java` interface and exposes its functionality using the `LogicManager.java` class.

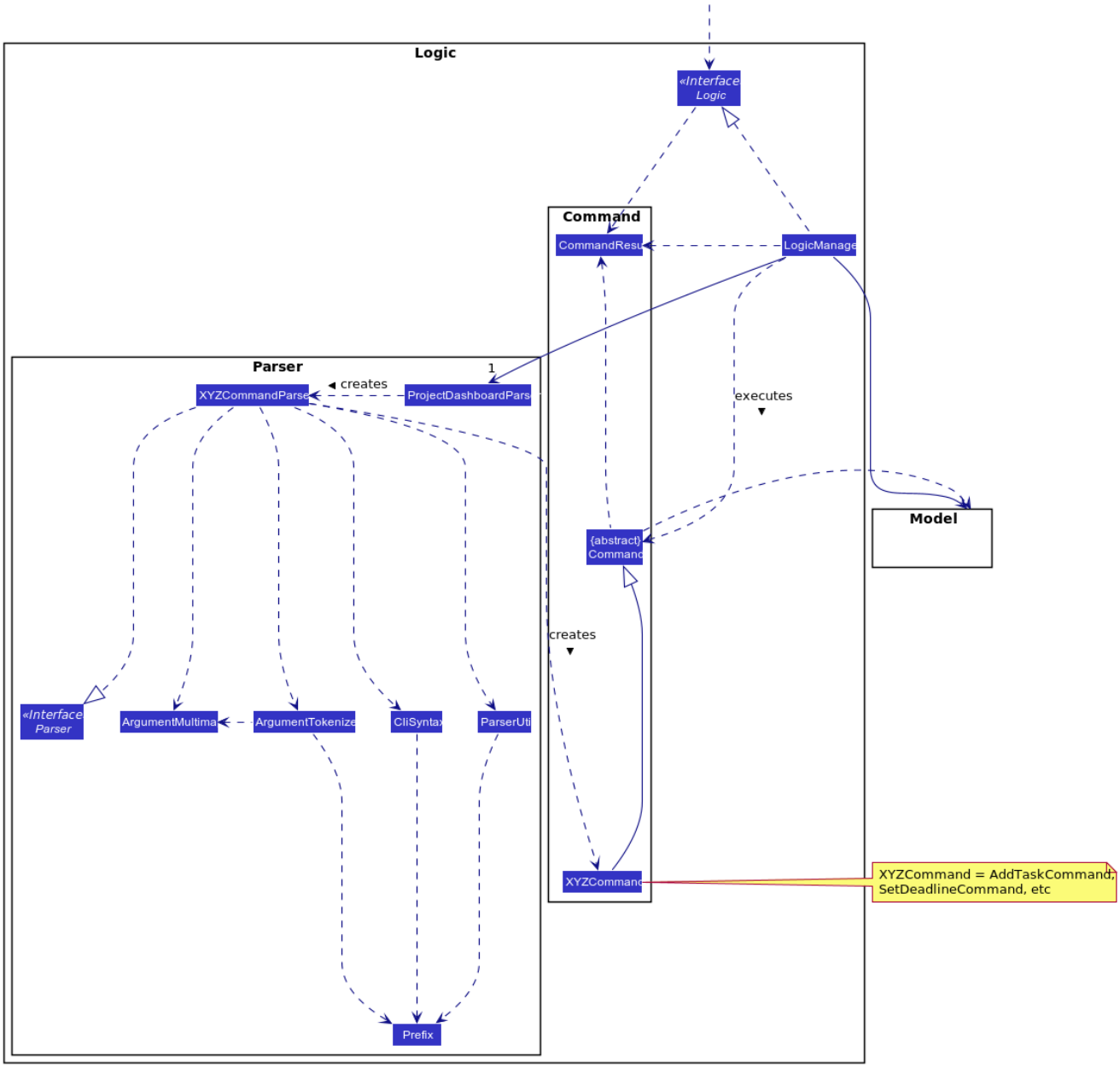


Figure 2. Class Diagram of the Logic Component

How the architecture components interact with each other

The *Sequence Diagram* below shows how the components interact with each other for the scenario where the user issues the command `add-member mn/Abhi`.

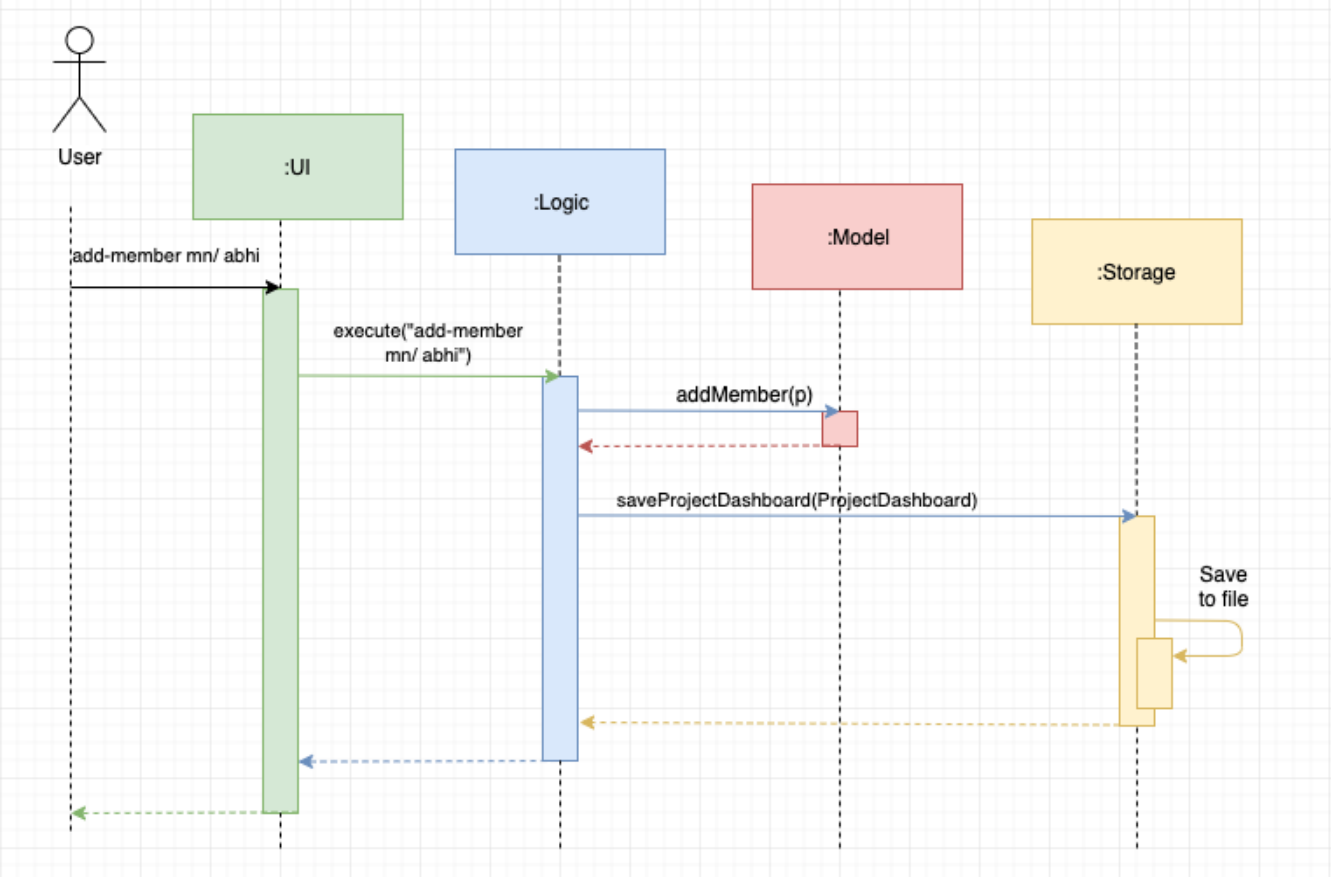


Figure 3. Component interactions for `add-member mn/Abhi` command

The sections below give more details of each component.

1.2. UI component

1.2.1. Overview

The `UI` consists of the `MainWindow` that is made up of static and non-static parts e.g.`CommandBox`, `ResultDisplay`, `StatusBarFooter` etc. All these, including the `MainWindow`, inherit from the abstract `UiPart` class.

NOTE	The non-static `Ui` component, `UserViewMain` is responsive to user commands. Its functionality will be further explained using a separate diagram.

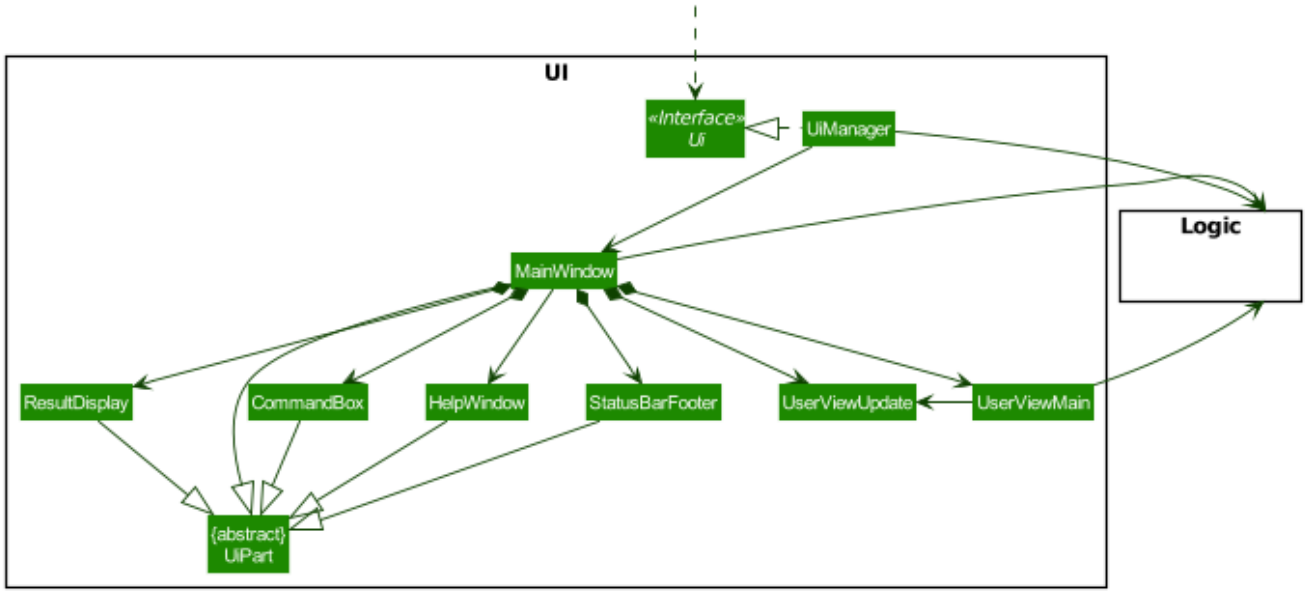


Figure 4. Structure of the UI Component

API : `Ui.java`

1.2.2. Functionality of `UserViewMain`

`UserViewMain` is the component responsible for switching the user view. As it fetches data from `Logic`, it is associated with `Logic`, hence Figure 5 below shows an association between `UserViewMain` and `Logic`. The controller class of `UserViewMain` is `UserViewController`.
`UserViewController` and `UserViewNavigator` both contain references to the various `UiPart` views offered in +Work in order to switch between them successfully.

The diagram below shows how `UserViewMain` integrates with `Logic`, `Model` and `UiPart`.

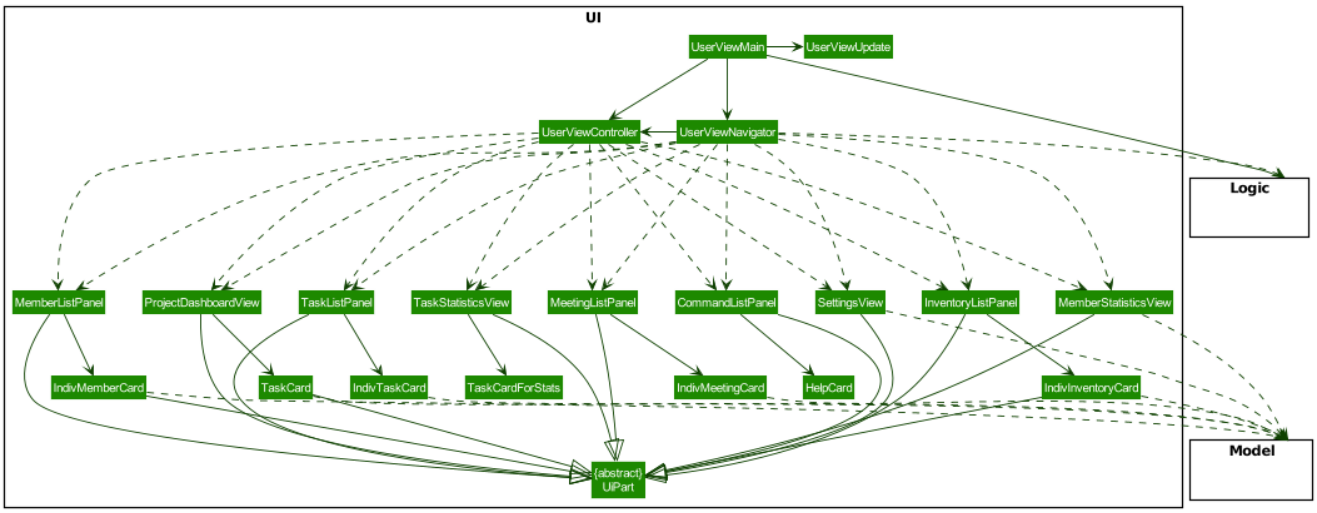


Figure 5. UserViewMain integration with UI component

Below is a guide to how `UserViewMain` switches the users view.

Step 1: User enters a command.

Step 2: `UserViewUpdate` parses said command and interacts with `UserViewMain` to display the

requested layout.

Step 3: `UserViewMain` interacts with `UserViewNavigator`, which obtains the relevant data from `Logic`, to create the relevant `UiPart`. These `UiPart` components are the `MemberListPanel`, `ProjectDashBoardView` etc.

NOTE	The non-static `UiPart` components are stored in `views` folder.

Step 4: This component is then passed to `UserViewController` to set the current view of `UserViewMain` component.

Step 5: Users view is then switched successfully.

The `UI` component uses the JavaFx UI framework. The layout of these UI parts are defined in matching `.fxml` files that are in the `src/main/resources/view` folder. For example, the layout of the `MainWindow` is specified in `MainWindow.fxml`

The `UI` component,

- Parses and executes user commands to show user the right view using the `Logic` component.
- Listens for changes to `Model` data so that the UI can be updated with the modified data.

