Dinh Nho Bao - Project Portfolio

PROJECT: Mod Manager

Overview

Mod Manager is a desktop application that assists NUS students in **managing tasks, schedules, and contacts for their modules in a semester**. Our team, which consists of five software developers, took over an existing desktop Java application Address Book (Level 3) with about 6 KLoC and evolve it into our **Mod Manager** with more than 10 KLoC. The project spans over a period of eight weeks, where each of us idealise and design the product, utilise CI/CD for weekly project enhancements, implement features and functionality, write documentation (User Guide & Developer Guide), as well as take part in Quality Assurance.

Technologies: Java, JavaFx, GitHub, Intellij IDEA

The following sections document all the contributions that I have made to **Mod Manager**.

Summary of contributions

- **Code contributed**: I personally contributed more than 3 KLoC to Mod Manager. All my code contributions can be found here.
- **Major enhancement**: idealise and design the `Task` component of Mod Manager. The `Task` component allows NUS students to manage their tasks, such as programming assignments, homework, tutorials, reviewing lecture content, exam revision for their respective modules in a semester.
    - What it does: every `Task` has a description, a time frame (for example, `13/04/2020 23:59`), and a `Module` it belongs to. It allows users to organise and manage their tasks easier, with commands such as CRUD, mark the task as completed, view uncompleted tasks, view tasks by `Module`, find `Task` by its description, and search for a `Task` by its date, month, or year.
    - Justification: This feature improves the product significantly because every NUS student has a lot of things to do in the semester, and `Task` management offers a way for them to plan, and manage their tasks better. The `Task` component blends in well with `Mod Manager` and its other components, for example, every `Task` is allocated to an academic `Module` .
    - Highlights: The implementation of the project is tedious. It required a great amount of effort to understand the original AddressBook (Level 3) code given in order to morph and build upon this original project. Every small increment in the `Task` features require changes to multiple existing parts of the code, which requires using debugging tools and tracing the code to understand the code execution sequence. For example, with a small change in the `Task` class design, multiple `JUnit` test cases and data `Storage` design need to be changed. The `Task` component also has some dependencies on other components, as well as multiple other components have dependencies on the `Task` component. This requires good communication

skills between the team to notify each other every time the high-level design of `Task` or other components change, even with just a slight change. The `Task` component also required an in-depth analysis of design alternatives, which lead to incremental changes over the period of the project. For the commands implemented, input validation and data formatting is critical to avoid unexpected behaviours to our application.

    - Credits: The idea is adapted from AddressBook (Level 3) and the Duke project

- **Minor enhancement**: design and implement the UI to represent a `Task`. The design inspired other components of **Mod Manager** to adapt a similar presentation.

Dark red color indicates a task that is not yet done.

Green color indicates a task that is already completed.

- **Other contributions**:
    - Project management:
        - Managed releases `v1.3` - `v1.5rc` (3 releases) on GitHub
    - Enhancements to existing features:
        - Updated the GUI color scheme (Pull requests #33, #34)
        - Wrote additional tests for existing features to increase coverage from 88% to 92% (Pull requests #36, #38)
    - Documentation:
        - Did cosmetic tweaks to existing contents of the User Guide: #14
    - Community:
        - PRs reviewed (with non-trivial review comments): #12, #32, #19, #42
        - Contributed to forum discussions (examples: 1, 2, 3, 4)
        - Reported bugs and suggestions for other teams in the class (examples: 1, 2, 3)
        - Some parts of the history feature I added was adopted by several other class mates (1, 2)
    - Tools:
        - Integrated a third party library (Natty) to the project (#42)
        - Integrated a new Github plugin (CircleCI) to the team repo

{you can add/remove categories in the list above}

Contributions to the User Guide

Given below are sections I contributed to the User Guide. They comprise most of the commands for the `Task` component. They showcase my ability to write documentation targeting end-users.

NOTE For your easier understanding of the content below, every `Task` in a `Module` has an `ID`, which uniquely identifies the task in the module. The `ID` is important for us to differentiate a task from the others in the `Module`, since two `Task` s may have the same description and time frame. The `Task` component uses Singapore's standard date format (`dd/MM/yyyy`).

Marking a task as done

You can mark the task as done in the module in Mod Manager.

NOTE A newly added task as above will be considered as not done by default.

NOTE Editing a task will not change the done/not done status of the task.

NOTE Tasks that are already marked as done cannot be re-marked as done.

Format:

- `task done /code MOD_CODE /id ID_NUMBER`

Command properties:

- MOD_CODE should belong to a valid and existing module in Mod Manager.
- ID_NUMBER should belong to a valid task for the module above.

Example:

You can mark a task as done in the module. To mark the task with task ID `ID_NUMBER` in module `MOD_CODE` to be done, you can type in the following command:

`task done /code CS2105 /id 224` and hit Enter

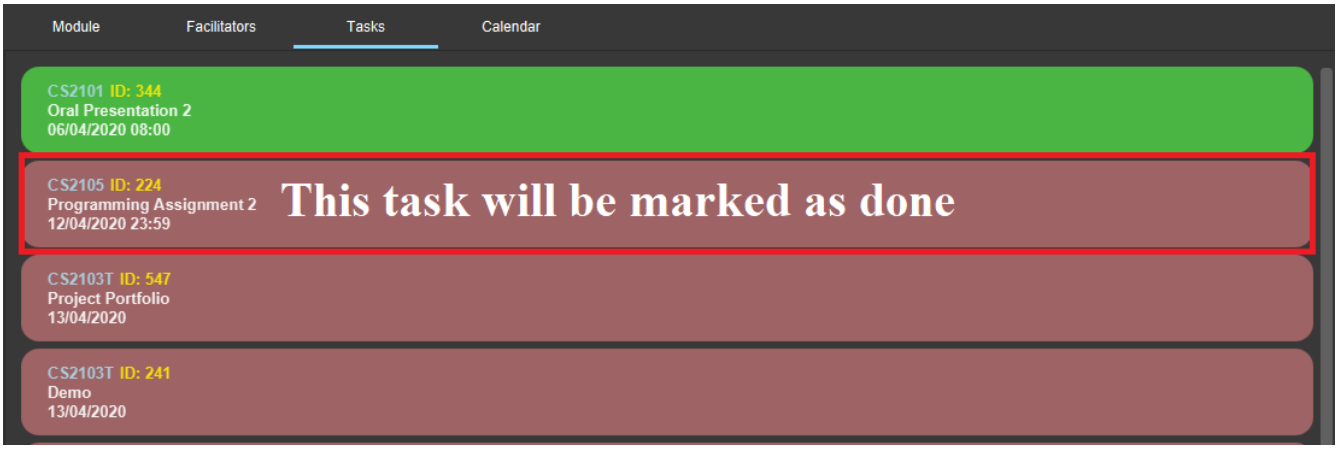

Figure 1. Before `task done /code CS2105 /id 224`

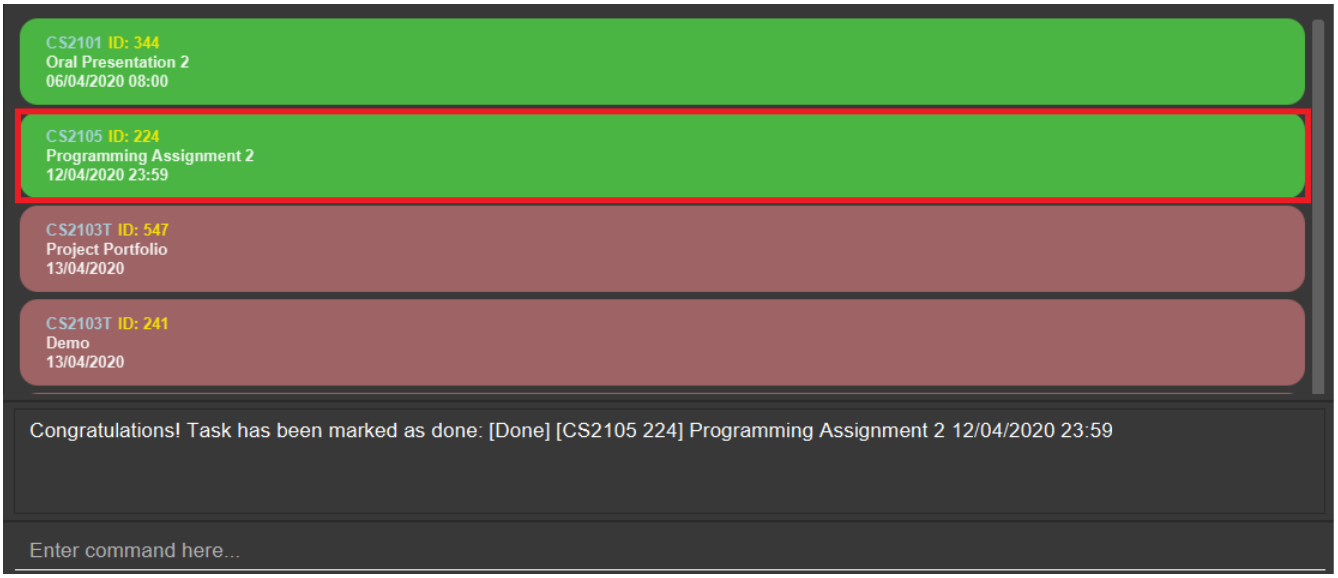

Figure 2. After `task done /code CS2105 /id 224`

The task card has changed to green; which means our task has been marked as done. Hooray! We just completed a task.

