Thng Yao Jie - Project Portfolio

PROJECT: JelphaBot

Overview

JelphaBot is a desktop application to help NUS students manage tasks. JelphaBot allows students to track and manage tasks conveniently. Users enter commands in JelphaBot through a CLI. However, a GUI is implemented with JavaFX for a smoother user experience. It is written in Java, and has about 13 kLoC.

About this portfolio

This portfolio lists my individual contributions to the JelphaBot. It includes a summary of the feature enhancements implemented and contribution to the User Guide and Developer Guide. It also details other contributions I made throughout the duration of the project to team-based tasks.

Summary of contributions

- **Major enhancement**: added the ability for users to **group their tasks by time or module code**, as well as **provided interfaces for other developers to implement new grouping categories**.
  - What it does: allows users to change the way their tasks are listed by switching between a list that shows:
    - Tasks listed by how soon their due date is to the current date.
    - Tasks listed by their Module Code.
  - **Justification**: Task management is the core feature of JelphaBot. This feature improves the task management process significantly because it allows users to see at first glance their most urgent or important tasks.
  - **Highlights**:
    - This feature involves full-stack development in `Logic`, `Model`, `Storage` and `Ui` components.
    - As all other features use `Task` as the underlying Model entity, there was high coupling with features implemented by other groupmates.
    - This enhancement affected existing commands as `GroupedTaskList` listed showed tasks in a different order from the underlying list stored in the model, and this order would vary depending on the `Category` displayed.
    - It required model objects to be queried from the displayed index in `Ui` instead of the the default order in `Model` (which was the order in which objects were implemented).

        - The implementation was challenging as it required changes to be made to existing commands while still reducing conflict with other teammates' work. This required many design alternatives to be considered.
        - All non-User-Interface classes added were backed up with tests.
    - Relevant pull requests: #122, #161, #196, #204, #205, #212, #219, #228, #298,
- **Minor enhancement**: Updated the Ui layout for each Task to allow tasks to be distinguished by priority.
    - Pull requests: #196, #212
- **Code contributed**: [Functional code] [Test code]
- **Other contributions**:
    - Project management:
        - Managed releases `v1.1` - `v1.2` (2 releases) on GitHub.
        - Refactored `AddressBook 3` to `JelphaBot` (Pull requests: 6, #37, #24, #41, #47)
    - Enhancements to existing features:
        - Updated the code-base to use Java 8 `LocalDateTime` instead of Java 7 `DateTime`, which fixed some deprecation issues. (Pull requests: #159)
        - Updated `SampleDataUtil` to propagate an empty instance of the app with time-sensitive test data to streamline manual testing. (Pull requests: #129)
    - Documentation:
        - Updated Logic section from AB3, added diagrams and documentation for the implementation of Tab Grouping feature in Developer Guide.
        - Added documentation for Task List panel and List Command in User Guide.
    - Tools:
        - Integrated Github plugins (gh-pages, Travis, Netlify) to the team repository.

Contributions to the User Guide

Given below are sections I contributed to the User Guide. They showcase my ability to write documentation targeting end-users.

Task Management (Yao Jie)

JelphaBot allows you to track and manage your tasks comprehensively as well! You can view and sort all your tasks from the Task List page.
You can enter the `list` command or its shortcuts `:T` or `:t` to instantly switch to the task list tab. The task list panel will then display all your tasks sorted into various categories.