1.3. Logic component

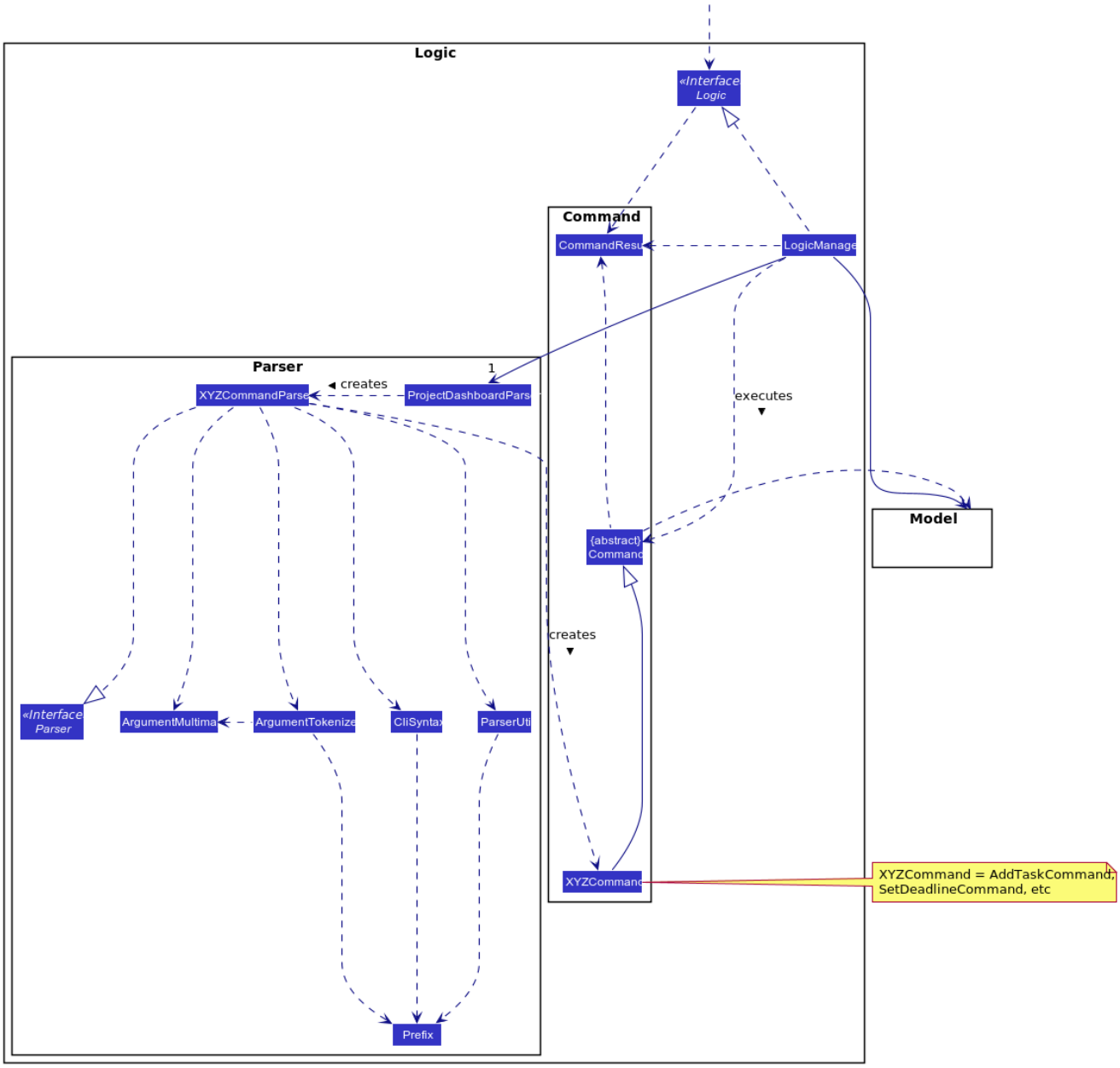


Figure 6. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `ProjectDashboardParser` class to parse the user command.
2. This results in a `Command` object which is executed by the `LogicManager`.
3. The command execution can affect the `Model` (e.g. adding a task).
4. The result of the command execution is encapsulated as a `CommandResult` object which is passed back to the `Ui`.
5. In addition, the `CommandResult` object can also instruct the `Ui` to perform certain actions, such as displaying help to the user.

Given below is the Sequence Diagram for interactions within the `Logic` component for the `execute("delete 1")` API call.

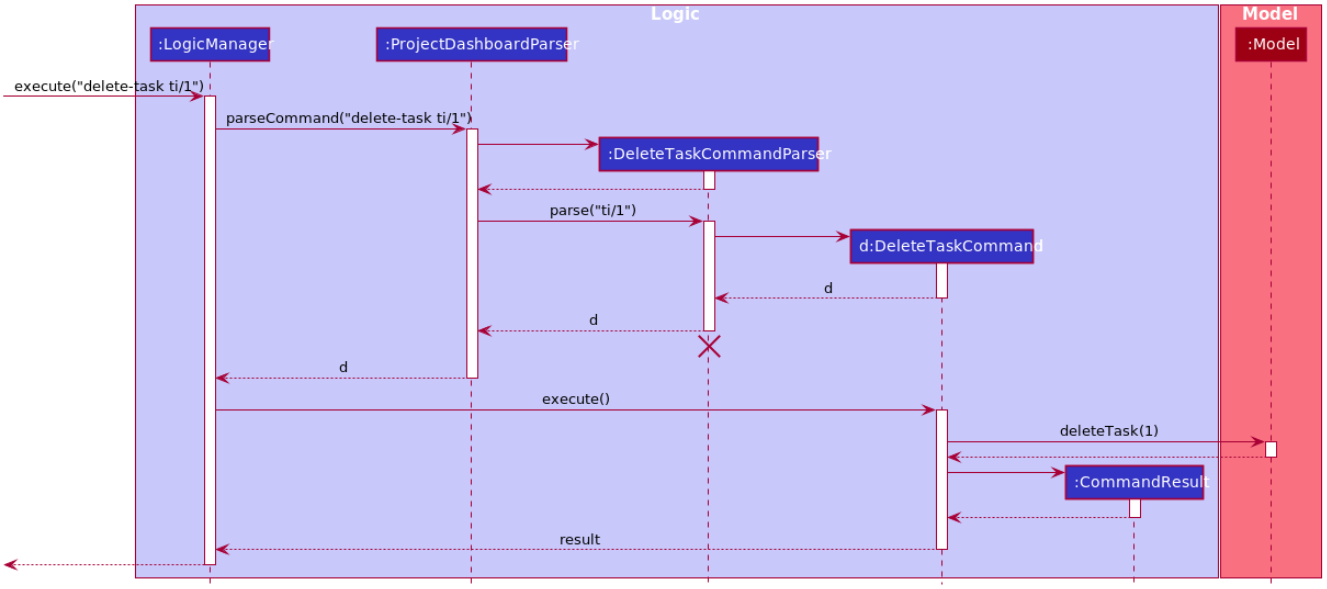


Figure 7. Interactions Inside the Logic Component for the `delete 1` Command

NOTE	The lifeline for `DeleteTaskCommandParser` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

1.4. Model component

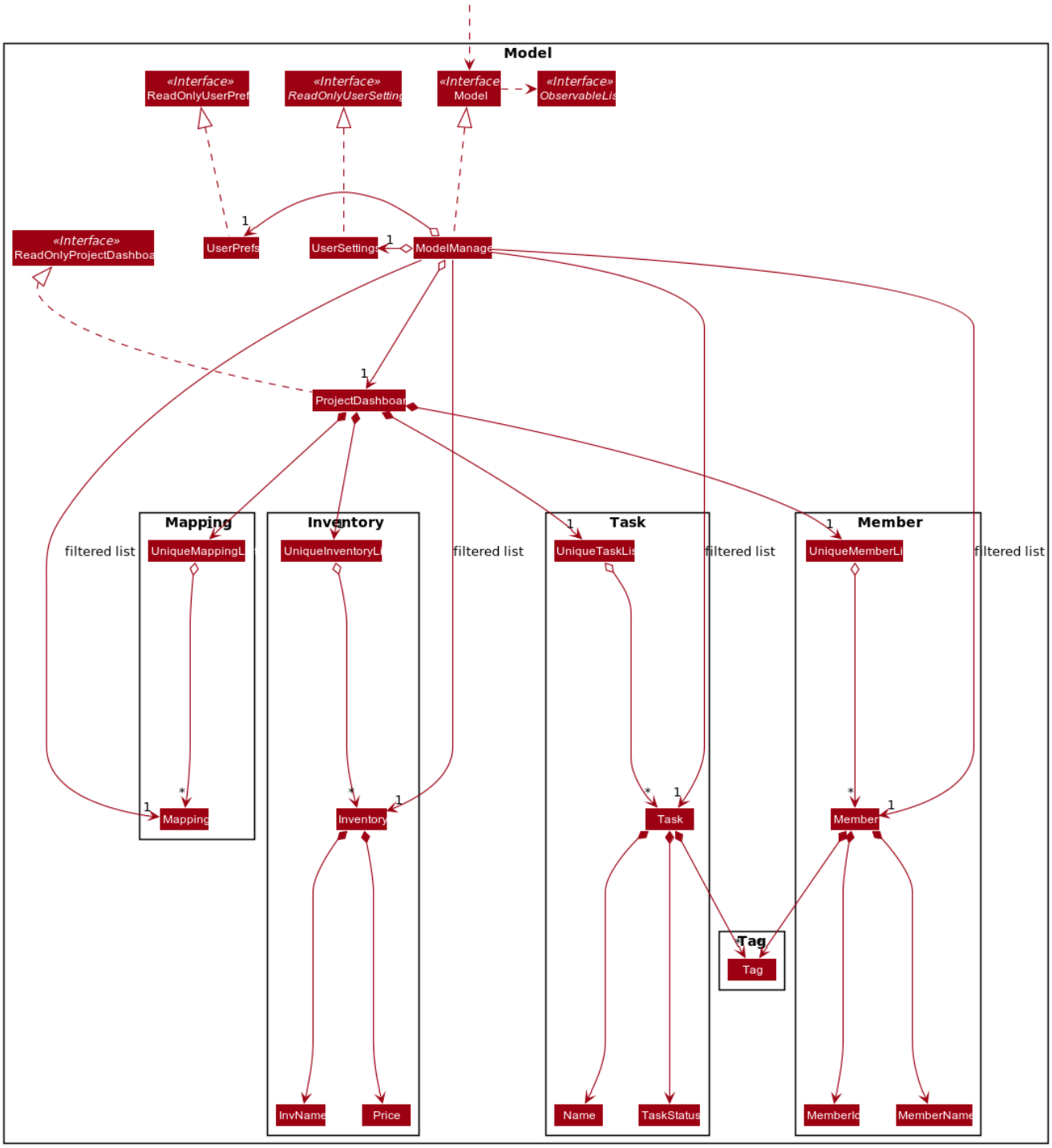


Figure 8. Structure of the Model Component

API : `Model.java`

The `Model`,

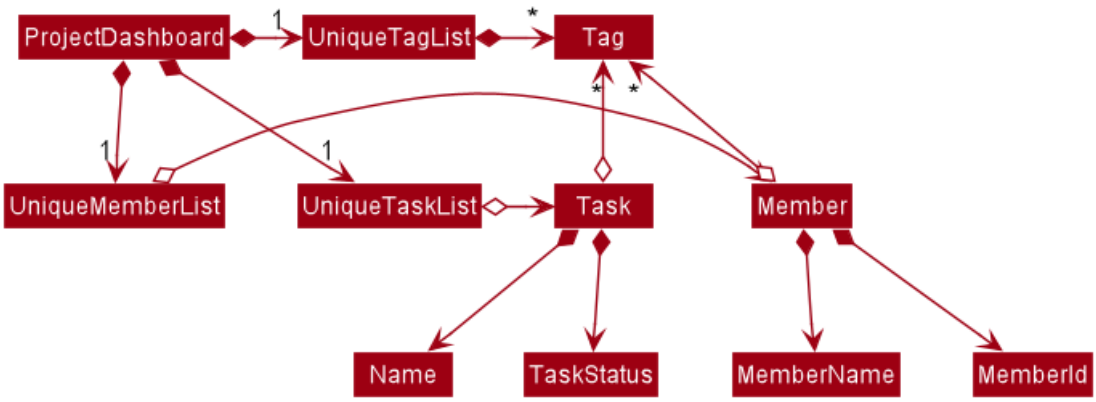
- stores a `UserPref` object that represents the user's preferences.
- stores the Project data.
- stores the `UserSettings` that represents the user's preferred settings of +Work.
- exposes the unmodifiable `ObservableList<Member>`, `ObservableList<Task>`, `ObservableList<Inventory>`, `ObservableList<Meetings>` and `ObservableList<Mapping>` that can be 'observed' e.g. the UI can be bound to this list so that the UI automatically updates when the

data in the list change.

- does not depend on any of the other three components.

NOTE

As a more OOP model, we can store a `Tag` list in `Project Dashboard`, which `Member` or `Task` can reference. This would allow `Project Dashboard` to only require one `Tag` object per unique `Tag`, instead of each `Member` or 'Task' needing their own `Tag` object. An example of how such a model may look like is given below.



1.5. Storage component

The Storage component serves the following purposes.

- It can save `UserPref` object in json format and read it back.
- It can save `UserSettings` object in json format and read it back.
- It can save the +Work data in json format and read it back.

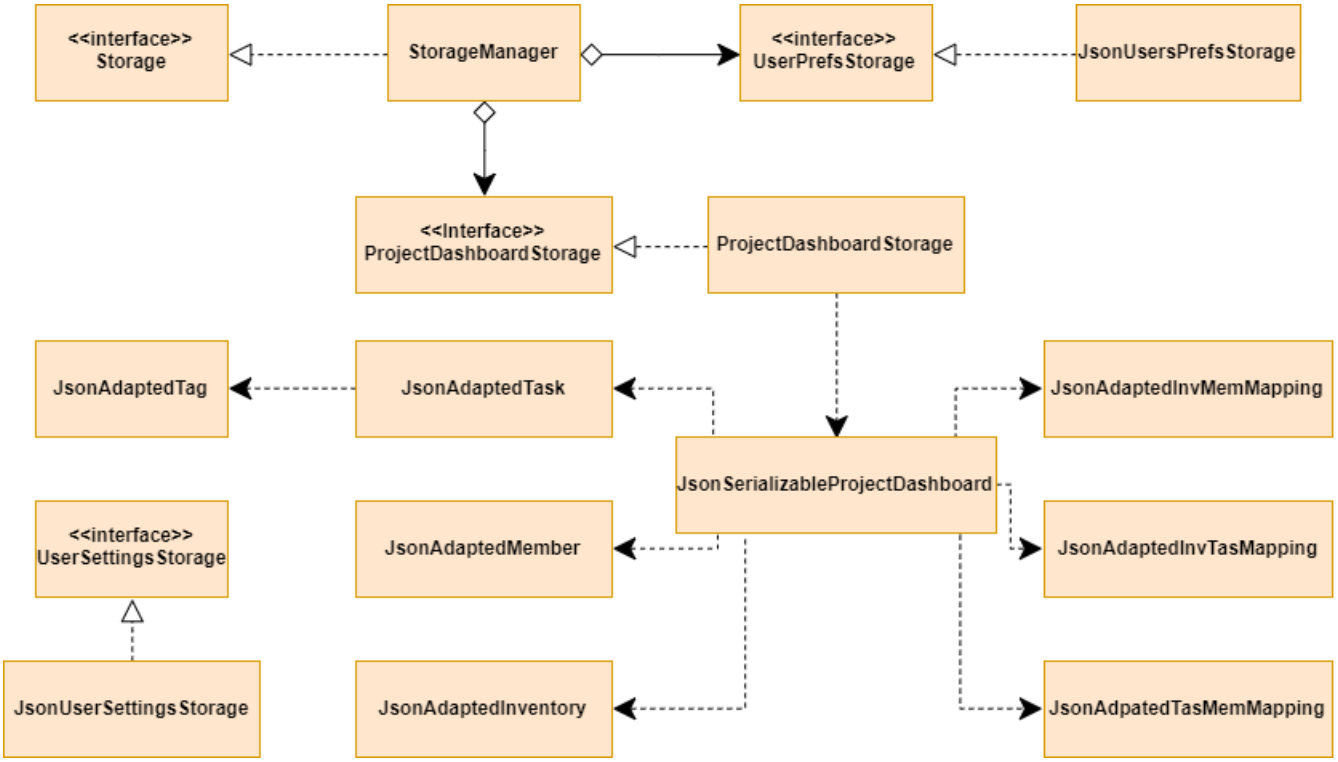


Figure 9. Structure of the Storage Component

API : `Storage.java`

1.6. Common classes

Classes used by multiple components are in the `seedu.pluswork.commons` package.