Viewing all tasks across all modules in ModManger

You can view a list of all tasks across all modules in Mod Manager. This is great when you need an overview of all tasks that you need to complete at present.

Format:

- `task list`

Example:

By typing the command above, you should see the following:

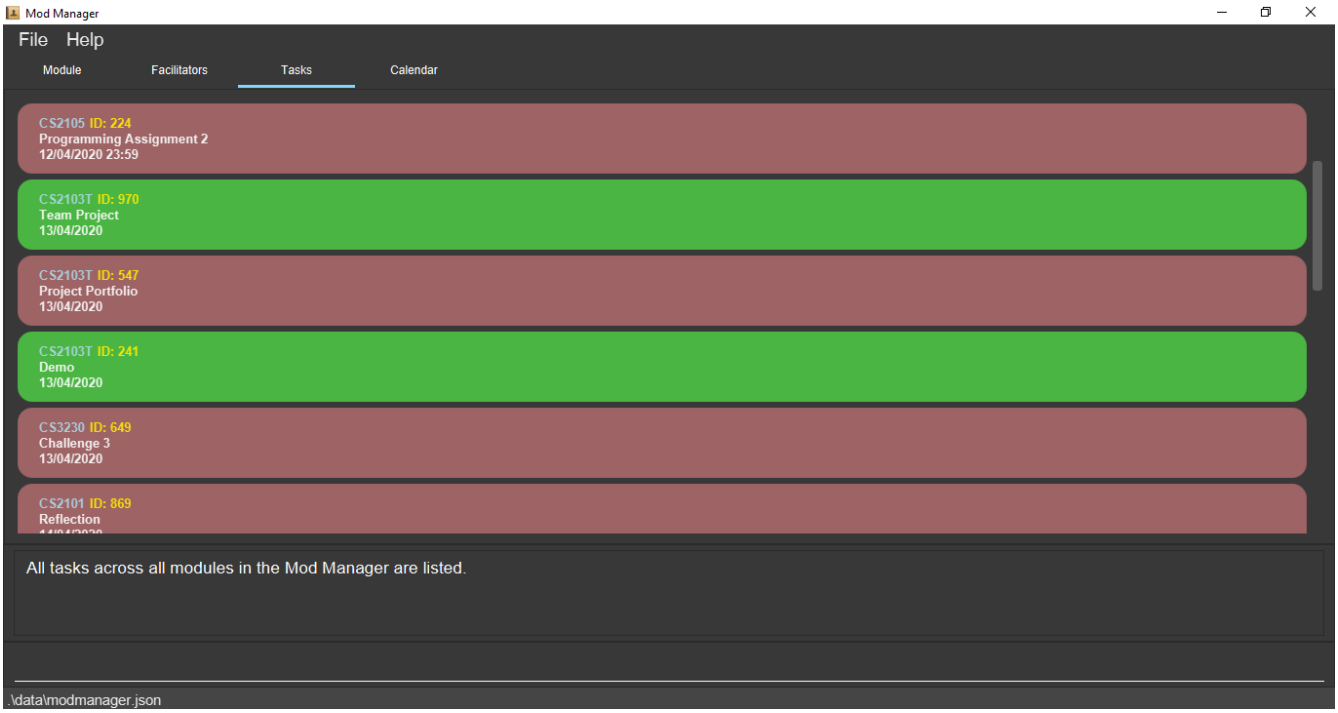

Figure 3. `task list` *shows a list of all tasks in Mod Manager*

Viewing tasks for a specific module in ModManger

If you want to find tasks for a specific module in Mod Manager, this is the command for you to use!

Format:

- `task module /code CS2103T`

Command properties:

- MOD_CODE should belong to a valid and existing module in Mod Manager.

NOTE	Alternatively, you can also view the tasks for a specific module in the Module tab (main dashboard).

Example:

If you want a list of current tasks for the module `CS3230`, you can type in the following command:

`task module /code CS3230` and hit Enter

It is not compulsory for you to be at the Tasks tab before typing in this command. Mod Manager will automatically redirect you to the Tasks tab if you are currently at another tab.

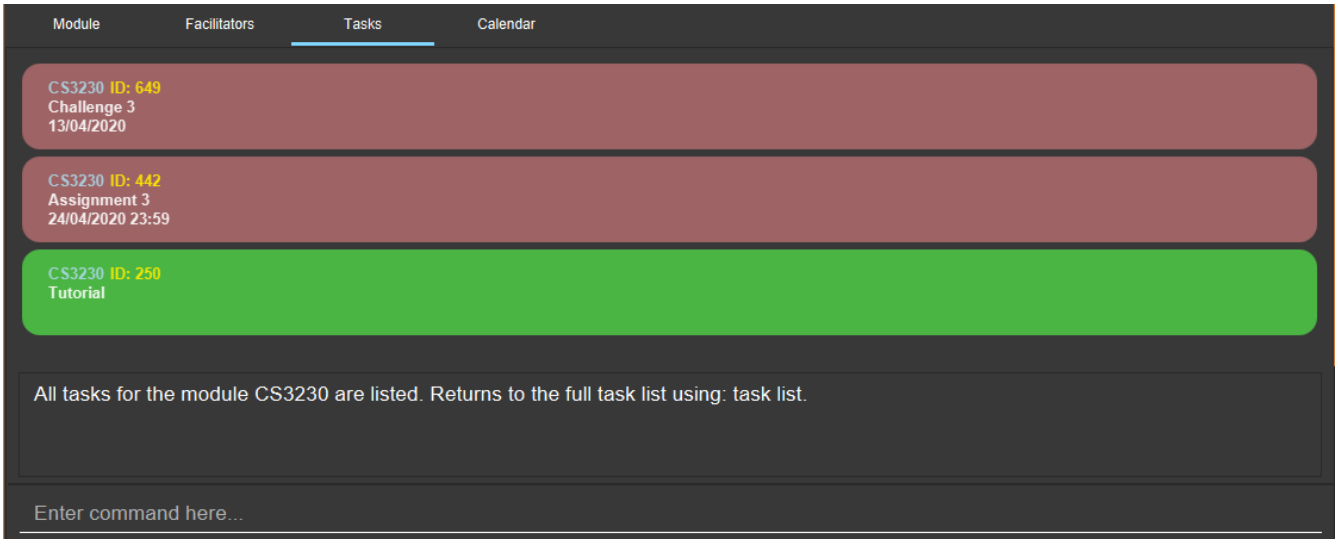

Figure 4. All tasks belonging to the module `CS3230` are listed

Viewing all tasks not done/finished

Previously, we know that we can mark a task as done, so as to organise, manage, and plan our tasks better. Now, with this command, you can see all the tasks that have not yet been finished.

Format: `task undone`

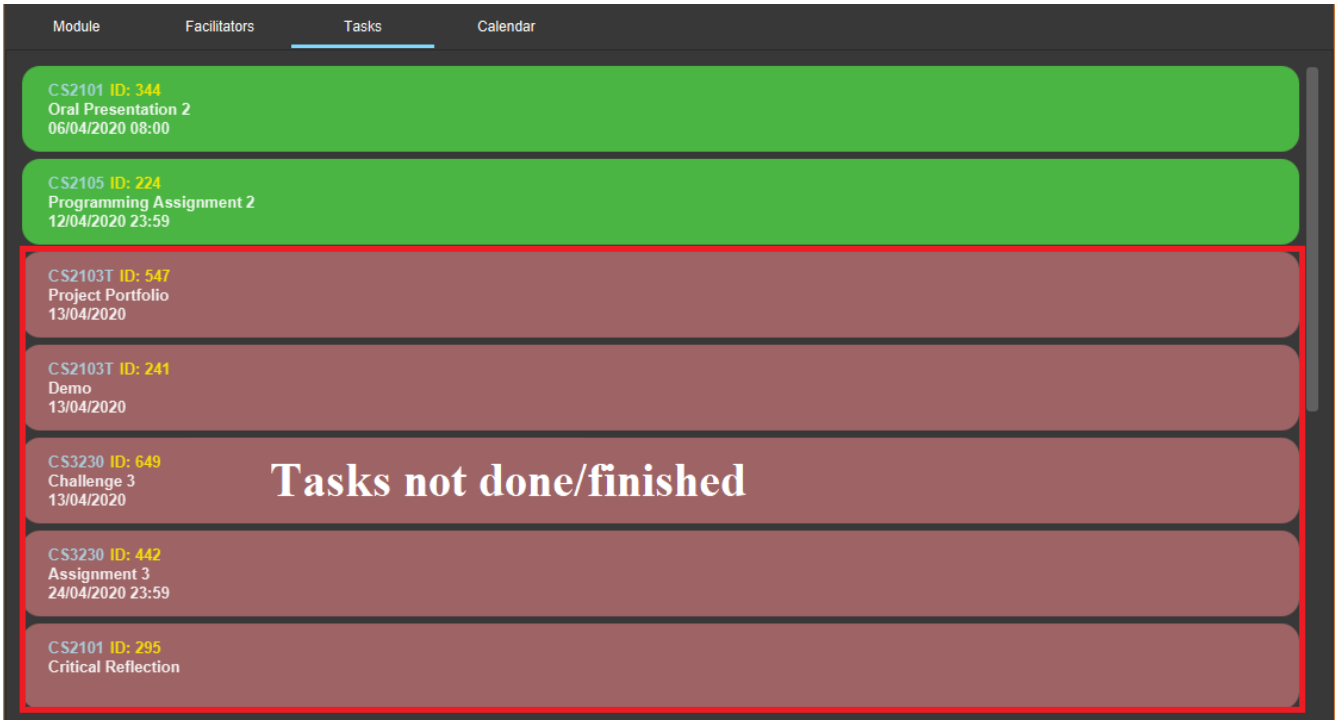

Figure 5. Before `task undone`, all tasks are listed

By typing the command above and hit Enter, you should only see uncompleted tasks, which are in dark red color:

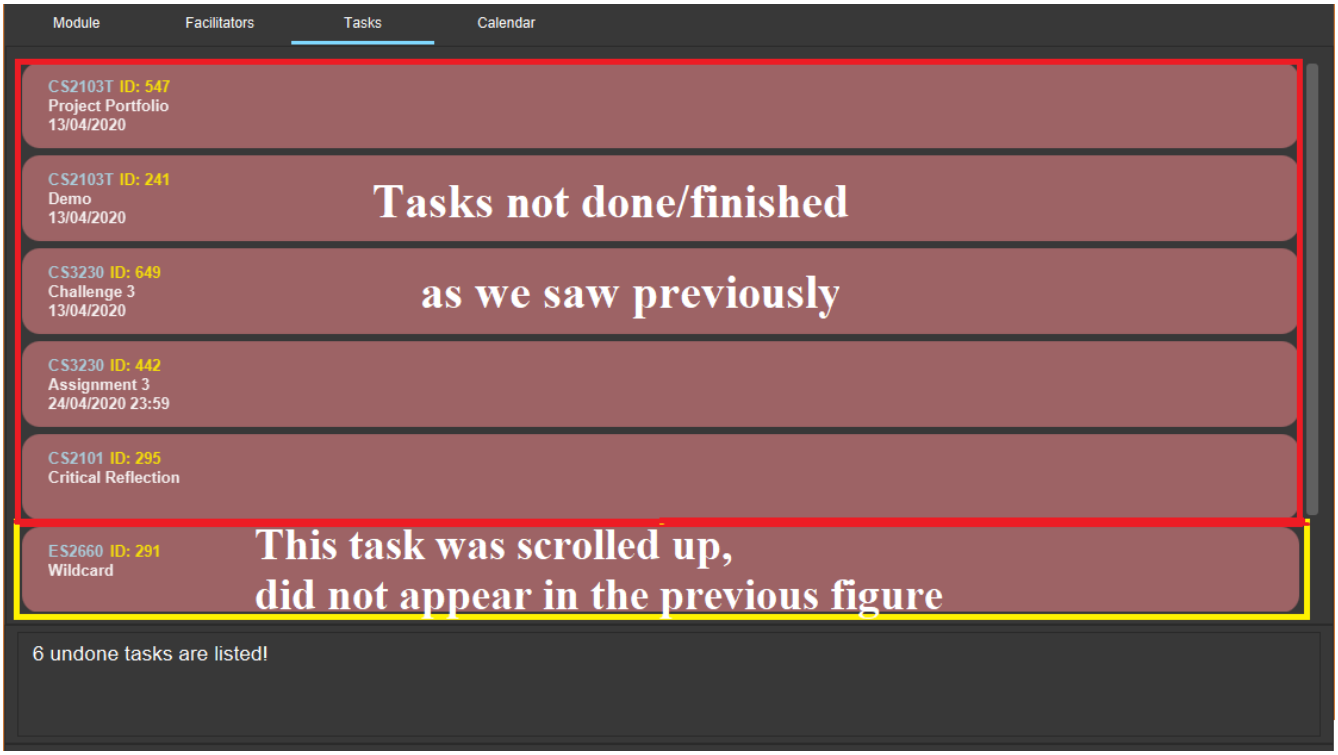

Figure 6. Only uncompleted tasks are shown

Finding tasks by description

You are browsing through the task list. But there are too many tasks! You suddenly remember a specific task that you want to do, but you can only vaguely remember its description, e.g. something related to assignment.

This command is exactly what you need. In your case, you can find all tasks that contain the word `assignment`, which may include `Programming Assignment`, `written assignment`, `Take-home Lab Assignment` (note that it can be case-insensitive). If you remember multiple words in your wanted tasks, you may also type in multiple words as you want. Tasks that meet at least one of the keywords you provided will be shown to you.

Format:

- `task find DESCRIPTION [MORE_DESCRIPTIONS]…`

Command properties:

- The `find` works across modules, so no `/code` command are required. For example, you may want to find all the `assignment` currently due.
- Searching for description is case insensitive. e.g `programming` will match `Programming`.
- The order of the descriptions does not matter. e.g. `Programming Assignment` will match `Assignment Programming`.
- Tasks are only searched in the description.
- Words can be partially matched e.g. `assign` will match `assignment`.
- Tasks matching at least one description will be returned (i.e. `OR` search). e.g. `assign home` will return both `Programming Assignment 2` and `Homework 3`.

Examples:

To find tasks that contain the word `oral`, `assign`, or `tut` in their description, you can type in the following command:

`task find oral assign tut` and press Enter

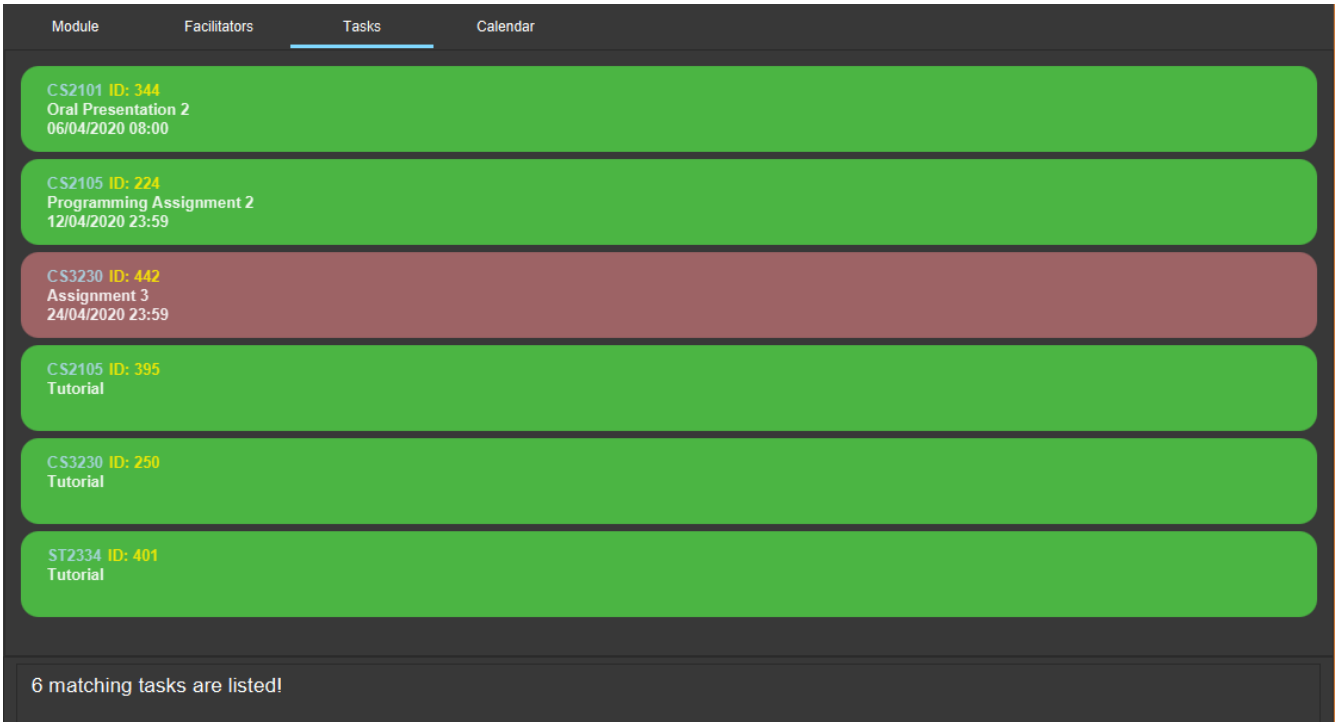

Figure 7. After `task find oral assign tut`*, all matching tasks are displayed.*

Explanation:
- `Oral Presentation 2` contains `Oral` which matches `oral` (case-insensitive).
- `Programming Assignment 2` contains `Assignment` which matches `assign` (case-insensitive, and words can be partial match)
- Similarly, `Assignment 3` will match `assign`, and `Tutorial` will match `tut`
- As long as a task's description matches **one** of the keywords provided, it will be shown.