Format: `list`
Shortcut: `:T` or `:t`

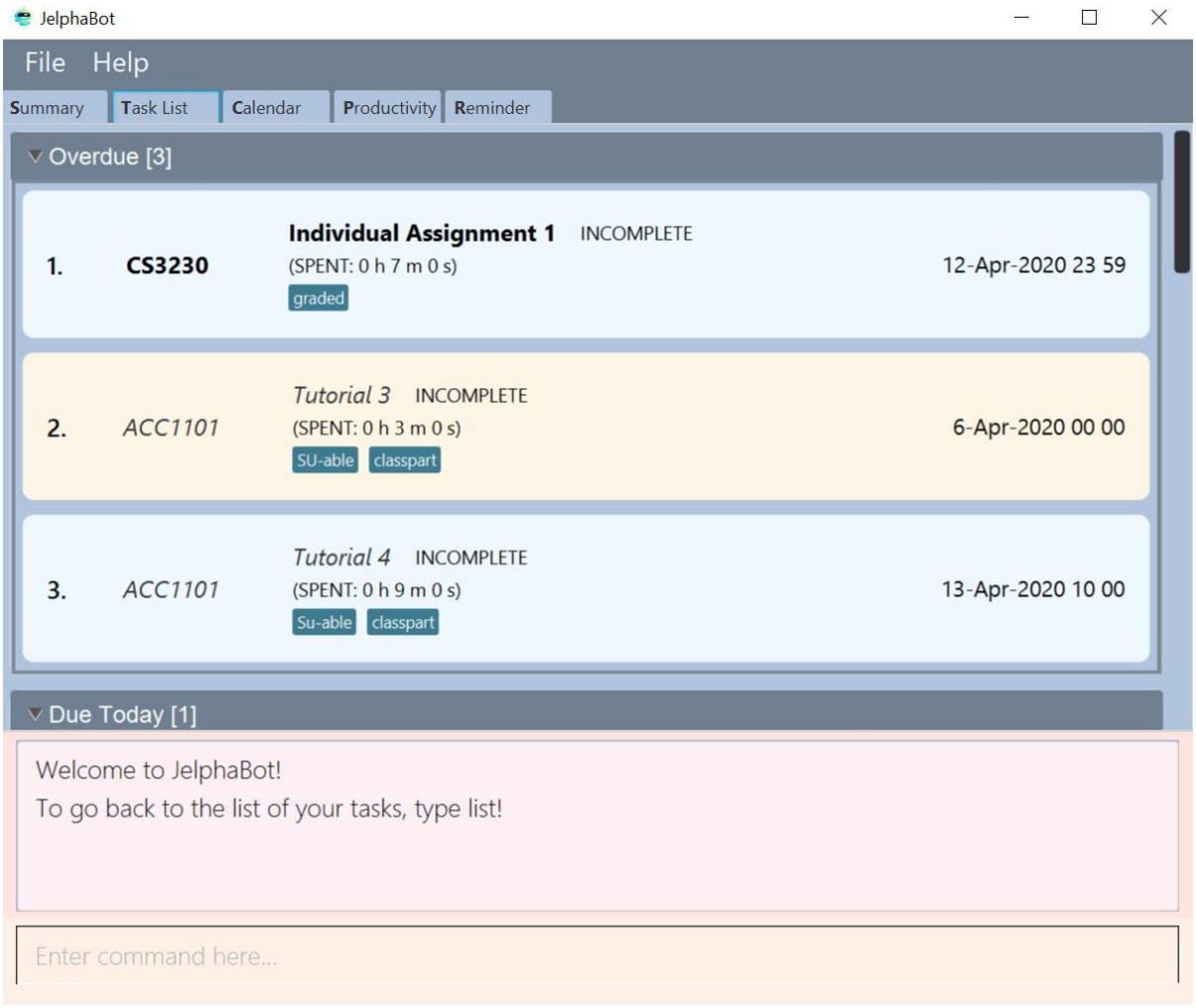

Figure 1. Example of expected result after executing `list`

Reading the Task List

The task list is formatted so that you can distinguish urgent tasks at first glance. The start of every task is labelled with a module code so that you can visually categorize them. Tasks are tagged according to their importance:

- Default priority
- **High Priority** tasks will be **bolded** to denote important tasks.
- *Low priority* tasks will be *italicized* to denote optional tasks.

The start of every task will be labelled with a module code so that you can visually categorize them. Go here to read more about adding tasks with priority and here for editing task priority.