2. Implementation

This section describes some noteworthy details on how certain features are implemented.

2.1. Calendar feature

2.1.1. 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 commands introduced by this feature include `generate-timings`, `ics` import and commands. The commands are facilitated by `ProjectCalendar`. The various `ics` files of the members are parsed in this component.

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

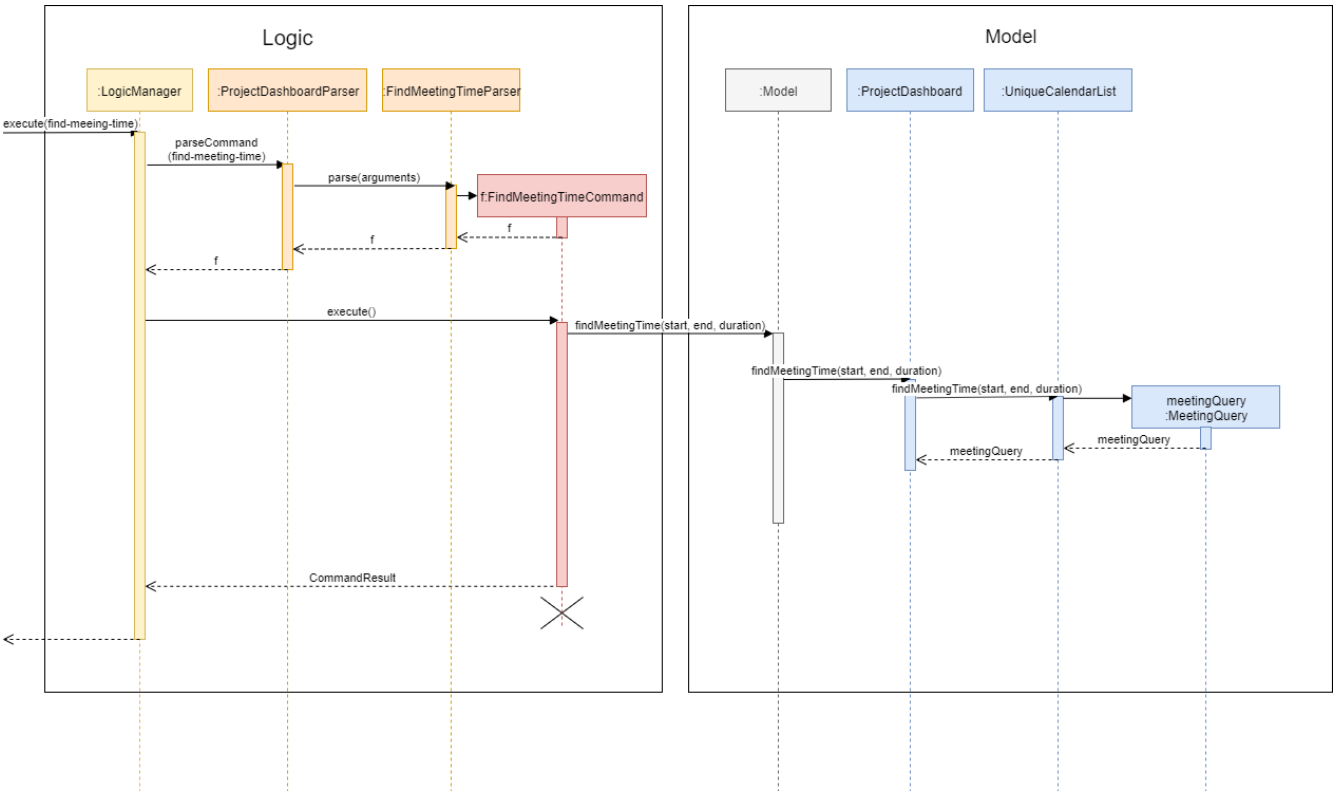


Figure 10. FindMeetingTimeCommand Sequence Diagram

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.

2.1.2. Design Considerations

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)

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

2.2. Dashboard feature

2.2.1. 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 `ProjectDasboard`. 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#splitTaksByStatus()` — Processes the current list of tasks and stores the tasks by status.
- `ProjectDashboard#splitTaksByDeadline()` — 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.

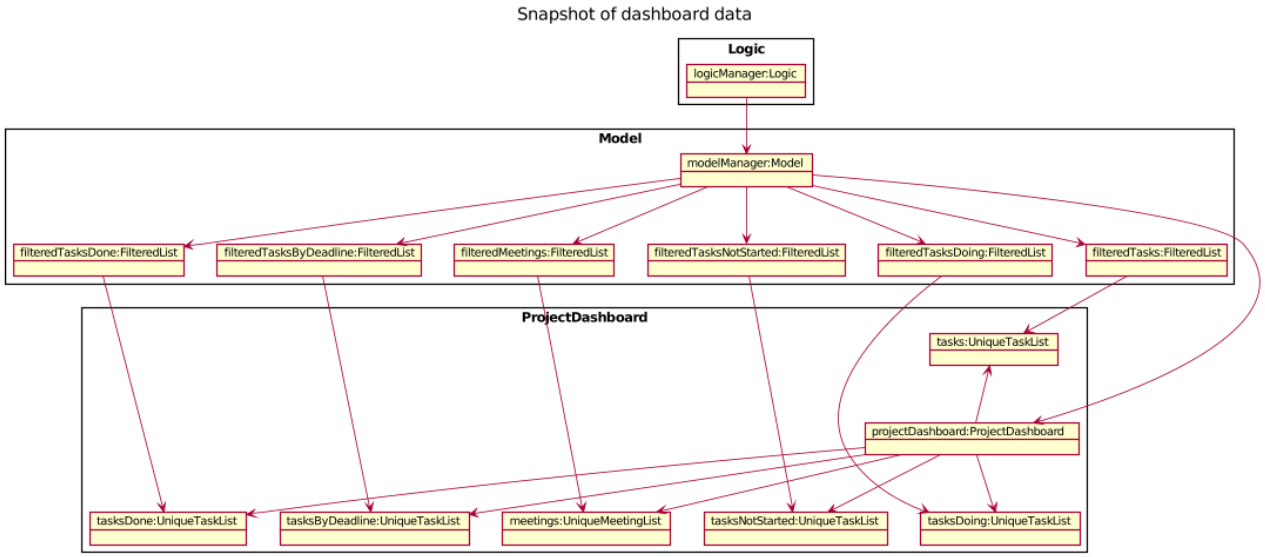


Figure 11. Snapshot of objects involved in populating the dashboard with data

2.2.2. Design Considerations

This section explores how the design of the dashboard can affect it's 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.

2.3. Settings feature

2.3.1. 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()` — Retrieves the current theme applied to +Work.
- `UserSettings#getClockFormat()` — Retrieves the current clock format applied to +Work.
- `UserSettings#setTheme(Theme newTheme)` — Sets the default theme of +Work to be `newTheme`
- `UserSettings#setClockFormat(ClockFormat newClockFormat)` — 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 `Ui` to be responsive to updates in the settings, two of the operations are similarly exposed in the `Logic` interface as `Logic#getTheme()` and `Logic#getClockFormat()`.

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

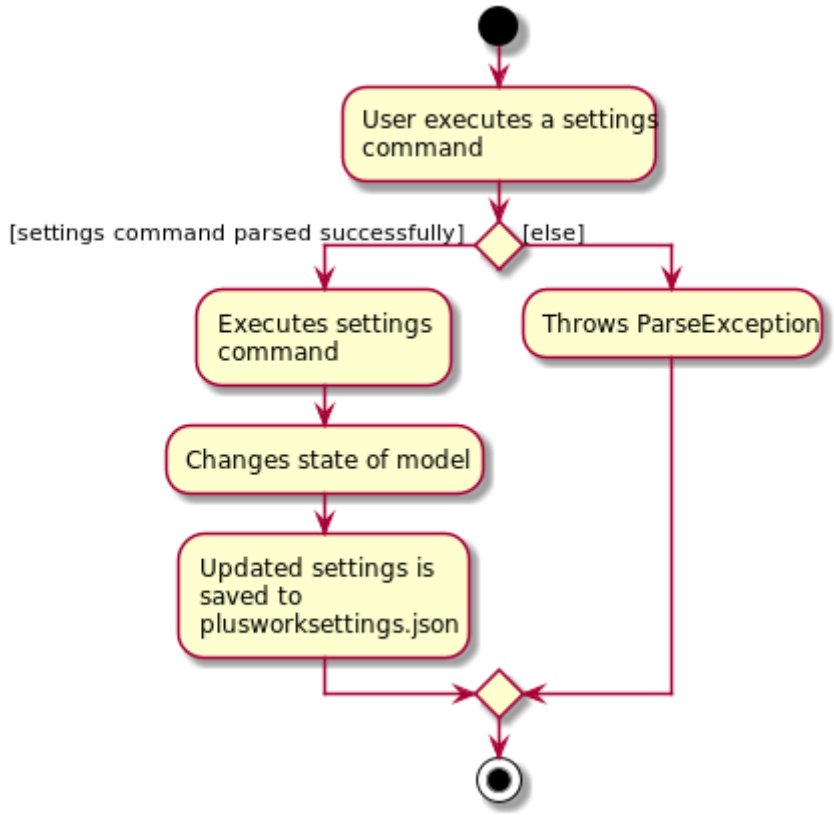


Figure 12. Activity diagram of settings command execution.

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.

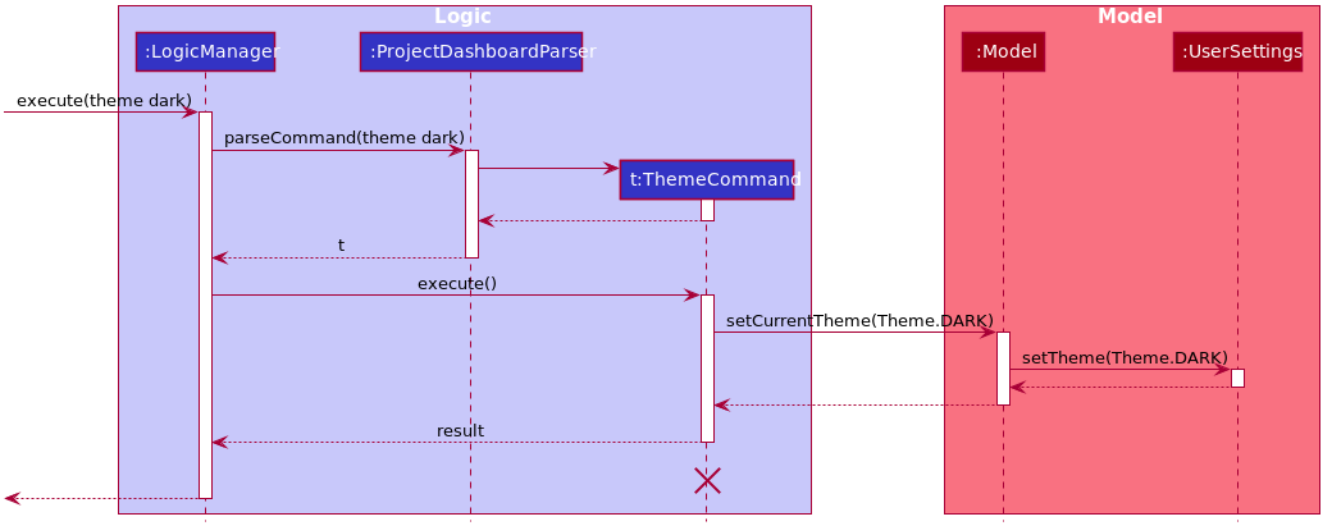


Figure 13. Operational flow of `ThemeCommand`

NOTE	The lifeline for `ThemeCommand` 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 `clock twelve` and `clock twenty_four` have a similar operation to `theme dark` as well. There are two differences, `ClockCommand` replaces `ThemeCommand` and the associated methods called in `Model` are different.

2.3.2. 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 fore 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. Ideadlly, `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.

2.4. Statistics feature

The Statistics feature allows users to retrieve statistics relating to members and tasks in +Work, so that users can get a broad overview of the project's and member's project.

2.4.1. Implementation

The commands introduced by this statistics feature includes: `task-stats` and `member-stats`. These commands are facilitated by the class 'Statistics' that resides within model. The `Statistics` class implements the following operations:

- `Statistics#doCalculations()` — Calculates the statistics needed using existing list of tasks, members and mappings.
- `Statistics#getPortionMembersByTasks()` — Retrieves statistics of all the members and number of tasks completed by the each individual member.
- `Statistics#getPortionMembersByItems()` — Retrieves statistics of all the members and number of items purchased by the each individual member.
- `Statistics#getPortionTasksByStatus()` — Retrieves statistics of all existing tasks and number of tasks of each status.

These operations are exposed in the `Model` interface as `Model#doCalculations`, and `Model#getStatistics`.

NOTE	To allow the `Ui` to be responsive, one of the operations is similarly exposed in the `Logic` interface as `Logic#getStatistics()`.

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

Step 1. The user launches the application for the first time. The `Statistics` object stored by ProjectDashboard is initialised based on the data previously saved.

NOTE	Data previously saved refers to the statistics calculation done based on list of members, tasks and mappings saved.

Step 2. The user executes the `task-stats` command to retrieve statistics related to the tasks in the application.

The `task-stats` command calls `Model#getFilteredTasksList()`, `Model#getFilteredMembersList()` and `Model#getFilteredMappingsList()` to obtain lists of all the members, tasks and mappings saved in the

application. Using the lists, a Statistics object is formed. `Model#setStatistics` is called to updated the statistics in ProjectDashboard.

The following sequence diagram (Figure 10) shows how the `task-stats` operation works.