You can try typing in `task find assign tut oral` and press Enter. This will return the same list of tasks, since the ordering of the keywords does not matter.

Other examples:

- `task find homework`
  Finds all tasks that contain the word `homework` in their description
- `task find math coding`
  Finds all tasks that contain the word `math` **or** `coding` in their description

Searching tasks by date

With this command, you can search for all tasks that occur on your specified date, month, or year.

NOTE	Tasks are only searched for its date. Tasks that do not have dates or times will not be found in this list.

Format:

- `task search [/date DATE] [/month MONTH] [/year YEAR]`

Command properties:

- The `search` works across modules, so no `/code` commands are required.
- If no optional fields are provided, Mod Manager will output all tasks that have a specified time period.
- Invalid inputs such as `/date monday`, `/month December`, `/year this year` are not allowed. Please use numbers for `/date`, `/month`, and `/year` instead.
- Invalid date, month, or year is not allowed. For example:
  - `/date 32`, `/date 0`: `date` can only range from `1` to `31`.
  - `/month 13`, `/month 0`: `month` can only range from `1` to `12`.
  - `/year 0`, `/year 99999`: the `search` only accept `year` ranging from `1` to `9999`
  - `/date 30 /month 2`: there is no 30/2 in any year
  - `/date 29 /month 2 /year 2019`: this is not possible since `2019` is not a leap year. However, `/date 29 /month 2` (year is not provided) is okay.
- Tasks matching **all** conditions will be returned (i.e. `AND` search). e.g. `/month 5 /year 2020` will only match tasks that are in May 2020.

Example:

You can search for tasks that are due on the submission date of CS2103T for AY19/20 S2 (13 April). To find tasks happening on 13 April, you can type `task search /date 13 /month 4` and press Enter. This will return all tasks that are happening on 13 April.

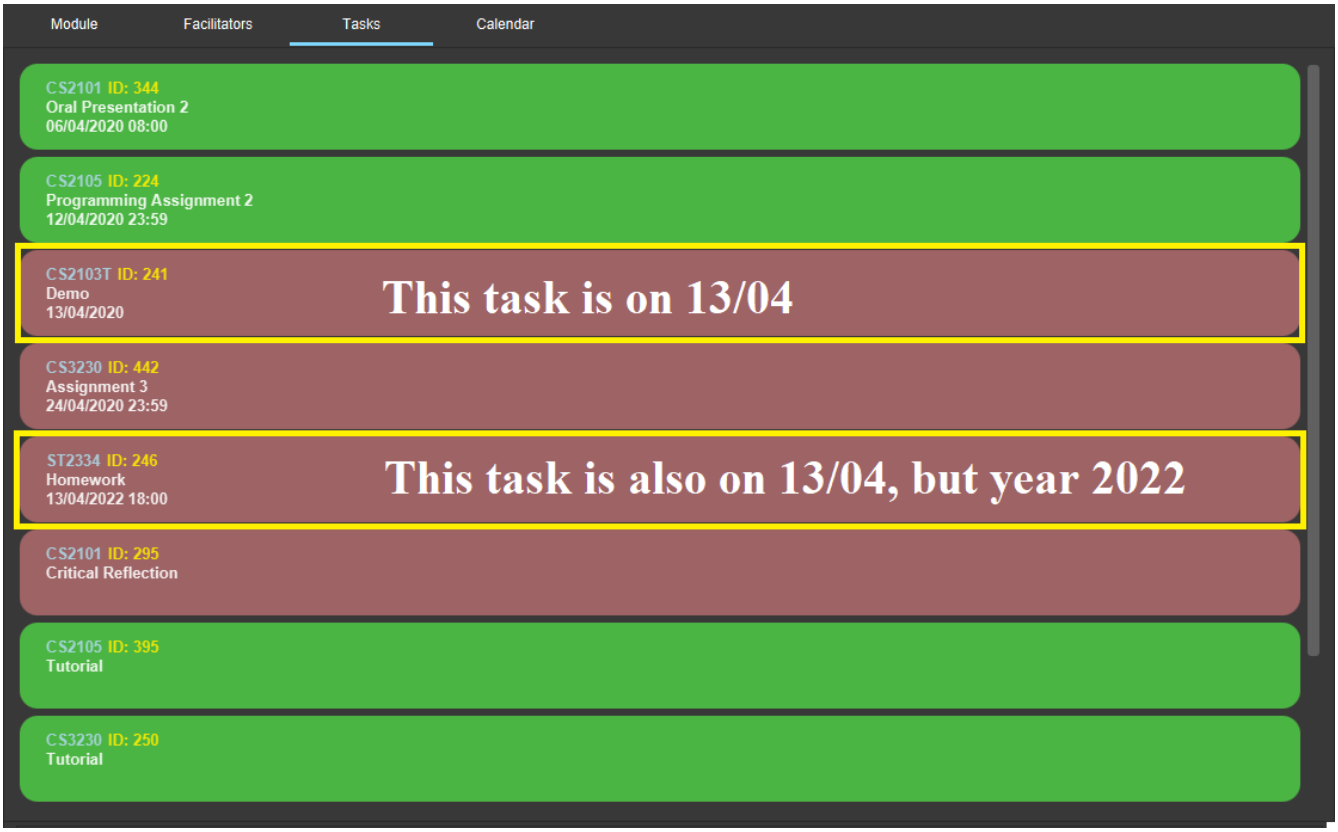

Figure 8. Before `task search /date 13 /month 4`*, all tasks are listed*

Note that the content above may be different from what is currently on your Mod Manager. You may `add` or `edit` the tasks to match we have above

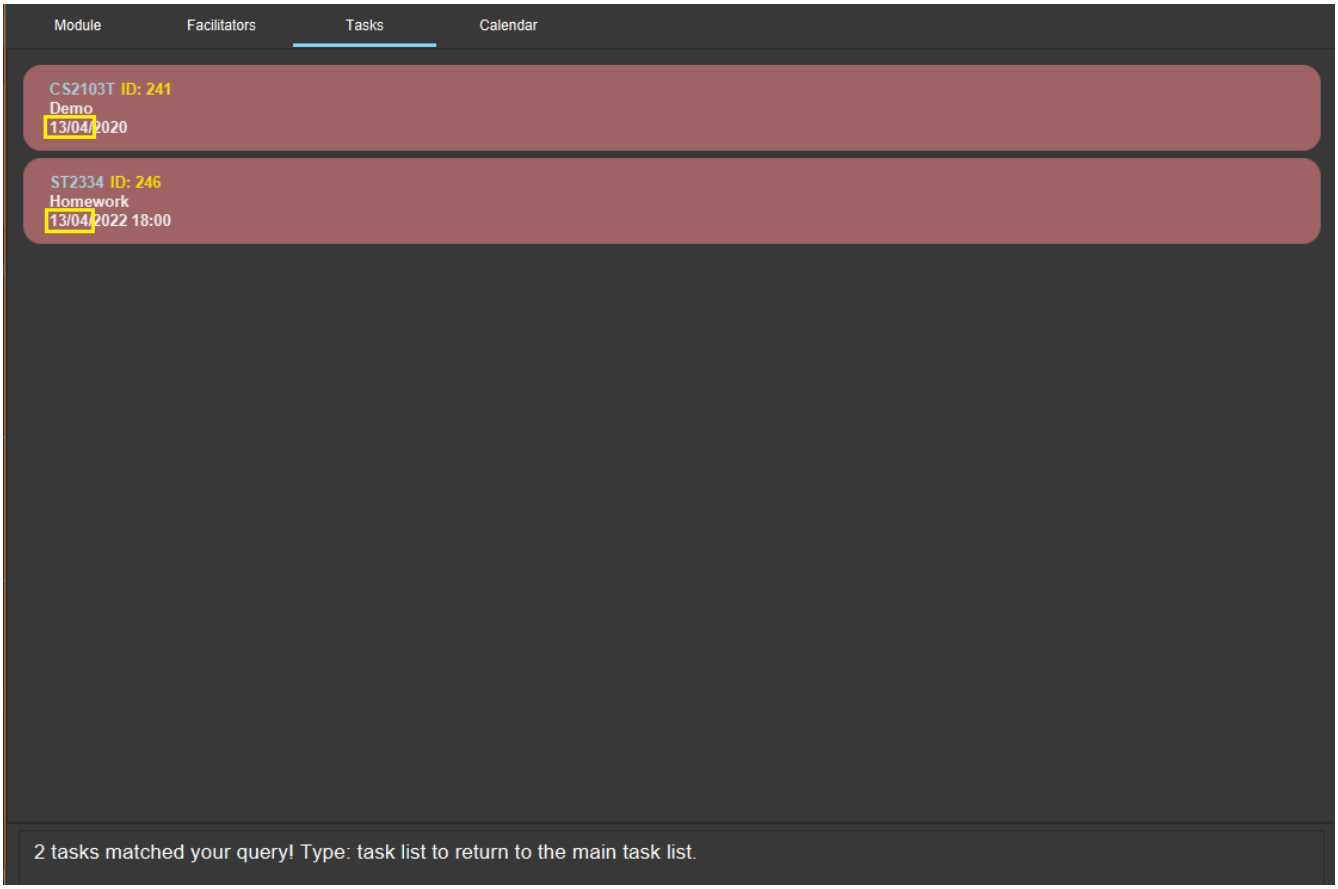

Figure 9. After `task search /date 13 /month 4`*, only matching tasks are listed*

Explanation: the two tasks both have date as `13` and month as `4`.