Command Format for Task list commands

- Parts of the command in `UPPER_CASE` represent command parameters that have to be supplied by you.
  e.g. in `add d/DESCRIPTION`, `DESCRIPTION` represents a field where you can provide the appropriate description, such as `add d/Assignment 1`.
- Parameters in square brackets are optional e.g `d/DESCRIPTION [p/PRIORITY]` can be used as `d/Assignment 1 p/0` or as `d/Assignment 1`.
- Parameters with a trailing ⋯ can be used as many times as you want, or can also be omitted.
  e.g. `[t/TAG]⋯` can be used once as `t/project`, or multiple times like `t/project t/graded`, and so on.
- Parameters can be in any order e.g. if the command specifies `d/DESCRIPTION p/PRIORITY`, `p/PRIORITY d/DESCRIPTION` is also acceptable.

Listing all Tasks : `list`

JelphaBot allows you to list all your current tasks. In order to make it easier for you to view and use JelphaBot, you can group your tasks by some categories. These categories can be specified through optional arguments. Format: `list [GROUPING_CATEGORY]`

- If no `GROUPING_CATEGORY` is provided, the `date` grouping will be applied by default.
- Valid `GROUPING_CATEGORY` values are `date` (groups tasks by date) and `module` (Groups tasks by module code)

Grouping Tasks by Date : `list date`

You can group tasks based on their due date. This is also the default interface for the task list tab. Format: `list date`