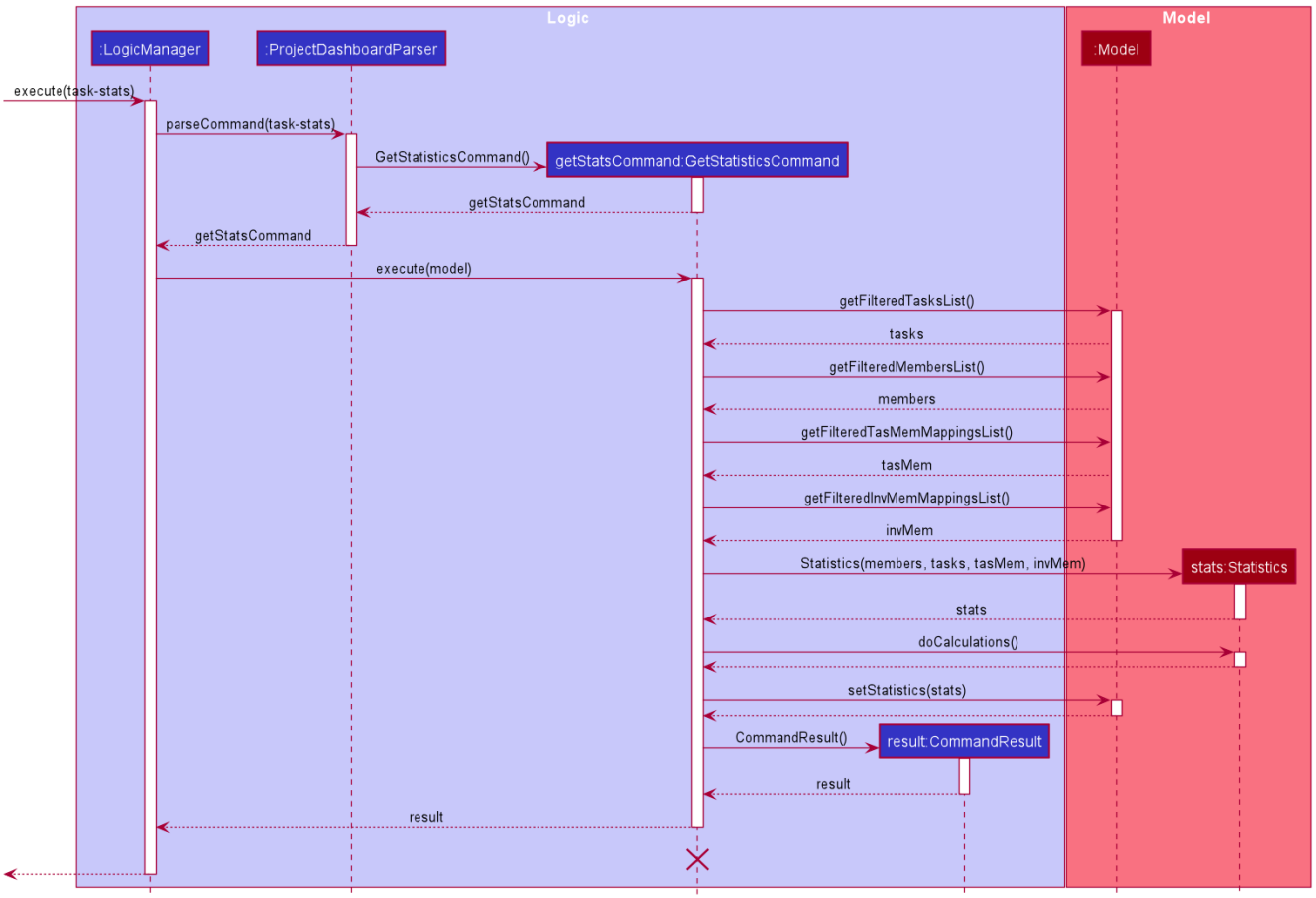


Figure 10. Operational flow of `GetStatisticsCommand`

NOTE	The `member-stats` operation is similar to the one shown in figure 10.

Step 3. In order for task statistics to be displayed in a comprehensive manner, when the `task-stats` command is called, `TaskStatisticsView` class is also called to display the task stats.

NOTE	To allow the `UI` to be responsive, `getStatistics()` is similarly exposed in the `Logic` interface as `Logic#getStatistics()`

The following sequence diagram (Figure 11) shows how calling the `task-stats` operation leads to the comprehensive UI display of task statistics.

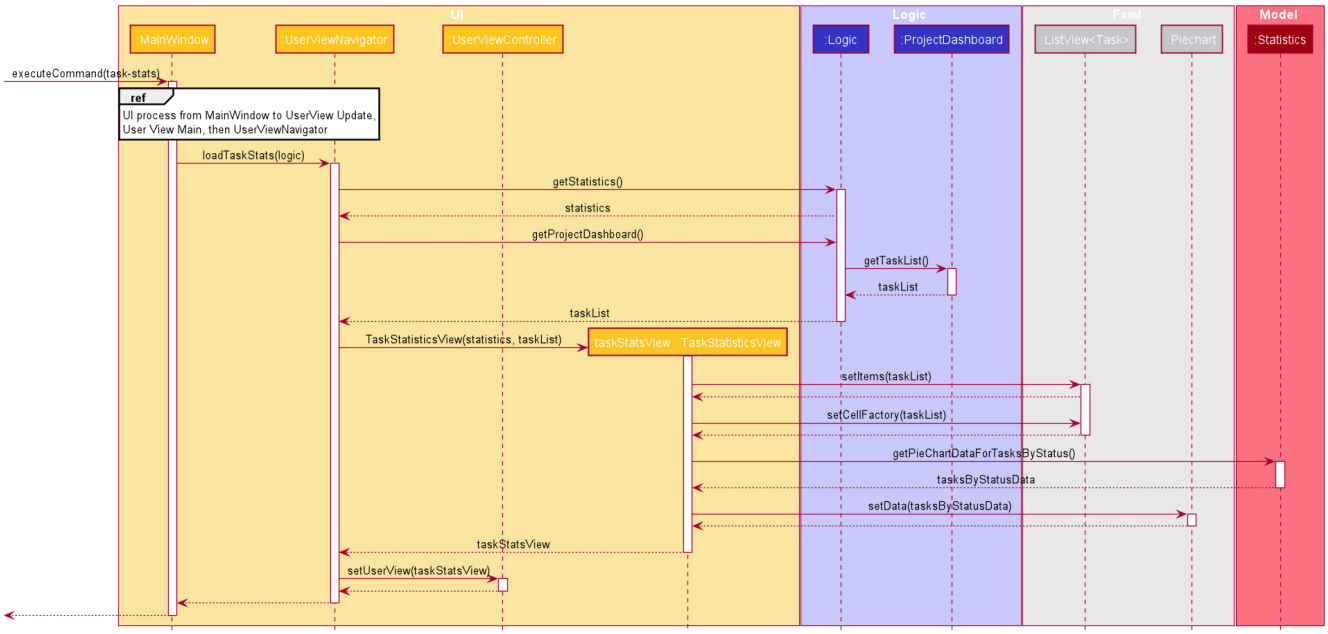


Figure 11. Operational flow of displaying statistics in +Work

The following activity diagram summarizes what happens when a user executes the `task-stats` command:

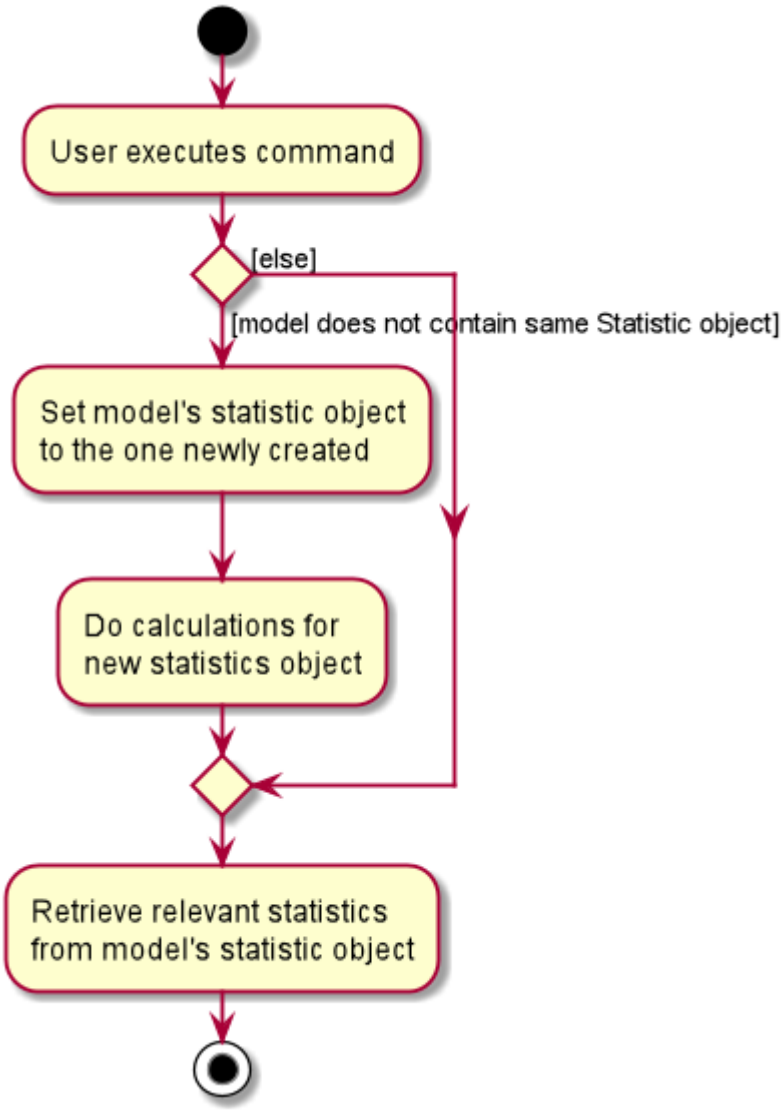


Figure 12. Operational flow during execution of `task-stats` command

2.4.2. Design Considerations

This section describes the pros and cons of the current and other alternative implementations of the Statistics class in +Work, as well as the display of statistics in +Work.

Aspect: Implementation of Statistics class

- **Alternative 1 (current choice):** One statistics object for the entire ProjectDashboard
  - Pros: Easy to implement, centralised class for all statistics
  - Cons: May have performance issues due to calculations involving large amounts of tasks and members.
- **Alternative 2:** Individual statistic objects for members and tasks.
  - Pros: Ensures faster performance, more detailed statistics can be included
  - Cons: Complicates the implementation of the statistics class, might not have enough time to implement it by v1.4

Alternative 1 was chosen given the time constraint in implementing the features in time for +Work Version 1.4.

Aspect: Display of Statistics for Project Dashboard

This section describes the pros and cons of the current and other alternative implementations of displaying the calculate statistics in +Work.

- **Alternative 1 (current choice):** Use a pie chart to represent information
  - Pros: Increases the ease of workload comparison
  - Cons: Decreases the amount of detail of individual tasks and members that are displayed
- **Alternative 2:** Use a list to represent information
  - Pros: Includes more details for individual elements
  - Cons: Decreases the ease of comparison between tasks and members

Because the team came to a consensus that the main objective of the Statistics feature in +Work is to provide the user with an overview of all the project tasks and members, for ease of comparison, **Alternative 1** was chosen as it fits the purpose more than Alternative 2 does.

2.5. Member Feature

The member feature introduces the ability for +Work to deal with project members, in the same way it deals with project tasks. This makes +Work a more comprehensive application because project tasks and members can be kept track of together.

2.5.1. Implementation

+Work's members and their related commands are supported by a `Member` class that resides within model. The following class diagram exposes the structure of the Member class, and shows how the different components relating to the Member class works together.

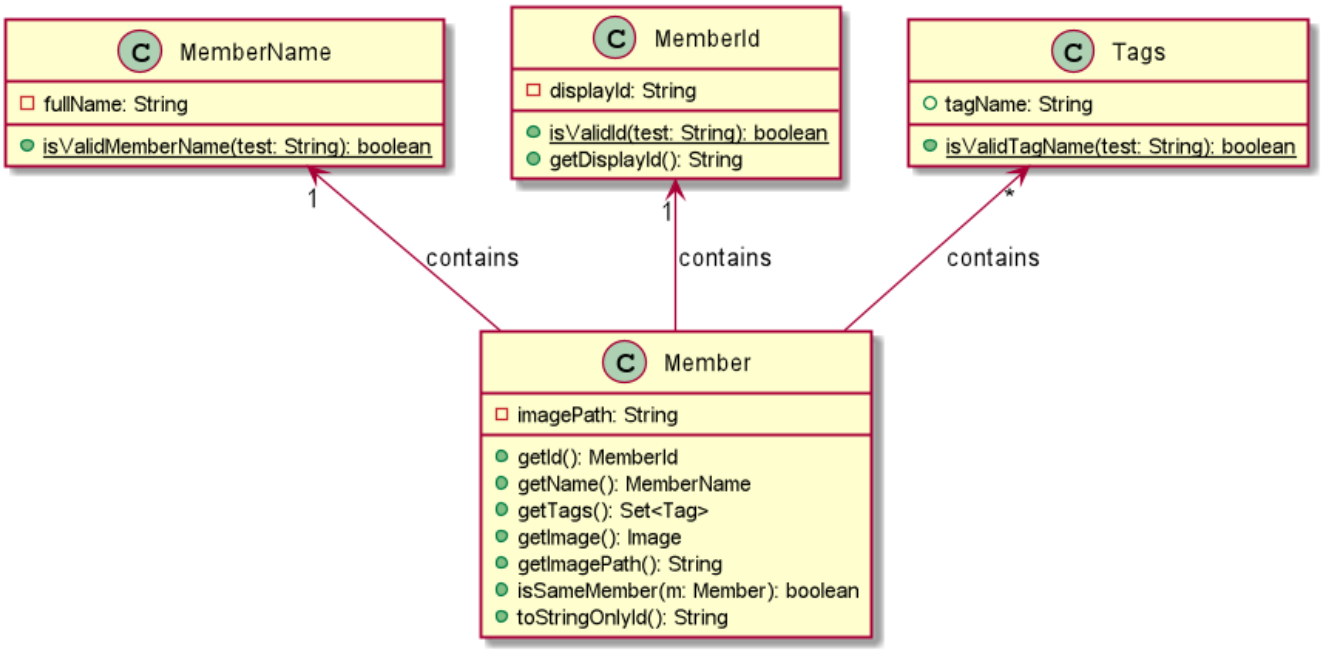


Diagram 13. Class diagram of Member package

Apart from the typical commands (`add-member`, `delete-member`, `find-member`) involved in such a central class, the member features also introduce a `set-image` command. The `set-image` command allows users to set an image in their computer as the profile picture of a member in +Work. To accommodate the `set-image` command, the `Member` class has an alternative constructor that takes in the image filepath as a parameter to save it as an attribute to the member object, when `set-image` command is called. Additionally, to support the command, the `Member` class implements the following operation:

- `Member#getImagePath()` — Retrieves the filepath of the image stored in the user's computer
- `Member#getImage()` — Retrieves the member's image using the image filepath

Given below is an example usage scenario and how the set-image mechanism behaves at each step.