Other examples:

- `task search /date 1`
  Searches for all tasks happening on the first day of the month, in any year. Who wants to study on New Year's Day really?
- `task search /month 4 /year 2020`
  Searches for all tasks in the current month (at the time of writing, April 2020).
- `task search /year 2020`
  Searches for all tasks in this year (at the time of writing). This will be useful if Mod Manager is used over a long period of time.
- `task search /date 14 /month 2 /year 2021`
  Searches for all tasks happening on 14/02/2021.

Finding upcoming tasks `[coming in v2.0]`

You can find upcoming tasks, such as assignment submission and final exam in Mod Manager.

I also contributed to some sessions of the User Guide for the entire Mod Manager.

Contributions to the Developer Guide

I idealise and design the `Task` component of **Mod Manager**. I implemented most of the commands for this `Task` component, the remaining are CRUD commands which are based on my initiated `Task` design.

Given below are sections I contributed to the Developer Guide. They showcase my ability to write technical documentation and the technical depth of my contributions to the project._

NOTE	For your easier understanding of the content below, every `Task` in a `Module` has an `ID`, which uniquely identifies the task in the module. The `ID` is important for us to differentiate a task from the others in the `Module`, since two `Task` s may have the same description and time frame.

Implementation, Task Component

Marking a task as done

The marking a task as done command allows users to mark a certain `Task` in a `Module` as done, based on its task ID called `taskNum`. This feature is facilitated by `TaskCommandParser`, `TaskMarkAsDoneCommandParser` and `TaskMarkAsDoneCommand`. The operation is exposed in the `Model` interface as `Model#setTask()`.

Given below is an example usage scenario and how the marking task as done mechanism behaves at each step.

1. The user executes the task mark as done command and provides the `moduleCode` and the `taskNum` of the task to be marked as done.
2. `TaskMarkAsDoneCommandParser` creates a new `TaskMarkAsDoneCommand` based on the `moduleCode` and `taskNum`.
3. `LogicManager` executes the `TaskMarkAsDoneCommand`.
4. `TaskMarkAsDoneCommand` retrieves the `moduleCode` and `taskNum` of the task to be marked as done, and then retrieves the current existing `Task` from `ModManager`.
5. `TaskMarkAsDoneCommand` creates a clone of the retrieved `Task`, then mark this new `Task` as done.
6. `ModManager` sets the existing task to the new task, marked as done in the `UniqueTaskList`.
7. `ModelManager` updates the `filteredTasks` in `ModelManager`.

The following sequence diagram shows how the task mark as done command works:

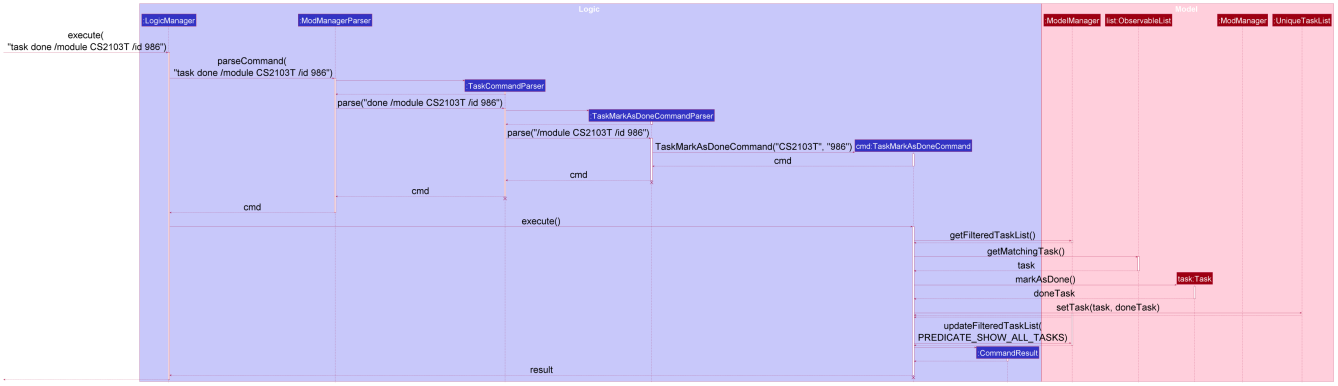

Figure 10. Sequence Diagram for `task done /module CS2103T /id 986` Command

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

The following activity diagram summarizes what happens when a user executes the task mark as done command:

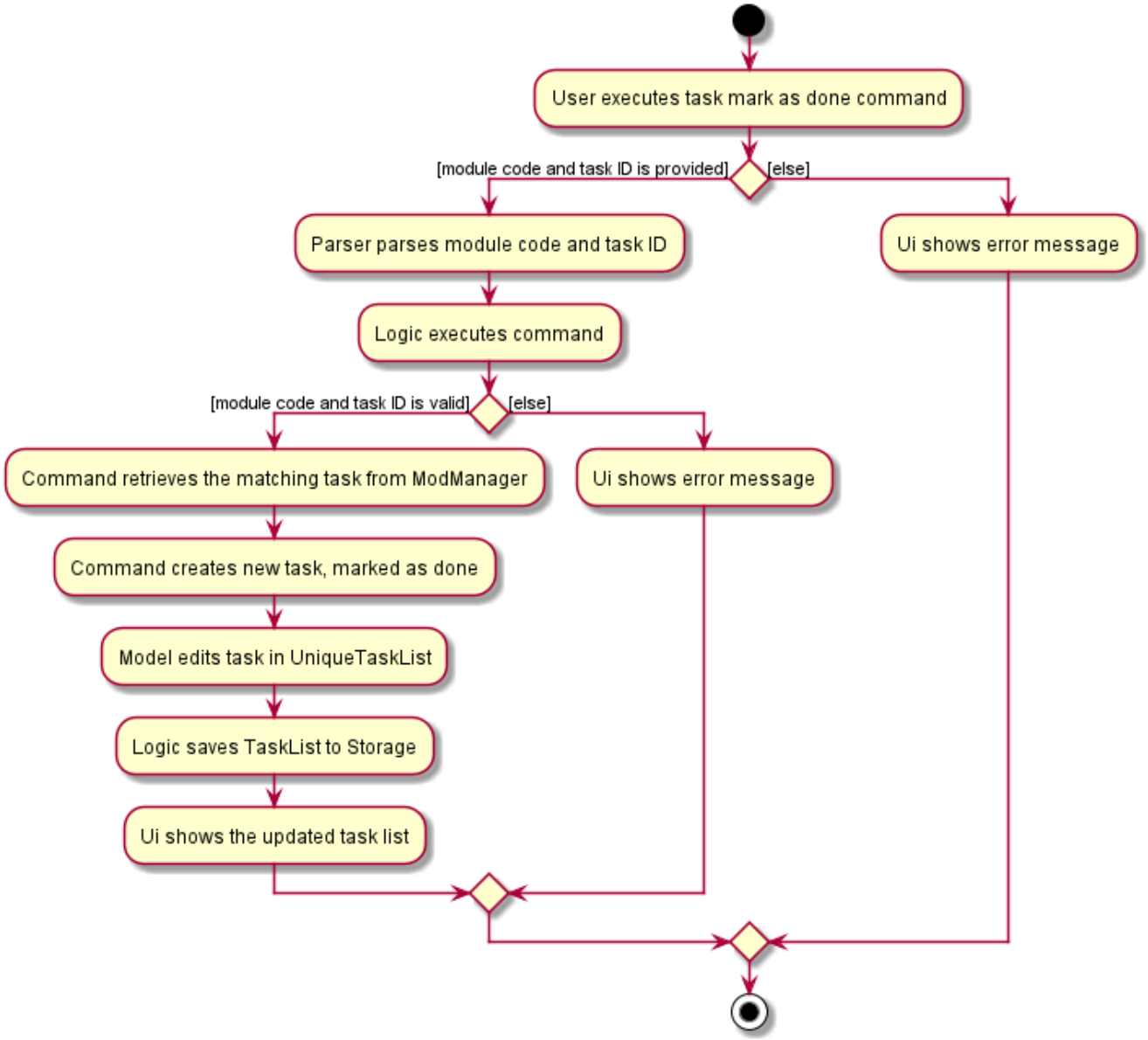

Figure 11. Activity Diagram for a general `task done` Command

Viewing all tasks across modules in Mod Manager

The list task feature allows users to list all tasks across all modules in Mod Manager. This feature is facilitated by `TaskCommandParser` and `TaskListCommand`. The operation is exposed in the `Model` interface as `Model#updateFilteredTaskList()`.