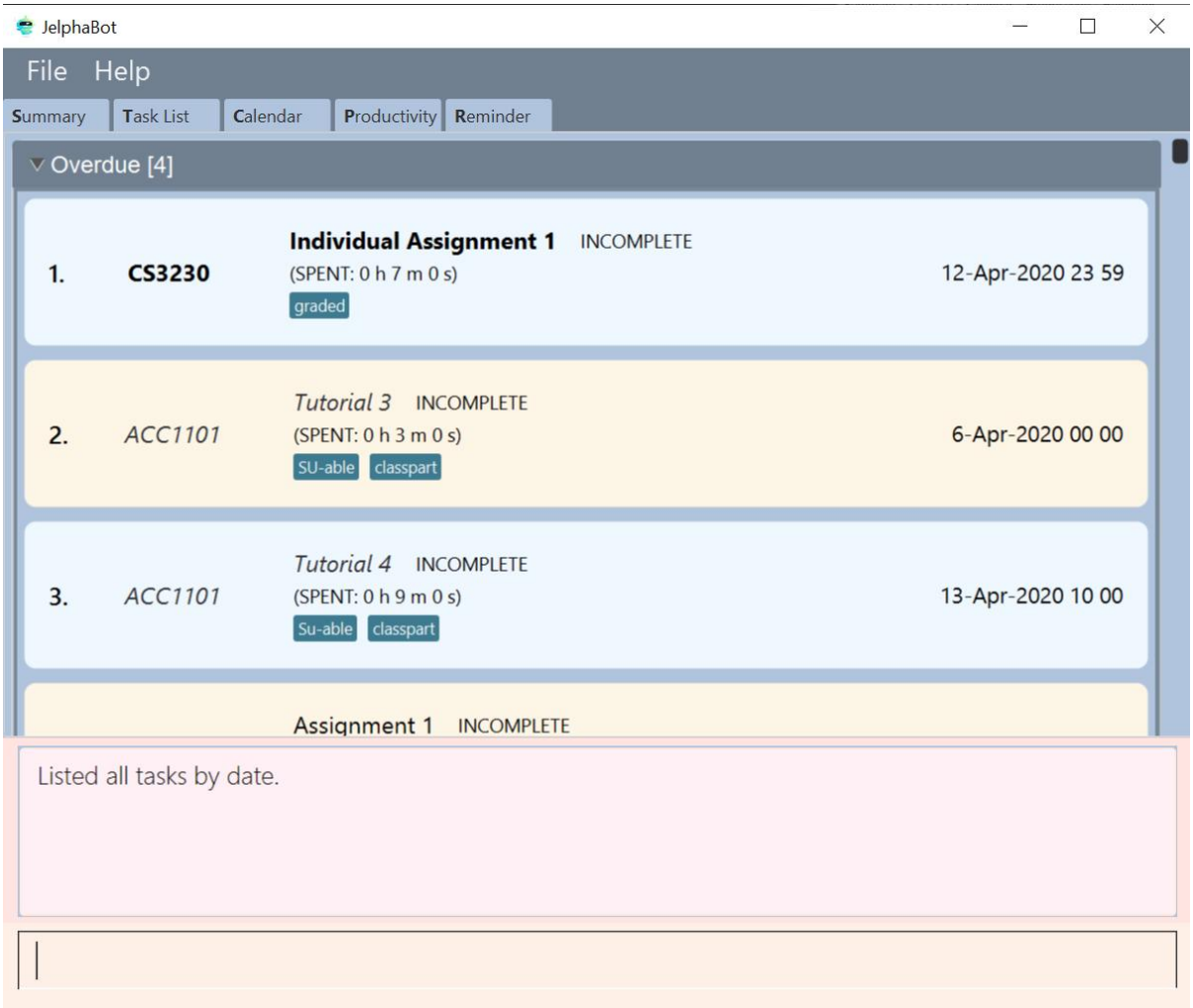

Figure 2. Example of an expected result after executing `list date`

`list date` allows you to group your tasks into the following categories:

- **Overdue**
  (Shows tasks which are past their due date)
- **Due Today**
  (Shows tasks not overdue and due by the end of the current day)
- **Due This Week**
  (Shows tasks due within the next seven days)
- **Due Someday**
  (Shows all other tasks that do not fit into prior categories)

These categories are arranged to make it easier for you to see what is immediately due. By moving tasks that are due soon to the top of the list, you can decide what to focus your time on.