Step 1. The user launches the application for the first time, and adds a team member into +Work. The member is displayed with a default profile picture.

Step 2. The user executes the `set-image` command to set an image in their computer as the profile picture of a member in +Work..

The `set-image` command calls `Model#getFilteredMembersList()` to retrieve the Member that is to be edited. A new member object is formed, with all the same parameters as the specified member object, and a new Image Filepath parameter. `Model#setMember` is called to replace the old member object with the new one in +Work.

The following sequence diagram shows how the `set-image` operation works.

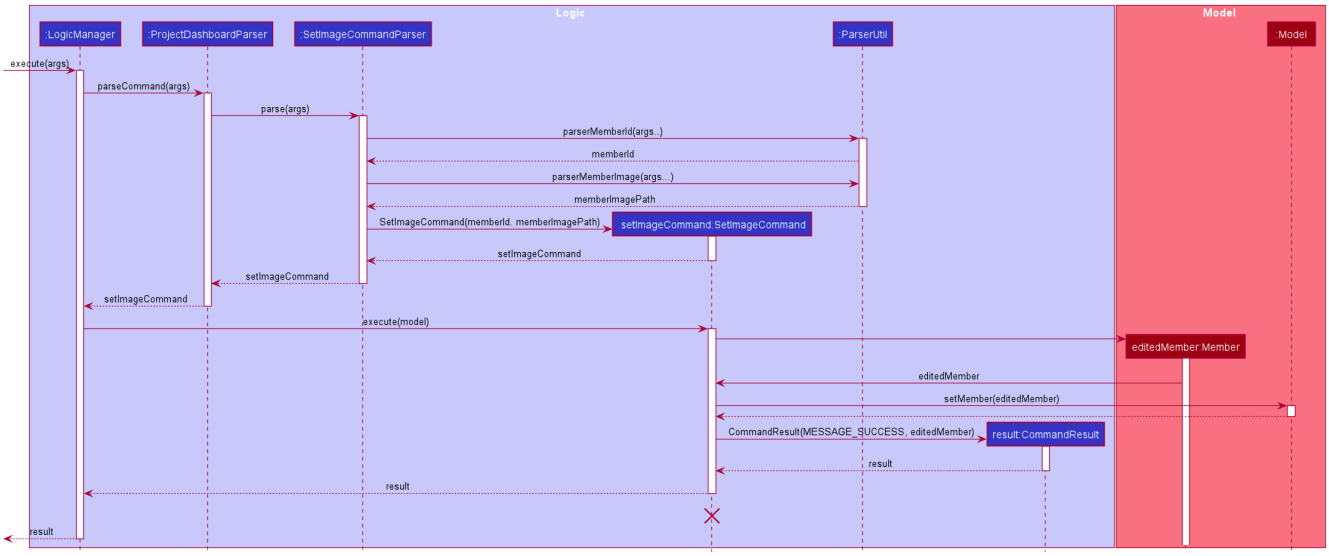


Figure 14. Operational flow of `SetImageCommand`

NOTE The image's file path is stored in the Member object. If the image is shifted to another location, the file path stored becomes invalid, and the user has to call the `set-image` command again, with the new file path.

Step 3. When an operation is called to display a member, `Member#getImage` is called to display the image using Javafx's ImageView.

The following sequence diagram shows how the image is called up and subsequently displayed in the +Work for an individual member.

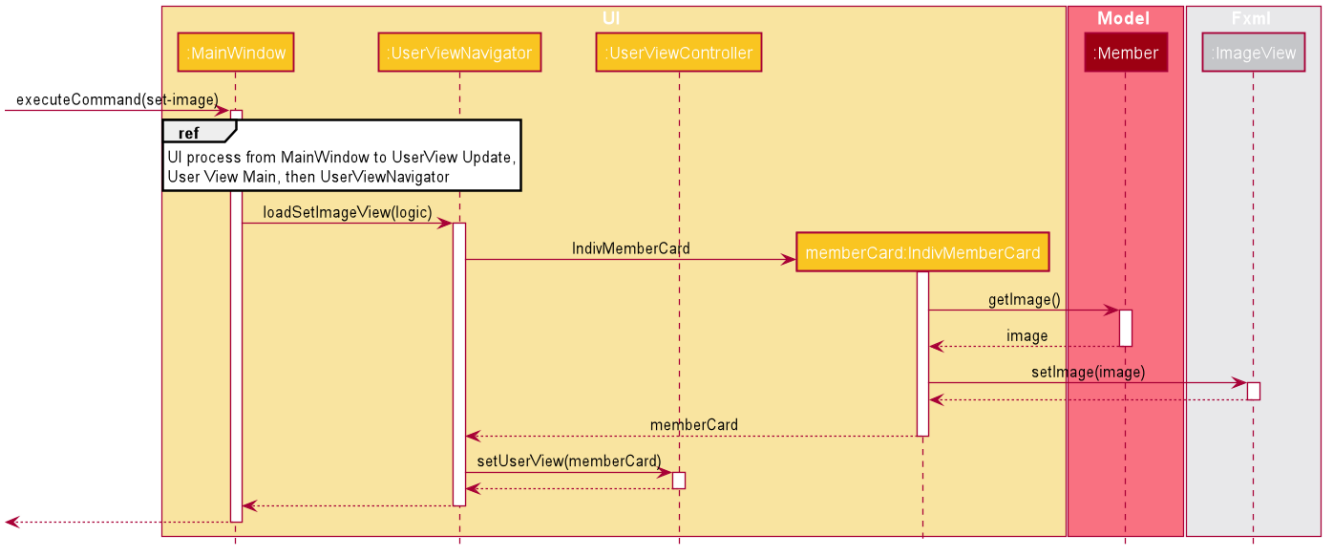


Figure 15. Operational flow of displaying a member with his profile picture

NOTE The sequence diagram only shows how a member's profile image is called up and displayed. It doesn't show how the member's name and tags are displayed, since this is very similar to how AB3 originally displays its `Person` name and tags.

The following activity diagram summarizes what happens when a user executes the `set-image` command:

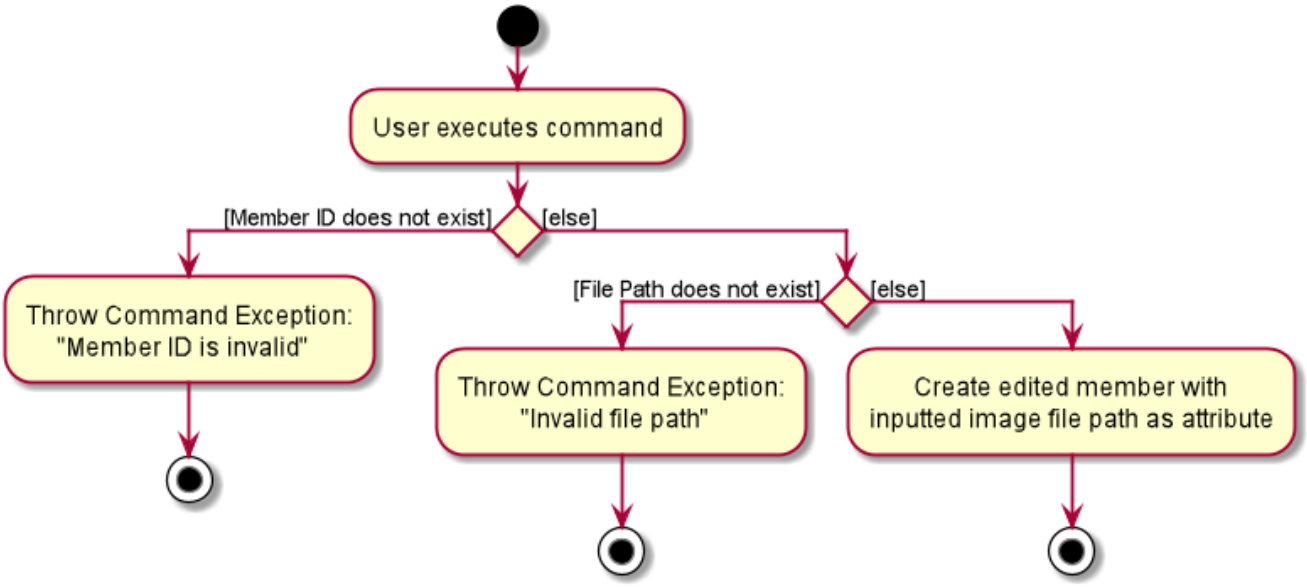


Figure 16. Operational flow during execution of `set-image` command

2.5.2. Design Considerations

This section describes the pros and cons of the current and other alternative implementations of the image attribute under members, as well as the display of members in +Work.

Aspect: Storage of image under member

- **Alternative 1:** Storing the image filepath as a changeable attribute
  - Pros: Editing a member's profile picture involves accessing the member and changing its file path attribute
  - Cons: The image file path attribute is exposed to the rest of the classes in +Work and may be unintentionally edited, causing the member's profile picture to be edited without the intention to.
- **Alternative 2 (current choice):** Storing the image filepath as a final attribute
  - Pros: Ensures the member's image filepath remains unchangeable and specific to the member
  - Cons: A new member object has to be created to replace the member being edited every time the member's profile picture is updated

Alternative 2 was chosen to keep in line with the original structure of the Person object in AB3, with all attributes being final and unchangeable.

Aspect: Display of members

- **Alternative 1:** Display each member with only its member name, ID and profile picture
  - Pros: Concise display of each member in +Work, with only the essential information being exposed
  - Cons: Less details of individual members are displayed, making it difficult to draw links between members and the tasks they are involved in

- **Alternative 2 (current choice):** Display members with its member name, ID, profile picture and tasks assigned
    - Pros: Includes more details for individual members, which increases the ease in which the user can identify a member's responsibilities
    - Cons: Display of members is cluttered, and may expose unnecessary information in certain situations

Alternative 2 was chosen because it is more in line with +Work's objective of drawing easy comparison between project members and tasks.

2.6. Inventory feature

2.6.1. Proposed Implementation

This feature was implemented to allow users to add inventories when using +Work.

The commands introduced by this feature include; `add-inv`, `delete-inv`, `edit-inv`, `list-inv`. The commands are facilitated by UniqueInventoryList class which resides in Model. The UniqueInventoryList class implements the following operations:

- `UniqueInventoryList #add(Inventory toAdd)` — This command adds the Inventory toAdd to the inventory list of +Work.
- `UniqueInventoryList #remove(Inventory toAdd)` — This command removes the Inventory toAdd from the inventory list of +Work.
- `UniqueInventoryList #setInventory (Inventory target, Inventory editedInventory)` — This command replaces the target Inventory with the new editedInventory.

These operations are exposed in the Model interface as `Model#addInventory(Inventory inventory)`, `Model#deleteInventory(Inventory target)`, `Model#setInventory(Inventory target, Inventory editedInventory)` respectively.

The activity diagram below summarises the process of executing an `add-inv` command.

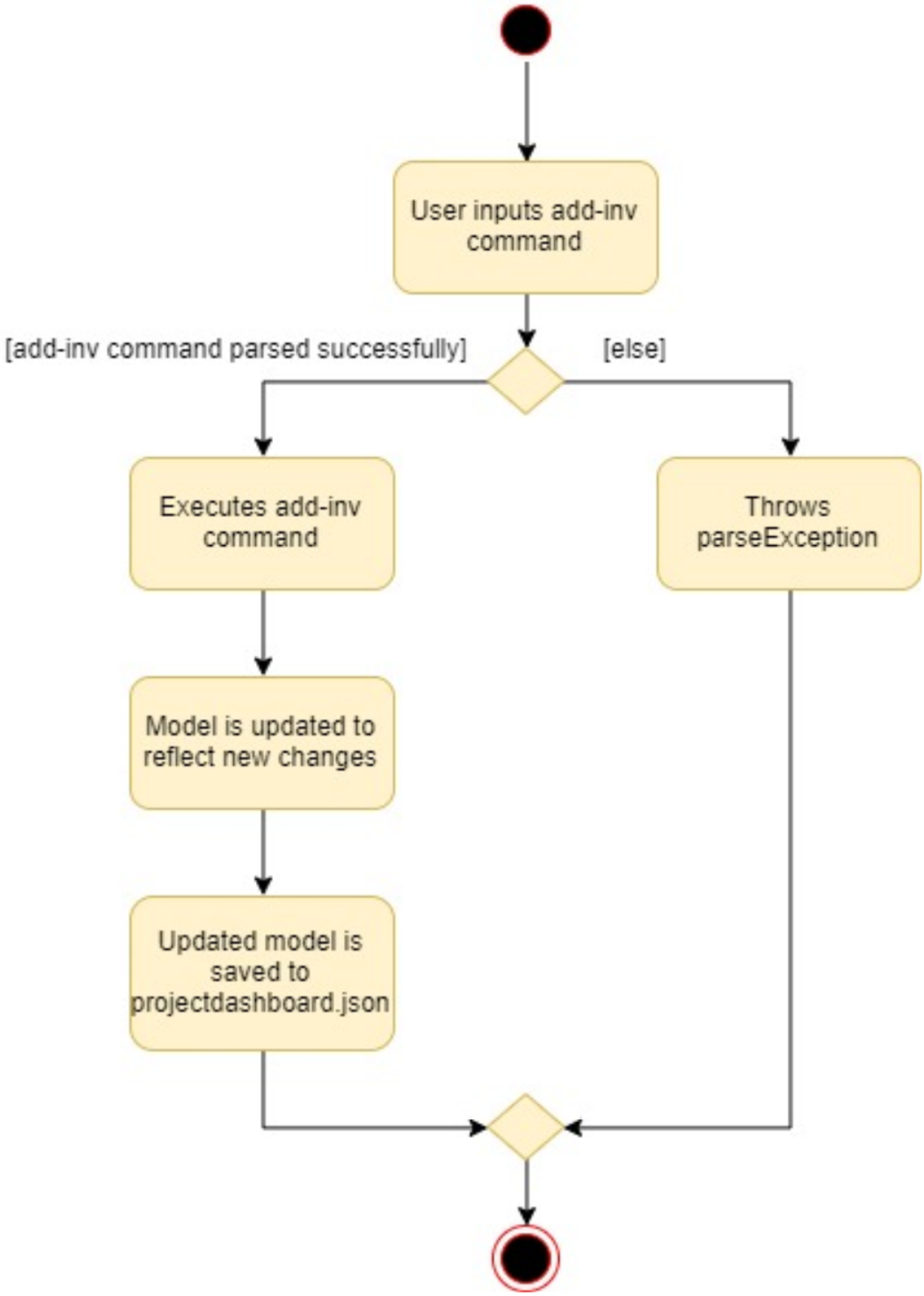


Figure 14. Activity diagram of `add-inv` *command execution.*

Given below is an example usage scenario and how add-inv command behaves at each step:

Step 1. The user launches the application. The inventories will be initiated by Model based on the

inventories previously saved.

Step 2. The user executes `add-inv` command.