Given below is an example usage scenario and how the task list mechanism behaves at each step:

1. The user executes the task list command.
2. `TaskCommandParser` creates a new `TaskListCommand`.
3. `LogicManager` executes the `TaskListCommand`.
4. `ModelManager` updates the `filteredTasks` in `ModelManager`.

The following sequence diagram shows how the task list command works:

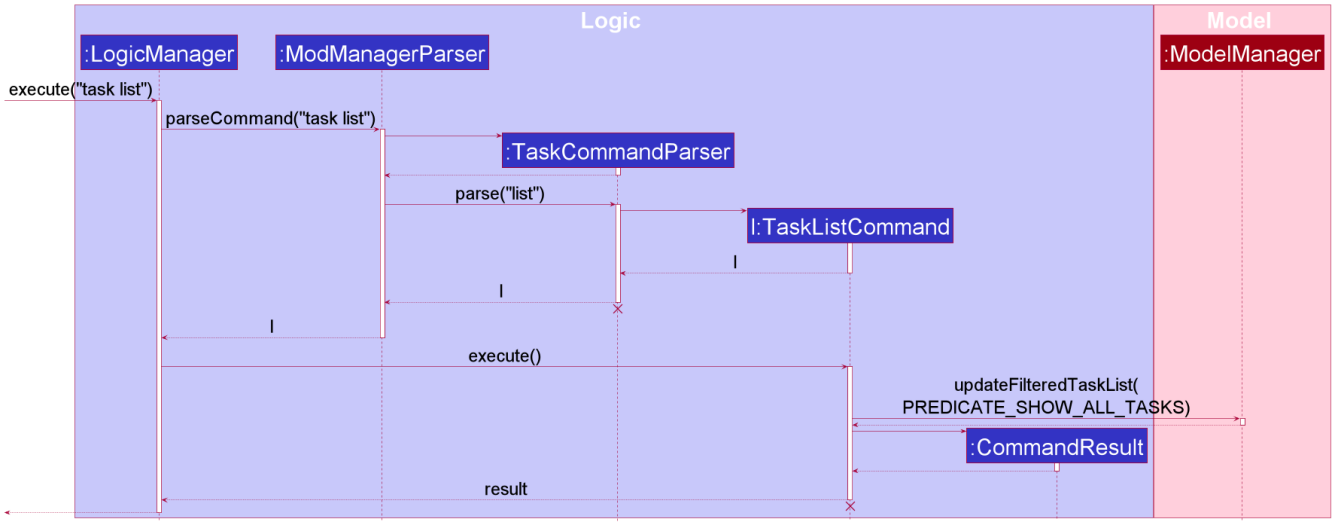

Figure 12. Sequence Diagram for `task list` Command

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

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

Figure 13. Activity Diagram for `task list` Command

Viewing tasks for a specific module in ModManger

The viewing task by module feature allows users to find all tasks belonging to a specific module in Mod Manager. This feature is facilitated by `TaskCommandParser`, `TaskForOneModuleCommandParser` and `TaskForOneModuleCommand`. The operation is exposed in the `Model` interface as `Model#updateFilteredTaskList()`.

Given below is an example usage scenario and how the task search mechanism behaves at each step:

1. The user executes the task search command and provides the day, month, or year, or any combination of which that they want to search for.
2. `TaskSearchCommandParser` creates a new `TaskSearchCommand` based on the names.
3. `LogicManager` executes the `TaskSearchCommand`.
4. `ModelManager` updates the `filteredTasks` in `ModelManager`.

The following sequence diagram shows how the search tasks for a specific module command works:

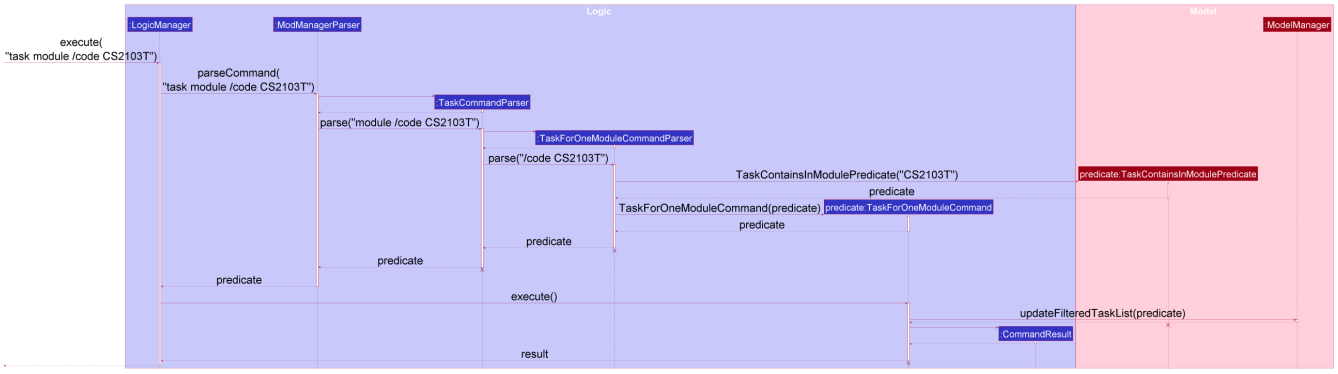

Figure 14. Sequence Diagram for `task module /code CS2103T` Command

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

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

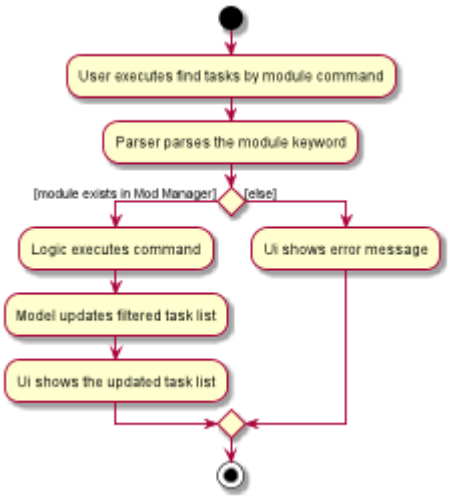

Figure 15. Activity Diagram for a general `task module` Command

Viewing undone tasks

The viewing undone tasks only feature allows users to view only tasks that are not yet completed in their `Tasks` tab. This feature is facilitated by `TaskCommandParser`, `TaskListUndoneCommandParser` and `TaskListUndoneCommand`. The operation is exposed in the `Model` interface as `Model#updateFilteredTaskList()`.

Given below is an example usage scenario and how the task view undone tasks mechanism behaves at each step:

1. The user executes the task view undone tasks command.
2. `TaskListUndoneCommandParser` creates a new `TaskListUndoneCommand`.
3. `LogicManager` executes the `TaskListUndoneCommand`.

4. `ModelManager` updates the `filteredTasks` in `ModelManager`.

The following sequence diagram shows how the task view undone tasks command works:

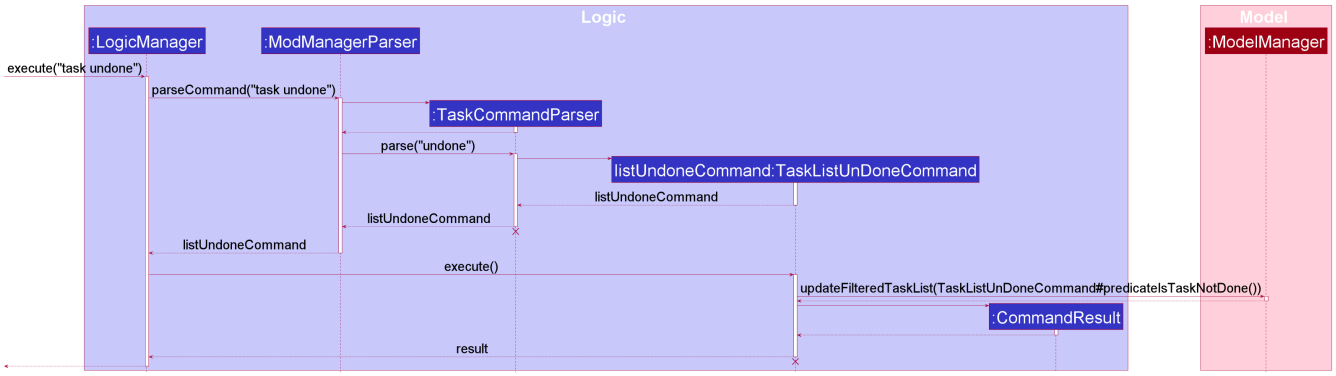

Figure 16. Sequence Diagram for `task undone` Command

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

The following activity diagram summarizes what happens when a user executes a task view undone tasks only command:

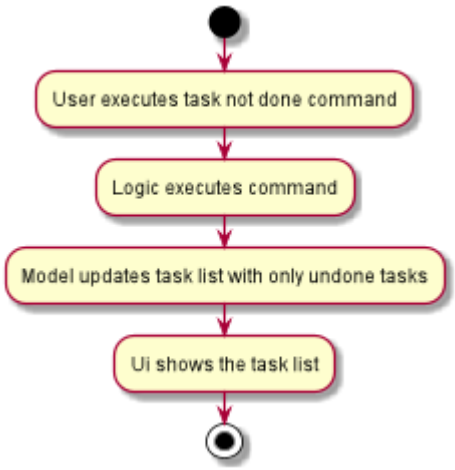

Figure 17. Activity Diagram for `task undone` Command

Finding tasks by description

The find task feature allows users to find a task by its description in Mod Manager. This feature is facilitated by `TaskCommandParser`, `TaskFindCommandParser` and `TaskFindCommand`. The operation is exposed in the `Model` interface as `Model#updateFilteredTaskList()`.