Grouping Tasks by Module : `list module`

You can also group your tasks based on their module code.
Format: `list module`

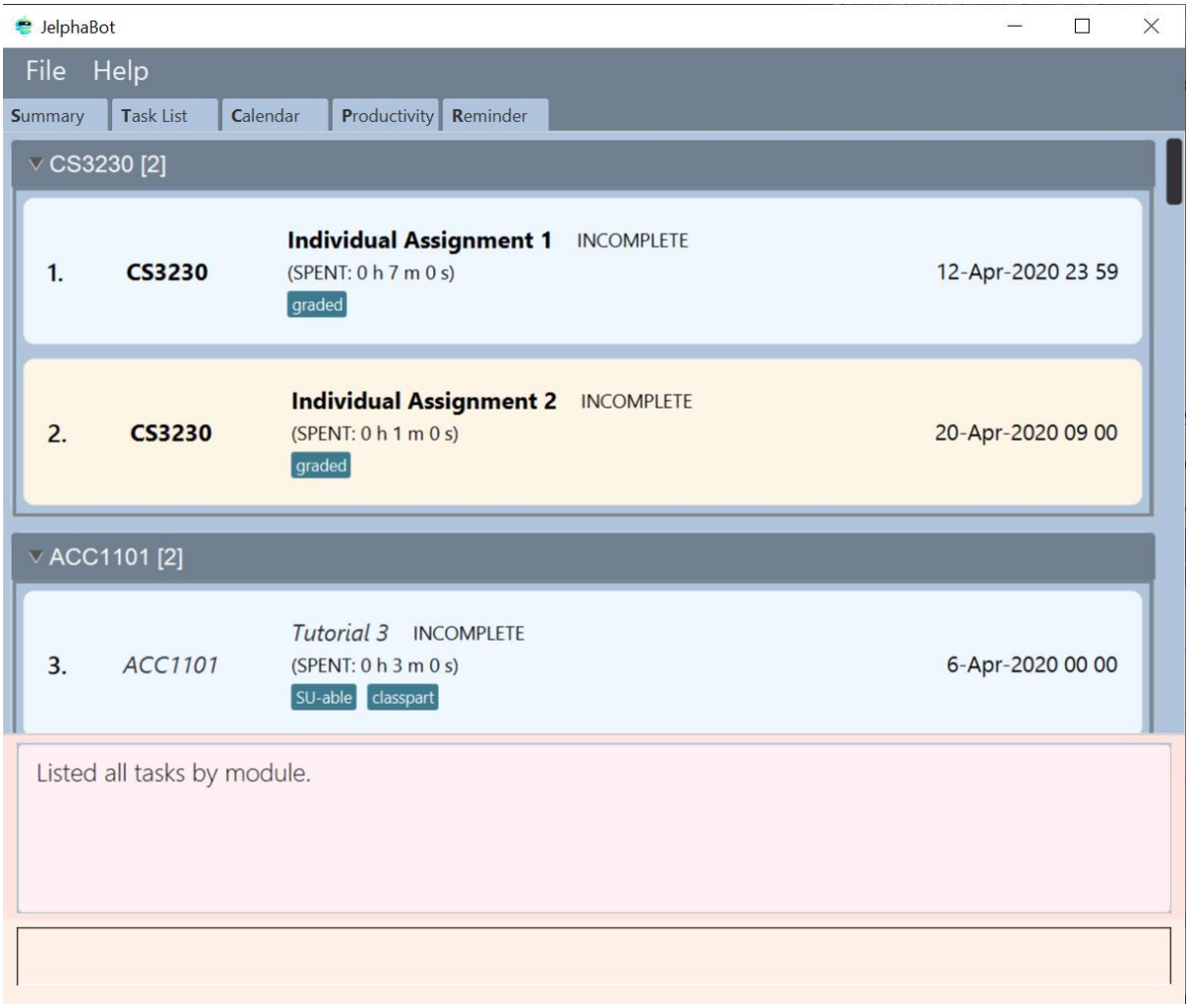

Figure 3. Example of an expected result after executing `list module`

This grouping allows you to manage your time by tracking the amount of time spent on each module. You can also see which modules have upcoming projects or assignments due.

Contributions to the Developer Guide

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.

Task Grouping feature in Task List tab (Yao Jie)

Implementation

The task category mechanism is facilitated by the `ViewTaskList` interface, which serves as a wrapper for any list of tasks.
The ViewTaskList interface supports methods that facilitate getting and iterating through the tasks contained within the list. This is to accommodate a common access for Tasks in `GroupedTaskList`, which contains multiple sub-lists.
The diagram below describes the class structure.