Step 3. `Logic#execute()` calls `Model#addInventory(Inventory inventory)`, which calls `UniqueInventoryList #add(Inventory toAdd)`. This adds the inventory to UniqueInventoryList.

Step 4. The UI will be updated to reflect the changes. This can be viewed using the `list-inv` command.

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

The following sequence diagram shows how the add-inv command works.

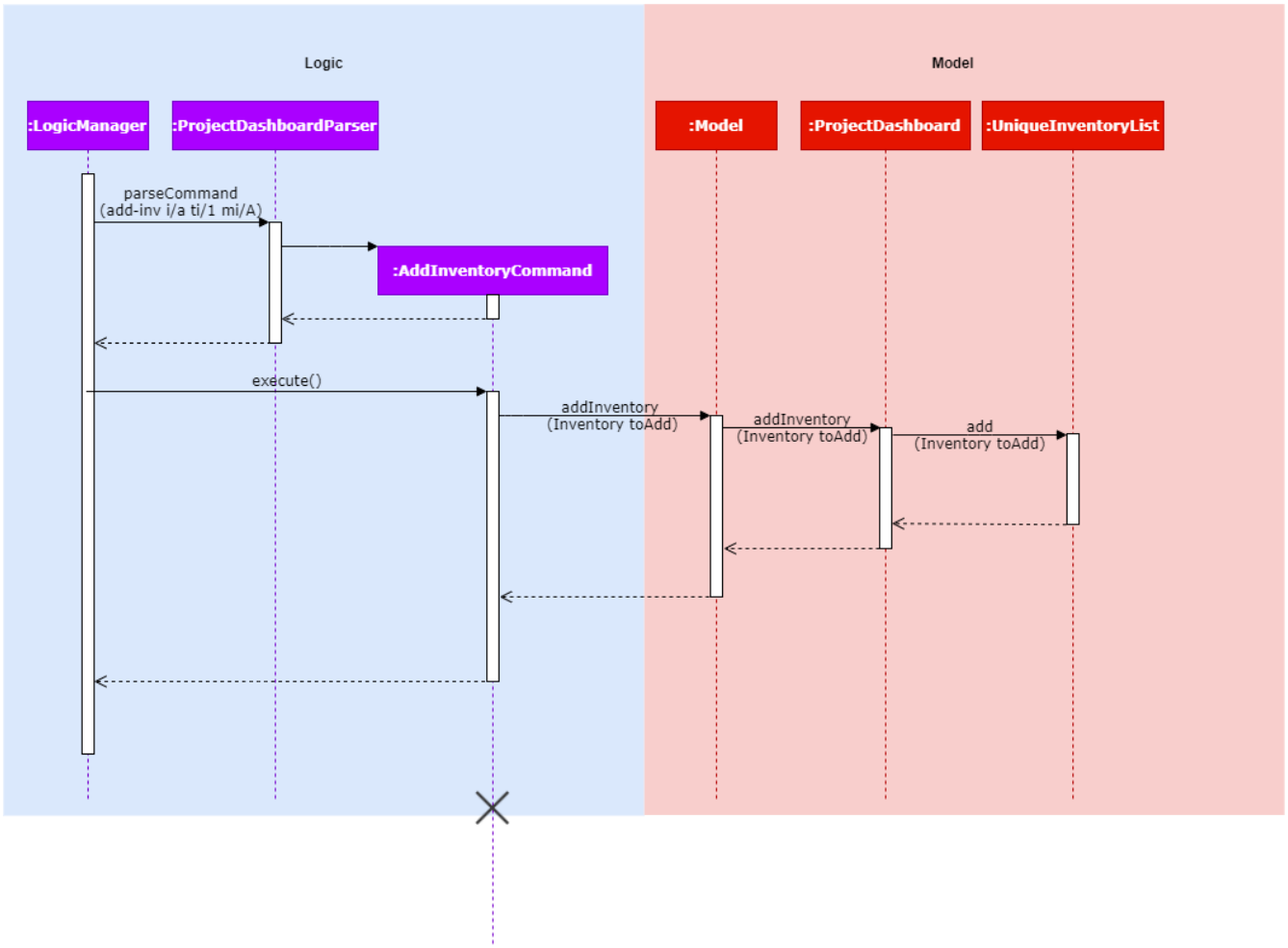


Figure 15. Operational flow of `AddInventoryCommand`

2.6.2. Design Considerations

This section explores how the design can affect the inventory features in +Work.

Aspect: Storage of an inventory

Within a specific setting, each option has data that helps yield a specific behaviour. Currently the available settings are represented as enumerations.

- **Alternative 1 (current choice)** : Each inventory only stores the inventory name and price. The

task and member attached to each inventory is stored in UniqueMappingManager.

    - Pros: Better design as it is more modular. If task or member is deleted, it only needs to be changed in UniqueMappingManager for the expected behaviour to be achieved.
    - Cons: Retrieving of the mappings (task and member attached) is more difficult and may result in bugs if not implemented accurately.

- **Alternative 2**: Each inventory also contains the attribute for task attached and member attached.
    - Pros: Retrieving the mappings is easier and faster.
    - Cons: When a task or member is deleted, all the inventories need to be checked and updated. This would be a very slow method.

Aspect: Display of inventories list

- **Alternative 1 (current choice)**: The inventories are listed without any classification and is not sorted by any attributes.
    - Pros: This would be easier to implement and maintain UI components.
    - Cons: User must use the pdf method to see any statistics, making for a less user-friendly experience.
- **Alternative 2**: More statistics, classifications and sorting methods available to customize the inventories list.
    - Pros: The inventories list is more user-friendly and more provides more details.
    - Cons: Implementation is harder, and we have to ensure minimal errors or bugs.

2.7. AutoComplete feature

2.7.1. Proposed Implementation

- **Alternative 1 (current choice)**: Logic handles the autocomplete logic. While Commandbox will receive the result and populate the context menu based on output from logic.
    - Pros: Easy to change autocomplete logic in the future if need be, such as integrating prefix suggestions, which require the dashboard's data;
    - Cons: Need to access the autocomplete component. Hard to pass props to logic with regards to textbox like caret position.
- **Alternative 2**: Handle all autocomplete logic within command box
    - Pros: This would be easier to implement and maintain UI components. Can access TextField directly.
    - Cons: Difficult to access other logic components. Handles both UI and logic in the same component.]

Given below is an example usage scenario and how AutoComplete behaves at each step:

Step 1. The user launches the application. The command box is initialized together in the main

window. Logic is initialised and passed into command box.

Step 2. The user attempts to type a command `input` in Command Box.

Step 3. Command Box calls `Logic#getAutoCompleteResults()` which calls `AutoComplete#completeText(String input)`, which calls `Keywords #commandList(String input)`. This returns a filtered Linked List of possible commands.

Step 4. The Linked List of commands will be passed back into Command Box who will call `populatePopup(LinkedList<String> searchResult)` to make its own ContextMenu to be displayed.

Step 5. The UI now reflects a list of available commands filtered based on text.

The following sequence diagram shows how the autocomplete works.

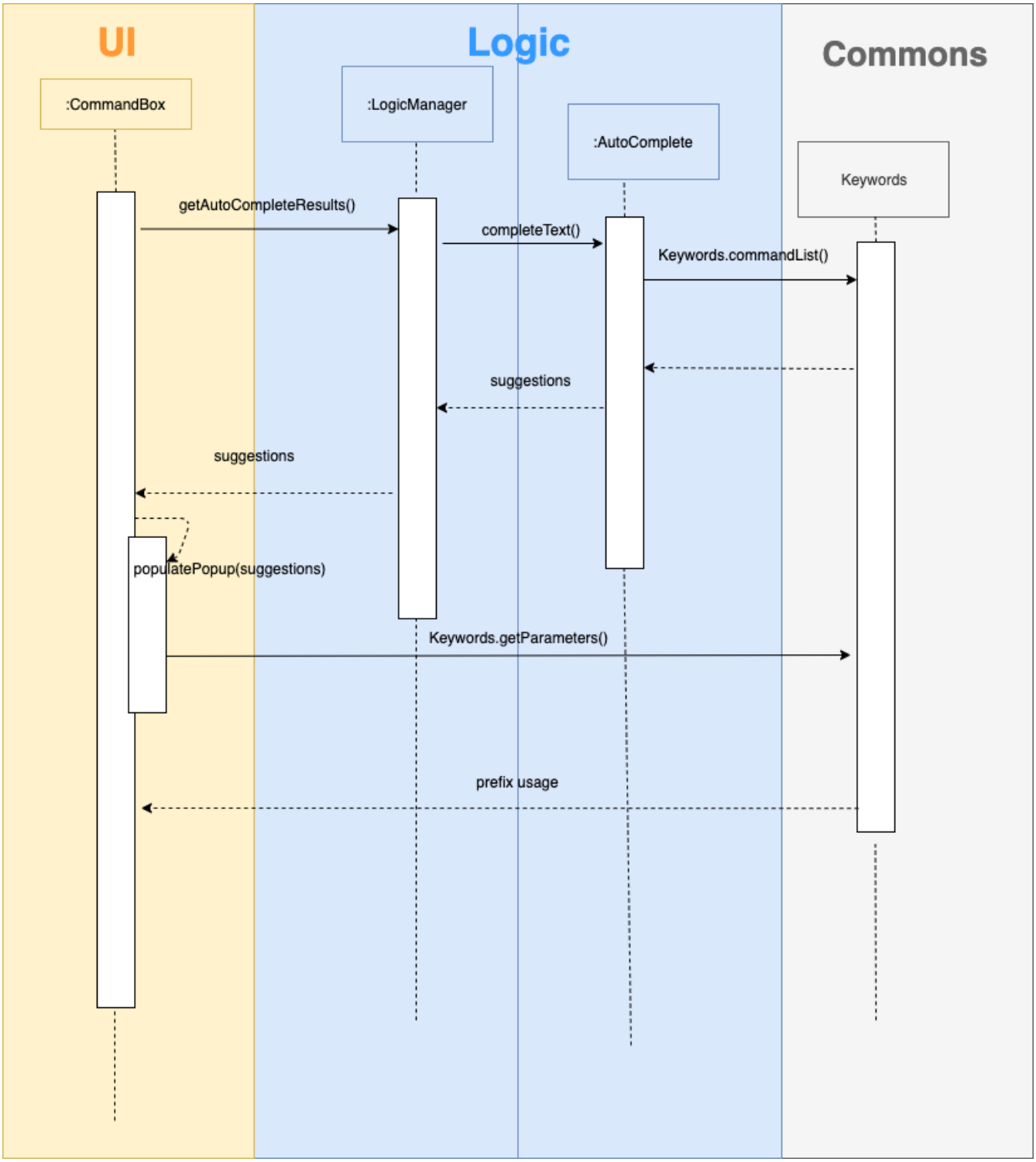


Figure 16. Operational flow of `getAutoCompleteResults()`

2.8. Undo/Redo feature

2.8.1. 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.

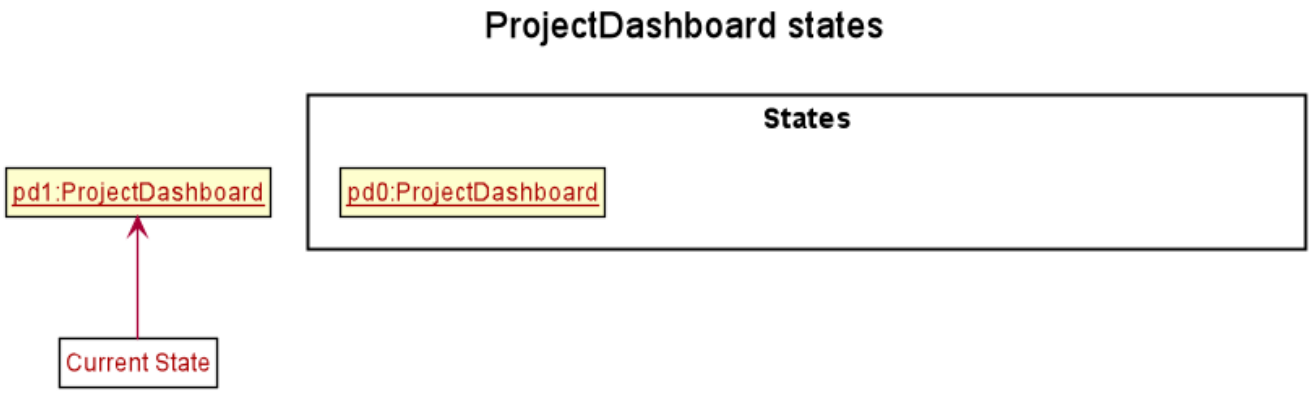


Figure 17. Storing previous ProjectDashboard states

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`.

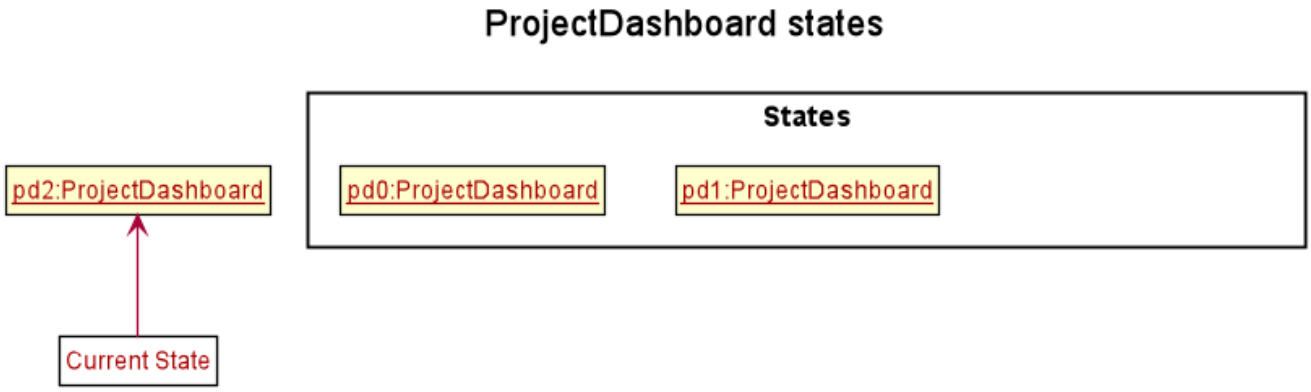


Figure 18. Executing more commands

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.