Given below is an example usage scenario and how the task find mechanism behaves at each step:

1. The user executes the task find command and provides the descriptions of the tasks to search for.
2. `TaskFindCommandParser` creates a new `TaskFindCommand` based on the descriptions.
3. `LogicManager` executes the `TaskFindCommand`.

4. `ModelManager` updates the `filteredTasks` in `ModelManager`.

The following sequence diagram shows how the task find command works:

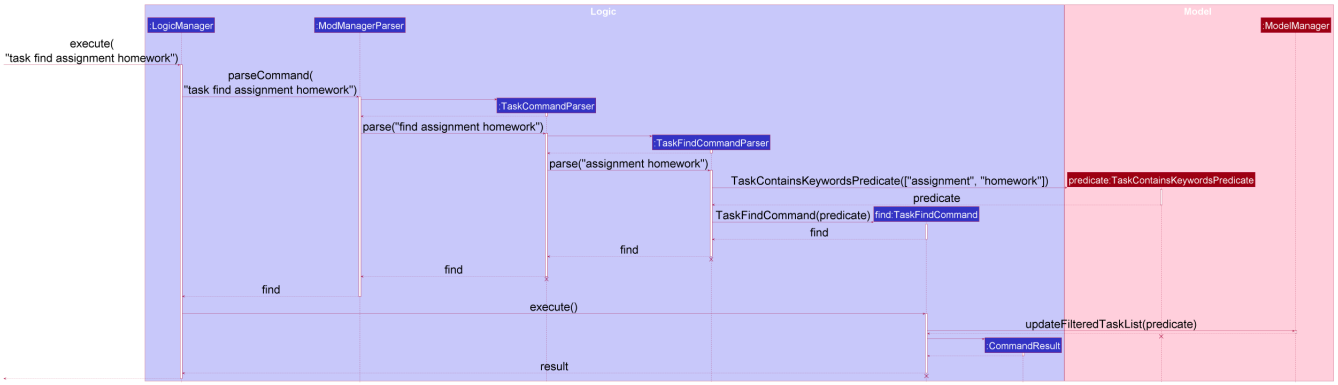

Figure 18. Sequence Diagram for `task find assignment homework` Command

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

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

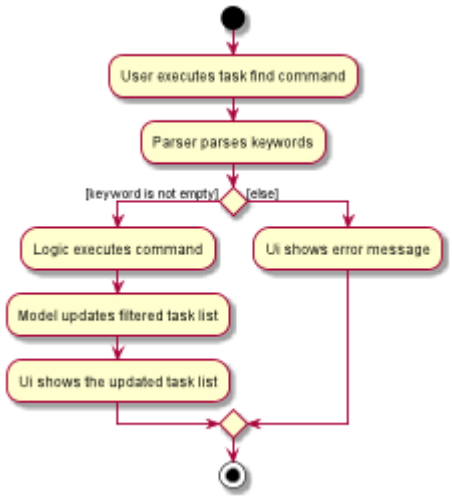

Figure 19. Activity Diagram for a general `task find` Command

Searching tasks by date

The search task feature allows users to search all tasks that occur on the specified date, month, or year. This feature is facilitated by `TaskCommandParser`, `TaskSearchCommandParser` and `TaskSearchCommand`. The operation is exposed in the `Model` interface as `Model#updateFilteredTaskList()`.

Given below is an example usage scenario and how the task search mechanism behaves at each step:

1. The user executes the task search command and provides the day, month, or year, or any combination of which that they want to search for.

2. `TaskSearchCommandParser` creates a new `TaskSearchCommand` based on the names.
3. `LogicManager` executes the `TaskSearchCommand`.
4. `ModelManager` updates the `filteredTasks` in `ModelManager`.

The following sequence diagram shows how the task search command works:

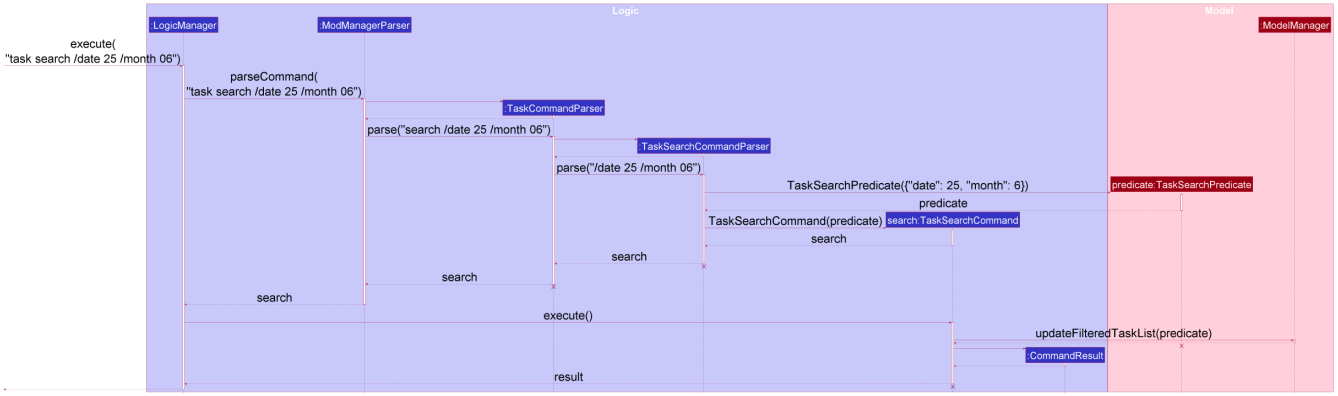

Figure 20. Sequence Diagram for `task search /date 25 /month 6` Command

> **NOTE** The lifeline for `TaskCommandParser`, `TaskSearchCommandParser`, `TaskSearchCommand` and `TaskSearchPredicate` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of the diagram.

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

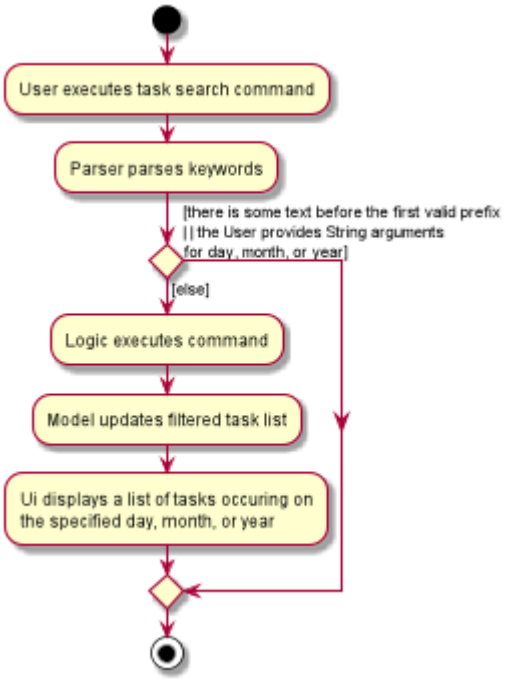

Figure 21. Activity Diagram for a general `task search` Command

Searching tasks by date

The search task feature allows users to search all tasks that occur on the specified date, month, or year. This feature is facilitated by `TaskCommandParser`, `TaskSearchCommandParser` and `TaskSearchCommand`. The operation is exposed in the `Model` interface as

`Model#updateFilteredTaskList()`.

Given below is an example usage scenario and how the **`task search`** mechanism behaves at each step:

1. The user executes the **`task search`** command and provides the day, month, or year, or any combination of which that they want to search for search for.
2. `TaskSearchCommandParser` creates a new `TaskSearchCommand` based on the names.
3. `LogicManager` executes the `TaskSearchCommand`.
4. `ModelManager` updates the `filteredTasks` in `ModelManager`.