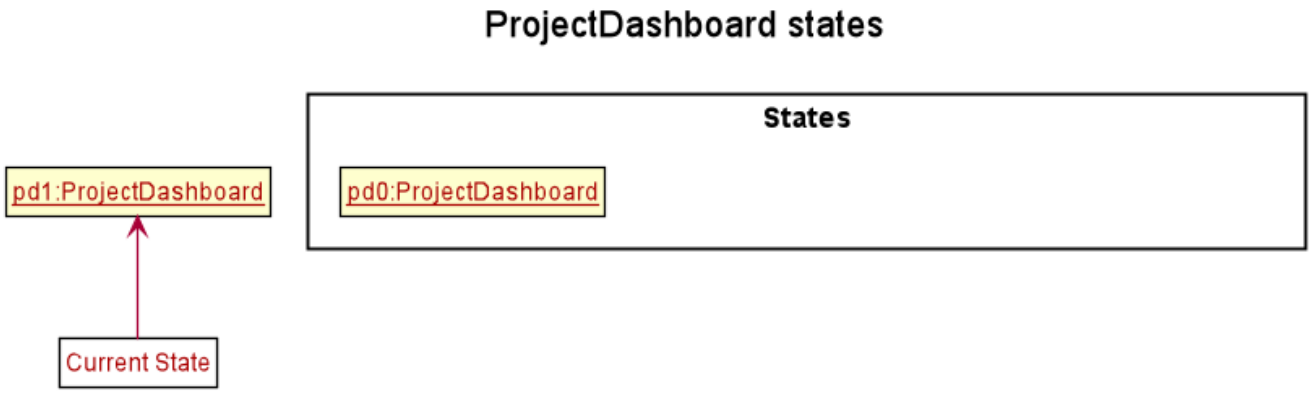


Figure 19. Retrieving a previous state

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

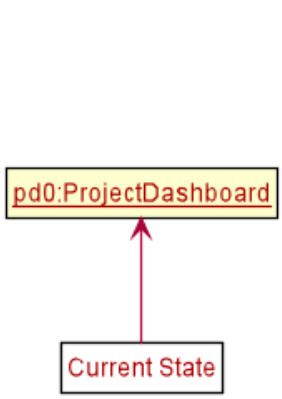


Figure 20. Retrieving previous states

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:

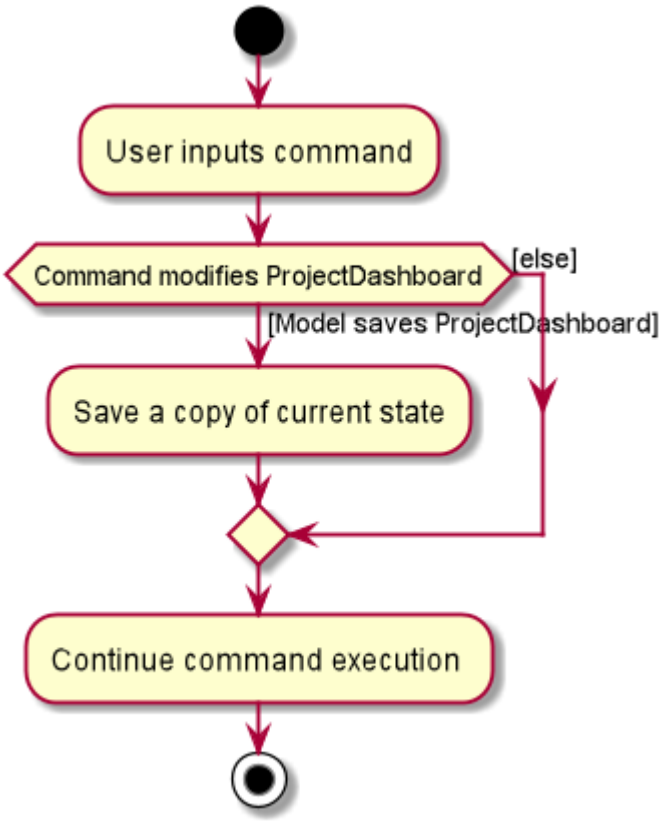


Figure 21. Executing a new command

The following sequence diagram shows how the undo operation works:

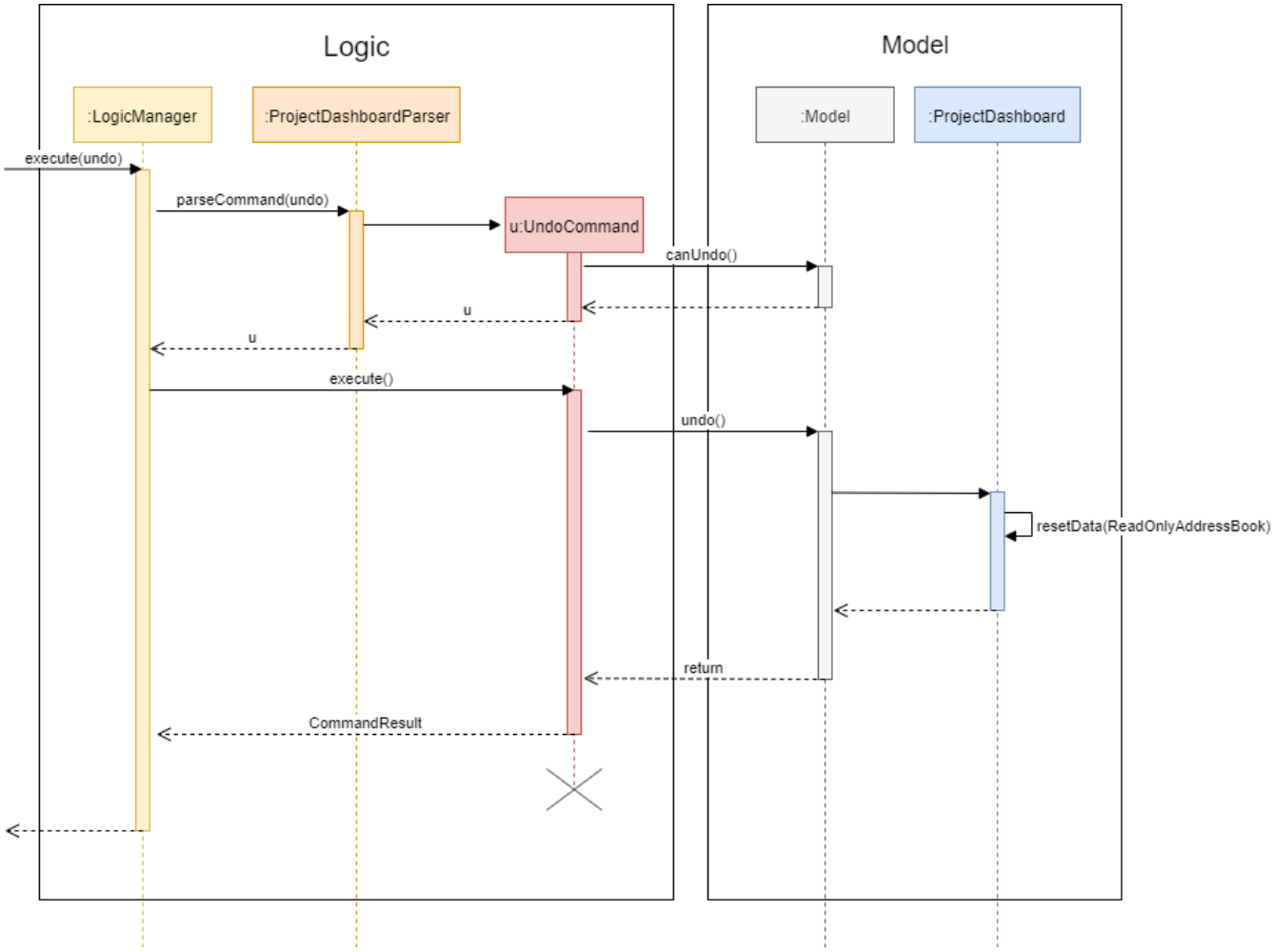


Figure 22. Interactions between Logic and Model

2.8.2. Design Considerations

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` couterpart.
  - 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.

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.

2.9. [Proposed] Data Encryption

{Explain here how the data encryption feature will be implemented}

2.10. Logging

We are using `java.util.logging` package for logging. The `LogsCenter` class is used to manage the logging levels and logging destinations.

- The logging level can be controlled using the `logLevel` setting in the configuration file (See Section 2.11, “Configuration”)
- The `Logger` for a class can be obtained using `LogsCenter.getLogger(Class)` which will log messages according to the specified logging level
- Currently log messages are output through: `Console` and to a `.log` file.

Logging Levels

- `SEVERE` : Critical problem detected which may possibly cause the termination of the application
- `WARNING` : Can continue, but with caution
- `INFO` : Information showing the noteworthy actions by the App
- `FINE` : Details that is not usually noteworthy but may be useful in debugging e.g. print the actual list instead of just its size

2.11. Configuration

Certain properties of the application can be controlled (e.g user prefs file location, logging level) through the configuration file (default: `config.json`).

3. Documentation

Refer to the guide here.

4. Testing

Refer to the guide here.

5. Dev Ops

Refer to the guide here.

Appendix A: Product Scope

Target user profile:

- an NUS student
- managing an NUS project team.
- has a significant amount of tasks to manage among team members
- can type fast
- prefers desktop apps over mobile apps
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition: manage tasks assigned to project mates, finding common time slots and keep track of inventory faster than GUI apps.

Appendix B: User Stories

Priorities: High (must have) - * * *, Medium (nice to have) - * *, Low (unlikely to have) - *

Priority	As a …	I want to …	So that I can…
* * *	New user	see usage instructions	refer to instructions when I forget how to use the App
* * *	Project leader	Add a project member	Assign tasks to them
* * *	Project leader	Add tasks to the project	
* * *	Project leader	Change task status	Get reminder of the progress of each task
* *	User	Change the app theme	Suit my viewing preferences
* *	User	Change the deadline display format	Suit my time preferences

Priority	As a …	I want to …	So that I can…
* *	Project leader	Sync team members schedules	Find a time slot where the maximum number of people, if not all, can attend
* *	Project leader with many members and tasks	Find out which group members are working on a particular task	Distribute workload evenly
* *	Project leader with tasks that require inventory	Assign inventory to tasks or materials that are needed	Download an inventory report whenever required
* *	Project leader	Create a meeting	Secure meeting slot at my desired time and place
* *	Project leader	View my next meeting in the home page	I know my team’s schedule at a glance
* *	Project leader who is keeping track of purchases	Tag the inventory purchase to the member who bought it	Produce an accurate claims report at the end of the project
*	Project leader managing a long-term project with ad-hoc members	Change members working on a task	Reassign tasks to incoming members

{More to be added}

Appendix C: Use Cases

(For all use cases below, the **System** is the `ProjectDashboard` and the **Actor** is the `user`, unless specified otherwise)

Use case: Add a team member

Main Success Scenario

1. User requests to add a team member and gives name of member
2. +Work informs user that member was successfully added

   Use case ends.

Extensions

- 1a. The user does not specify name
  - 1a1. +Work requests user for a name
  - 1a2. User enters members name
  - Steps 1a1 - 1a2 are repeated until user provides a name

    Use case resumes from step 2.

Use case: List all team members

Main Success Scenario

1. User requests to list team members
2. +Work displays list of team members

   Use case ends.

Extensions

- 2a. The list is empty
  - 2a1. +Work informs user that there are no members added yet

    Use case ends.

Use case: Delete member

Main Success Scenario

1. User requests to list members
2. +Work shows a list of persons

3. User requests to delete a specific person in the list and specifies the ID of the person
4. +Work deletes the person

   Use case ends.

Extensions

- 1a. The list is empty.
  - 1a1. +Work informs user that there are no members

    Use case ends.
- 1b. The user specifies an invalid member ID
  - 1b1. +Work informs user that they have entered an invalid ID
  - 1b2. User enters a valid ID
  - Steps 1b1 - 1b2 are repeated until user provides a valid ID

    Use case resumes from step 2.

Use case: Assign a task to a team member

Main Success Scenario

1. User requests to assign a task to a team member and specifies the task ID and the corresponding team member ID
2. +Work informs user that task has been assigned to the member successfully
3. +Work updates the dashboard

   Use case ends.

Extensions * 1a. The user specifies an invalid team member and/or task ID **1a1. +Work informs user that they have entered an invalid ID** 1a2. User enters a valid ID ** Steps 1a1 - 1a2 are repeated until user provides a valid ID

+ Use case resumes from step 2.

- 1b. The user does not specify either/both team member and member ID
  - 1b1. +Work informs user that they need to enter a valid ID
  - 1b2. User enters specifies the valid ID
  - Steps 1b1 - 1b2 are repeated until user provides a valid ID

    Use case resumes from step 2.

Use case: Remove a task for a team member

Main Success Scenario

1. User requests to remove a task for a team member and specifies the task ID and the corresponding team member ID
2. +Work informs user that member is not assigned to task any longer
3. +Work updates the dashboard

   Use case ends.

Extensions

- 1a. The user specifies an invalid team member and/or task ID
  - 1a1. +Work informs user that they have entered an invalid ID
  - 1a2. User enters a valid ID
  - Steps 1a1 - 1a2 are repeated until user provides a valid ID

    Use case resumes from step 2.
- 1b. The user does not specify either/both team member and member ID
  - 1b1. +Work informs user that they need to enter the valid ID
  - 1b2. User enters specifies the valid ID
  - Steps 1b1 - 1b2 are repeated until user provides a valid ID

    Use case resumes from step 2.

Use case: Add a task

Main Success Scenario

1. User requests to add a task and specifies the name of the task and may specify the member id of the member assigned to the task
2. +Work informs the user that task has been updated
3. +Work updates the dashboard

   Use case ends.

Extensions

- 1a. The user specifies an invalid team member ID and/or does not give a task name
  - 1a1. +Work informs user that they have entered an incomplete command
  - 1a2. User enters a valid command
  - Steps 1a1 - 1a2 are repeated until user provides a valid command

    Use case resumes from step 2.

Use case: Mark a task as ‘done’

Main Success Scenario

1. User requests to mark a task as ‘done’ and specifies the ID of the task
2. +Work informs user that the task is marked as done successfully
3. +Work updates the dashboard

   Use case ends.

Extensions

- 1a. The user specifies an invalid task ID
  - 1a1. +Work informs user that they have entered an invalid ID
  - 1a2. User enters a valid ID
  - Steps 1a1 - 1a2 are repeated until user provides a valid id

    Use case resumes from step 2.

Use case: Mark a task as ‘doing’

Main Success Scenario

1. User requests to mark a task as ‘doing’ and specifies the ID of the task
2. +Work informs user that the task is marked as ’doing’ successfully
3. +Work updates the dashboard

   Use case ends.

Extensions

- 1a. The user specifies an invalid task ID
  - 1a1. +Work informs user that they have entered an invalid ID
  - 1a2. User enters a valid ID
  - Steps 1a1 - 1a2 are repeated until user provides a valid id

    Use case resumes from step 2.

Use case: List all tasks in the dashboard

Main Success Scenario

1. User requests to list all tasks in the dashboard
2. +Work displays list of tasks

Use case ends.

Extensions

- 1a. The list is empty
  - 1a1. +Work informs user that there are no tasks

    Use case ends.

Use case: Remove a task from the dashboard

Main Success Scenario

1. User requests to remove a task from the dashboard and specifies the task ID
2. +Work informs the user that task is removed succesfully
3. +Work updates the dashboard

   Use case ends.

Extensions

- 1a. The user specifies an invalid task ID
  - 1a1. +Work informs user that they have entered an invalid ID
  - 1a2. User enters a valid ID
  - Steps 1a1 - 1a2 are repeated until user provides a valid id

    Use case resumes from step 2.

Use case: Assign a deadline to a task

Main Success Scenario

1. User requests to assign a deadline to a task and specifies the task ID and the corresponding deadline
2. +Work informs user that task deadline set successfully
3. +Work updates the dashboard

   Use case ends.

Extensions

- 1a. The user specifies an invalid task ID
  - 1a1. +Work informs user that they have entered an invalid ID
  - 1a2. User enters a valid ID

    - Steps 1a1 - 1a2 are repeated until user provides a valid ID

      Use case resumes from step 2.

- 1b. The user gives the deadline in the wrong format
    - 1b1. +Work informs user that deadline must be in the format `dd-mm-yy hh:mm`
    - 1b2. User enters the deadline in the correct format
    - Steps 1b1 - 1b2 are repeated until user provides a valid deadline

      Use case resumes from step 2.

Use case: Generate availability timings of team members

Main Success Scenario

1. User adds timetable of team mates to +Work
2. User requests to generate availability timings of team members
3. +Work displays list of timings where the most number of team members are available

   Use case ends.

Extensions

- 2a. +Work finds that there are no available timings
    - 2a1. +Work informs user that no available timings were found

      Use case ends.

Use case: Add a meeting to the timetable

Main Success Scenario

1. User requests to add a meeting and specifies a description and the time of the meeting
2. +Work informs user that meeting was successfully created
3. +Work updates the dashboard

   Use case ends.

Extensions

- 1a. User specifies the time in an invalid format
    - 1a1. +Work informs user that meeting time must be in format `dd-mm-yy hh:mm`
    - 1a2. User enters the time in the correct format

- Steps 1a1 - 1a2 are repeated until user provides a valid time

  Use case resumes from step 2.

Use case: Add an item to the inventory

Main Success Scenario

1. User requests to add an item to the inventory and specifies the name and price of the inventory item, ID of the member as well as task associated with the item
2. +Work informs user that the inventory has been added successfully
3. +Work updates the inventory

   Use case ends.

Extensions

- 1a. User specifies an invalid or missing value
  - 1a1. +Work informs the user that command is incomplete
  - 1a2. User enters complete command
  - Steps 1a1 - 1a2 are repeated until user provides complete command

    Use case resumes from step 2.

Use case: Delete an item from the inventory

Main Success Scenario

1. User requests to delete an item from the inventory and specifies the item ID
2. +Work informs user that the inventory item has been successfully deleted
3. +Work updates the inventory

   Use case ends.

Extensions

- 1a. The user specifies an invalid inventory ID
  - 1a1. +Work informs user that they have entered an invalid ID
  - 1a2. User enters a valid ID
  - Steps 1a1 - 1a2 are repeated until user provides a valid ID

    Use case resumes from step 2.

Use case: Edit an inventory item

Main Success Scenario

1. User requests to edit an item to the inventory and specifies the ID of the inventory item first, followed by the parameter(s) to be edited
2. +Work informs user that the inventory item has been successfully edited
3. +Work updates the inventory

   Use case ends.

Extensions

- 1a. The user specifies an invalid inventory ID or specifies no parameters to be edited
  - 1a1. +Work informs user that they have entered an invalid ID and at least one parameter
  - 1a2. User enters a valid ID and the parameter
  - Steps 1a1 - 1a2 are repeated until user provides a valid ID and a parameter to be changed

    Use case resumes from step 2.

Use case: Generate an inventory report

Main Success Scenario

1. User requests to generate an inventory report
2. User specifies whether inventory is generated based on the member or task
3. +Work displays the inventory report

   Use case ends.

Use case: Toggle the display theme of +Work

Main Success Scenario

1. User requests to toggle the theme between light and dark
2. +Work displays the requested theme

   Use case ends.

Use case: Toggle the clock display format of task deadlines

Main Success Scenario

1. User requests to toggle the clock between 24 hour and 12 hour
2. User enters `home` and switches to the dashboard
3. +Work displays the requested clock format for task deadlines

   Use case ends.

Appendix D: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java 11 or above installed.
2. Should be able to hold up to 1000 tasks without a noticeable sluggishness in performance for typical usage.
3. A user with above average typing speed for regular English text (i.e. not code, not system admin commands) should be able to accomplish most of the tasks faster using commands than using the mouse.

Appendix E: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

NUS

National University of Singapore

CLI

command line interface (bash, git)

GUI

graphical user interface

Appendix F: Product Survey

Product Name

Author: …

Pros:

- …
- …

Cons:

- …
- …

Appendix G: Instructions for Manual Testing

Given below are instructions to test the app manually.

NOTE	These instructions only provide a starting point for testers to work on; testers are expected to do more *exploratory* testing.

G.1. Launch and Shutdown

1. Initial launch
    a. Download the jar file and copy into an empty folder
    b. Double-click the jar file
    Expected: Shows the GUI with the dashboard. The window size is optimum to view the dashboard entirely.
2. Altering window preferences
    a. Resize the window to an optimum size. Move the window to a different location. Close the window.
    b. Re-launch the app by double-clicking the jar file.
    Expected: If the window size is less than the minimum, it will be re-sized automatically. Else, the window preferences will be retained.

G.2. Member Feature

1. Add a member:
    a. Test case: "add-member mn/YOUR_NAME mi/T1 mt/testing"
    Expected: Shows that a member is successfully added into +Work. Entering "list-members" in command prompt displays a list of members, including the one newly added.
    b. Test case: "add-member mn/YOUR_NAME mi/test!!! mt/testing"
    Expected: Invalid member ID, since member ID has to be alphanumeric.
    c. Test case: "add-member mn/YOUR_NAME mi/T2 testing"
    Expected: A member is successfully added to +Work, with member ID "T2 testing"
2. Edit a member:
    a. Test case: "edit-member mi/T1 mn/NICKNAME"
    Expected: Shows that the previously added member is successfully edited, with a name change from YOUR_NAME to NICKNAME.
    b. Test case: "edit-member mi/T1"
    Expected: Fails to edit the specified member given the lack of fields to edit.
3. Set an image:
    a. Test case:
        i. In your laptop, find an image that you would like to set as the profile picture of a member. Take note of the image's filepath.

ii. In the command prompt, enter "set-image mi/T1 im/FILE_PATH". Note that file path should end with 'IMAGE_NAME.png'.
Expected: The member with member id 'T1' has a new profile picture, depicting the image you chose from your laptop.

b. Test case: "set-image mi/T1 im/random string"
Expected: No image is set for the specified member, as an image cannot be found.

4. Assign a member to a task:

a. Test case: "assign ti/1 mi/T1"
Expected: Shows that the member with member id 'T1' has one more task added to his list of assigned tasks. Entering "list-tasks" in command prompt displays the list of tasks in +Work, with the involved member listed under the task at index 1, as assigned.

b. Test case: "assign ti/x mi/t1" where x is larger than the number of tasks
Expected: +Work cannot assign the member since the task does not exist.

c. Other test case to try: "assign ti/1 mi/x" where x is an invalid member ID

5. Fire a member from a task:

a. In the command prompt, enter "fire ti/1 mi/T1"
Expected: Shows that the task with index 1 has been removed from member with member id 'T1' (under his 'list of tasks assigned'). Entering "list-tasks" in command prompt displays the list of tasks in +Work, with the involved member removed from the task at index 1.

G.3. Statistics Feature

1. Obtaining member-related statistics:

a. Test case: "member-stats"
Expected: Shows the GUI with two pie charts, relating to the amount of tasks and inventory items under each member.

b. Test case: "testing member-stats"
Expected: +Work is unable to recognise this as a valid command.

c. Test case: "member-stats testinggg"
Expected: +Work recognises this command as member-stats, and displays the GUI.

d. Test case: Assign / remove (fire) more tasks to/from any member, and enter the command "member-stats" once more.
Expected: Pie charts displayed changes accordingly.

2. Obtaining task-related statistics:

a. In the command prompt, enter "task-stats"
Expected: Shows the GUI with a list of tasks and the time taken to complete them on the left, and a pie chart relating to the number of tasks of each status (unbegun, doing, done) on the right.

b. Test case: "testing task-stats"
Expected: +Work is unable to recognise this as a valid command.

c. Test case: Use command "add-task tn/name of task ts/unbegun" to add a task to +Work. Then, use commands "doing-task ti/" and "done-task ti/" on the task at a few minutes interval,

before calling "task-stats" again.
Expected: The time taken for the newly added task has been updated accordingly.

d. Continue to add tasks, and change task status from unbegun to doing to done, to watch task-stats get updated.

G.4. Task commands

1. Deleting a task
   a. Prerequisites: List all tasks using the `list-tasks` command. Multiple tasks in the list.
   b. Test case: `delete-task ti/1`
   Expected: First task is deleted from the list. Details of the deleted task shown in the status message. Timestamp in the status bar is updated.
   c. Test case: `delete-task ti/0`
   Expected: No task is deleted. Error details shown in the status message. Status bar remains the same.
   d. Other incorrect delete commands to try: `delete-task`, `delete-task ti/x` (where x is larger than the list size)
   Expected: Similar to previous.
2. Editing a task
   a. Prerequisites: List all tasks using the `list-tasks` command. Multiple tasks in the list. Third task has no deadline.
   b. Test case: `edit-task ti/1 tn/New name`
   Expected: First task name is changed to "New name". Details of the edited task shown in the status message.
   c. Test case: `edit-task ti/3 at/10-10-2025 19:00`
   Expected: Third task is not edited. Error details shown in the status message.
   d. Other incorrect edit commands to try: `edit-task ti/1`, `edit-task ti/x` (where x is larger than list size)
3. Setting a deadline for a task
   a. Prerequisites: List all tasks using the `list-tasks` command. Multiple tasks in the list. First task has a deadline, third task has no deadline.
   b. Test case: `set-deadline ti/1 at/12-12-2020 19:00`
   Expected: Deadline is set for the task. Details of the task shown in the status message.
   c. Test case: `set-deadline ti/3 at/13-10-2031 18:00`
   Expected: Deadline not set for third task. Error details shown in the status message.
   d. Other incorrect set deadline commands to try: `set-deadline ti/1 at/10/10/2020`. `set-deadline ti/x at/10-10-2020 10:00` (where x is larger than list size) `set-deadline ti/2 at/time` (where time refers to a date in the past).
4. Completing a task
   a. Prerequisite: List all tasks using the `list-tasks` command. Multiple tasks in the list, second task with status `doing` and has deadline in less than two week. Third task has status `done`.

b. Test case: `done-task ti/2`
   Expected: Second task status is changed to `done`. Deadline is removed as well.

c. Test case: `done-task ti/3`
   Expected: Third task status is unchanged. Error details shown in the status message.

G.5. Dashboard feature

Test any commands related to manipulating `Task` and `Meeting` data, changes should be reflected in the dashbaord.

1. Navigating to the dashboard
   a. Prerequisites: Start at a different view of +Work.
   b. Test case: `home`
      Expected: View is switched to dashboard.
   c. Test case: `hom`
      Expected: View is not changed. Error details shown in the status message.
2. Viewing task data in the dashboard
   a. Prerequisites: View initial state of dashboard.
   b. Test case: Perform test 3b in [Task]. Expected: Task is removed from upcoming deadlines list and is moved to the `done` column of the dashboard.

G.6. Settings feature

1. Navigating to the settings panel
   a. Prerequisites: Start at a different view of +Work.
   b. Test case: `settings`
      Expected: View is switched to settings panel.
   c. Test case: `settin`
      Expected: View is not changed. Error details shown in the status message.
2. Changing the theme of +Work
   a. Prerequisites: Current theme is the `dark`.
   b. Test case: `theme light`
      Expected: +Work theme changes to light.
   c. Test case: `theme yellow`
      Expected: +Work theme is not changed. Error details shown in the status message.
   d. Other incorrect theme commands to try: `theme x` (Where x is not `light` or `dark`).
3. Changing the time format of +Work.
   a. Prerequisites: Current time format is 24 hour clock.
   b. Test case: `clock twelve`
      Expected: +Work time format changed to 12 hour clock. Switch view to task list to confirm.

c. Test case: `clock ten`
Expected: +Work time format is not changed. Error details shown in the status message.

d. Other incorrect theme commands to try: `clock x` (Where x is not `twelve` or `twenty_four`).

G.7. Calendar Feature

1. Adding a calendar

   a. Prerequisites: Downloaded a valid `.ics` file.

   b. Test case: `add-calendar mn/John fp/INVALID_FILE`
   Expected: +Work throws an error message

   c. Test case: `add-calendar mn/John fp/YOUR_FILE_PATH`
   Expected: +Work notifies you that the Calendar has been added

   d. Test case: `add-calendar mn/DUPLICATE_NAME fp/VALID_FILE_PATH`
   Expected: +Work notifies you that a duplicate calendar has been found

2. Removing a calendar

   a. Test case: `delete-calendar mn/John`
   Expected: If there is a calendar previously added under John's name, +Work notifies you that the calendar has been removed

3. Finding a meeting time

   a. Prerequisites: Added at least 1 calendar with the `add-calendar` command

   b. Test case: `find-meeting-time start/11-11-2019 end/12-11-2019 hours/1`
   Expected: +Work displays a list of suitable meeting times, while indicating which members are available

   c. Test case: `find-meeting-time start/12-11-2019 end/11-11-2019 hours/1`
   Expected: +Work prompts user to input a valid end date

   d. Test case: `find-meeting-time start/12-11-2019 end/15-11-2019 hours/-1`
   Expected: +Work prompts user to input a valid duration

4. Adding a meeting

   a. Test case: `add-meeting meeting/1`
   Expected: +Work adds the first meeting in the list and displays the meeting in the `home` page

   b. Test case: `add-meeting meeting/INVALID_INDEX`
   Expected: +Work displays an error message

   c. Prerequisites: Did not use the `find-meeting-time` command

   d. Test case: `add-meeting meeting/2`
   Expected: +Work prompts the user to first use the `find-meeting-time` command

5. Deleting a meeting

   a. Test case: `delete-meeting meeting/1`
   Expected: +Work removes the meeting

   b. Test case: `delete-meeting meeting/INVALID_INDEX`
   Expected: +Work displays an error message

G.8. Undo/Redo Feature

1. Undo a command
    a. Test case: `list-tasks`
    Expected: +Work will not undo the `list-tasks` command, and will find the next most recent command to undo
    b. Test case: `delete-meeting meeting/1`
    Expected: +Work will undo the deletion
    c. Prerequisites: App has just and no command has been executed
    d. Test case: `undo`
    Expected: +Work will display a message saying that there is no command to undo
2. Redo a command
    a. Prerequisites: No `undo` command has been called
    b. Test case: `redo` Expected: +Work will display a message saying that there is no command to redo
    c. Prerequisites: An `undo` command has been called
    d. Test case: `redo` Expected: +Work will redo the most recent `undo` command

G.9. Saving data

1. Dealing with missing/corrupted data files
    a. Prerequisites: Clone the repo and delete the `plusworksettings` and `projectdashboard` (under data folder) json files. Launch the application.
    b. Test case: Perform valid operations.
    c. Expected outcome: New data is saved and the previously missing `plusworksettings` and `projectdashboard` json files are created and contain the updated data.