The following sequence diagram shows how the **`task search`** command works:

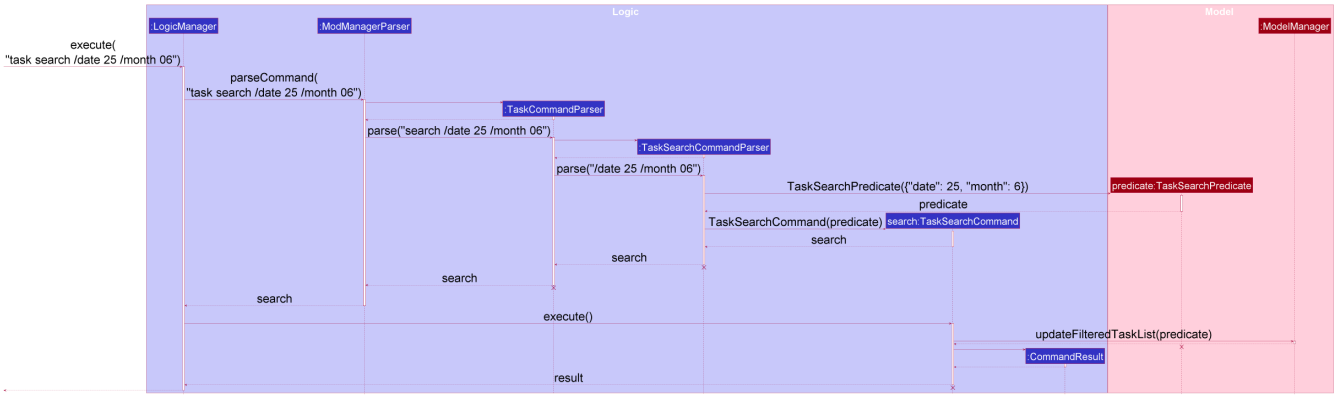

*Figure 22. Sequence Diagram for **`task search`** Command*

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

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

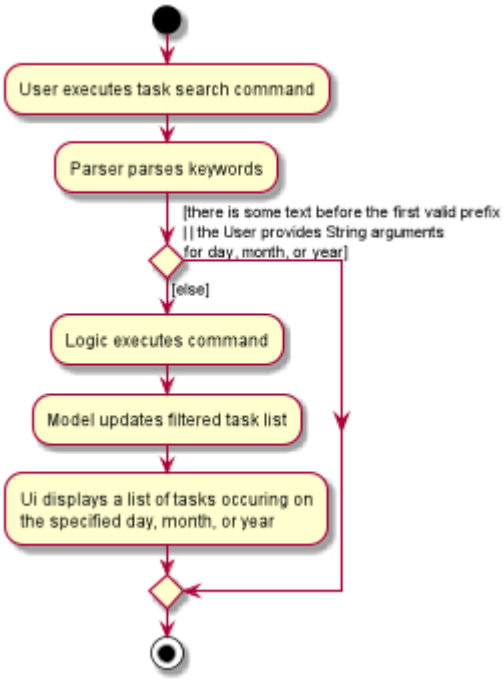

Figure 23. Activity Diagram for `task search` Command

Design Considerations

Aspect: A task may have a specified time frame, or not. How do we implement this feature?

- **Alternative 1 (current choice):** Implement `Task` as an abstract class for Mod Manager. A task with a specified time period will be created as a `ScheduledTask`, while a task with no time period specified will be created as a `NonScheduledTask`, with both `ScheduledTask` and `NonScheduledTask` are concrete subclasses of `Task`.
    - Pros: Utilises Object-Oriented Programming. Easy to implement **`search`** functionality, which we need to search for tasks that occur on a specified date, month, or year, and **`upcoming`** functionality `[coming in v2.0]`, which we need to find the upcoming tasks in Mod Manager. For these two features, we only need to work on `ScheduledTask` instances, which reduces the burden of checking for `null` `TaskDateTime` instances as the second approach below.
    - Cons: More difficulty in implementation due to time constraints. Moreover, command **`edit`** that allows us to edits the information of the task will be troublesome, when a user decides to add a time period to a `NonScheduledTask`. In this case, we have to re-create a new `ScheduledTask` with the same description and its time provided. If we need to maintain a `List<ScheduledTask>` or `List<Task>` somewhere in the code, for example, in our `Module` instance, we also have to update the list contents in our `Module` s too. This requires the association between `Module` and `Task` to be bi-directional, which increases coupling and make it harder for us to maintain and conduct tests. There is also extra overhead time communicating and collaborating with another member in our team responsible for the `Module` component, Because of these challenges, we decide to weaken the association between `Task` and `Module`, which is elaborated in our next aspect.
- **Alternative 2:** Implement `Task` as a concrete class in Mod Manager. `Task` s without a specified time period will have its time attribute `taskDateTime` set to `null`, while `Task` s with a given time period will be assign a non-null instance of `taskDateTime`.
    - Pros: Easier to implement, as we only need to create one class `Task`.

- Cons: We must handle `null` cases every time we query something about the time of a `Task`. For example, it's more challenging to implement the **`search`** and **`upcoming`** command, since we have to check whether the task has a non-null `taskDateTime` or not. Moreover, it's complex to implement the method `compareTo` of `Comparable` interface for `Task` to compare the time between tasks, when one, or both of our `taskDateTime` attributes can be `null`.

Aspect: The association between `Module` and `Task`

- **Alternative 1 (current choice):** Aggregation: Each `Task` can have an unique `ModuleCode` tag, which uniquely identifies which `Module` the task belongs to. This is a aggregation relationship, which is weaker than composition in our second approach.

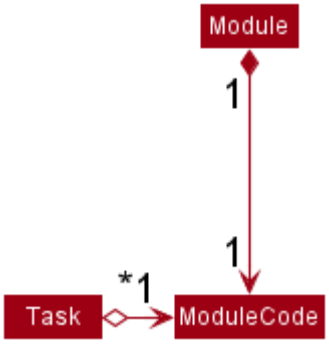

Figure 24. Class Diagram: A `Task` acts as a container for `ModuleCode` object of a `Module`. `ModuleCode` objects can survive without a `Task` object.

- Pros: Easier to implementation, and weak coupling with `Module` implementation. The `Module` need not to be aware that there are a list of `Task` s for it.
- Cons: The association between `Module` and `Task` cannot be extensive and fully descriptive as in our second approach, but this is a trade-off given the time constraints.
    - **Alternative 2:** Composition: each `Module` has a list of `Task` s corresponding to it. If the `Module` is deleted, all of the related `Task` s for the `Module` will also be removed.

Figure 25. Class Diagram: A `Module` consists of `Task` objects.

- Pros: This design choice better simulates the real-life interactions between `Module` and `Task`. For example, if we drop a `Module` in NUS, we will also drop all the `Task` s related to the `Module`, such as assignments, homework, term tests, and exams.
- Cons: Difficulty in implementation due to time constraints, as well as strong content and data coupling. More overhead in communicating and collaborating with the team member responsible for the `Module` component, as mentioned above.

Appendix C: Use Cases

Use case: UC14 - Marking a task as done

MSS

1. User requests to mark a task as done and provides the module code and task ID of the task.
2. Mod Manager marks the task as done. The corresponding task card is changed to green.

Use case ends.

Extensions

1a. The module code and task ID provided is invalid.

1a1. Mod Manager shows an error message.

Use case resumes from step 1.

1b. The task is already marked as done.

1b1. Mod Manager shows an error message, notifying the task is already done.

Use case resumes from step 1.

Use case: UC15 - Viewing all tasks across modules in Mod Manager

MSS

1. User requests to list all tasks across modules in Mod Manager.
2. Mod Manager shows the list of all the tasks.

Use case ends.

Extensions

1a. There are no tasks currently available in Mod Manager.

Use case ends.

Use case: UC16 - Viewing tasks for a specific module in Mod Manger

MSS

1. User requests to list tasks for a specific module and provides the module code.

2. Mod Manager shows the list of tasks belonging to the specified module.

Use case ends.

Extensions

1a. The module code is invalid (module not available in Mod Manager).

1a1. Mod Manager shows an error message.

Use case resumes from step 1.

1b. There are no tasks currently available for the specified module.

Use case ends.

Use case: UC17 - Viewing undone tasks

MSS

1. User requests to list all undone tasks across modules in Mod Manager.
2. Mod Manager shows the list of all undone tasks.

Use case ends.

Extensions

1a. There are no undone tasks currently available in Mod Manager.

Use case ends.

Use case: UC18 - Finding tasks by description

MSS

1. User requests to find a task by its description and provides a number of keywords.
2. Mod Manager shows the list of tasks whose descriptions contain at least one of the keywords.

Use case ends.

Extensions

1a. None of the task descriptions contain any of the keywords.

Use case ends.

1b. No keywords are provided.

1b1. Mod Manager shows an error message.

Use case resumes from step 1.

Use case: UC19 - Searching tasks by date

MSS

1. User requests to searches for a task by its date and provides the date, month, and year, or any of which.

2. Mod Manager shows the list of tasks occurring on the specified date, month, and year, or any of which.

Use case ends.

Extensions

1a. The date, month, or year provided is invalid.

1a1. Mod Manager shows an error message.

Use case resumes from step 1.

1b. No parameters are provided.

1b1. Mod Manager shows an error message.

Use case resumes from step 1.

1c. There are no tasks matching the specified date, month, and year.

Use case ends.

PROJECT: PowerPointLabs