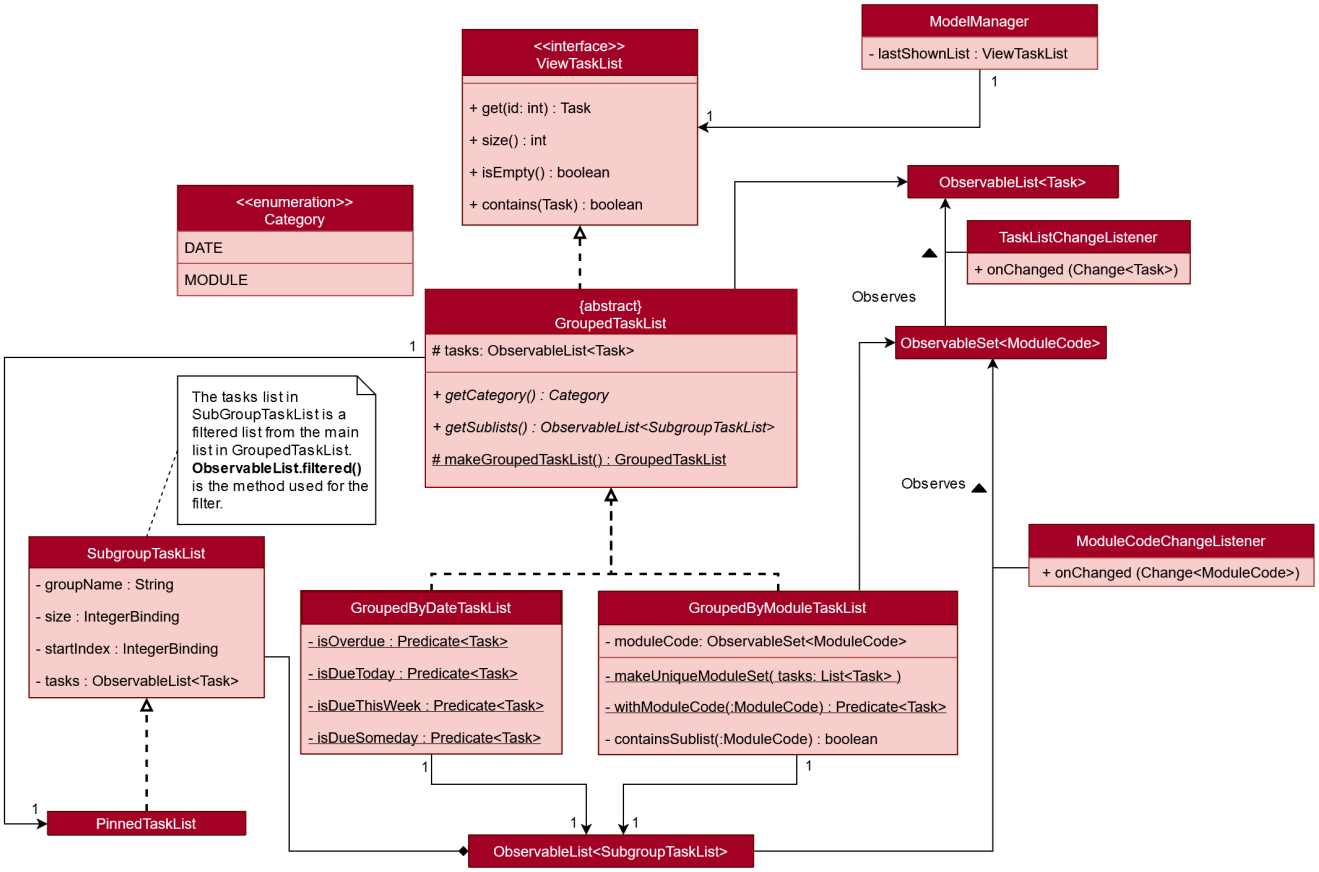

Figure 4. Class Diagram for Task List classes

Grouping tasks into sub-lists is done through the `GroupedTaskList` class.
Each `GroupedTaskList` is a container for `ObservableList<Task>` objects, each containing a unique filter over the full task list.

Each `GroupedTaskList` implements the following operations on top of those in `ViewTaskList`:

- A enum class which describes the valid `Category` groupings, and the corresponding methods of getting these groupings from a `String`.

- An `ObservableList` of `SubgroupTaskList` that represents the sub-groupings of each corresponding `Category`.
- A public method for instantiating a `GroupedTaskList` called `getGroupedList` with the return from `getFilteredTaskList()` as argument.
- An iterator method which iterates through a list of `SubgroupTaskList`.

Users can modify the `GroupTaskList` being displayed in the main panel by executing a `ListCommand`. The operation for retrieving the corresponding `GroupedTaskList` is exposed in the `Model` interface as `Model#getGroupedTaskList(Category category)`.
Currently, the supported groupings for JelphaBot are group by date (`GroupedTaskList.Category.DATE` and `GroupedByDateTaskList`) and group by module (`GroupedTaskList.Category.MODULE` and `GroupedByModuleTaskList`).

The following diagram shows the sequence flow of a `ListCommand` which modifies the currently shown Task List:

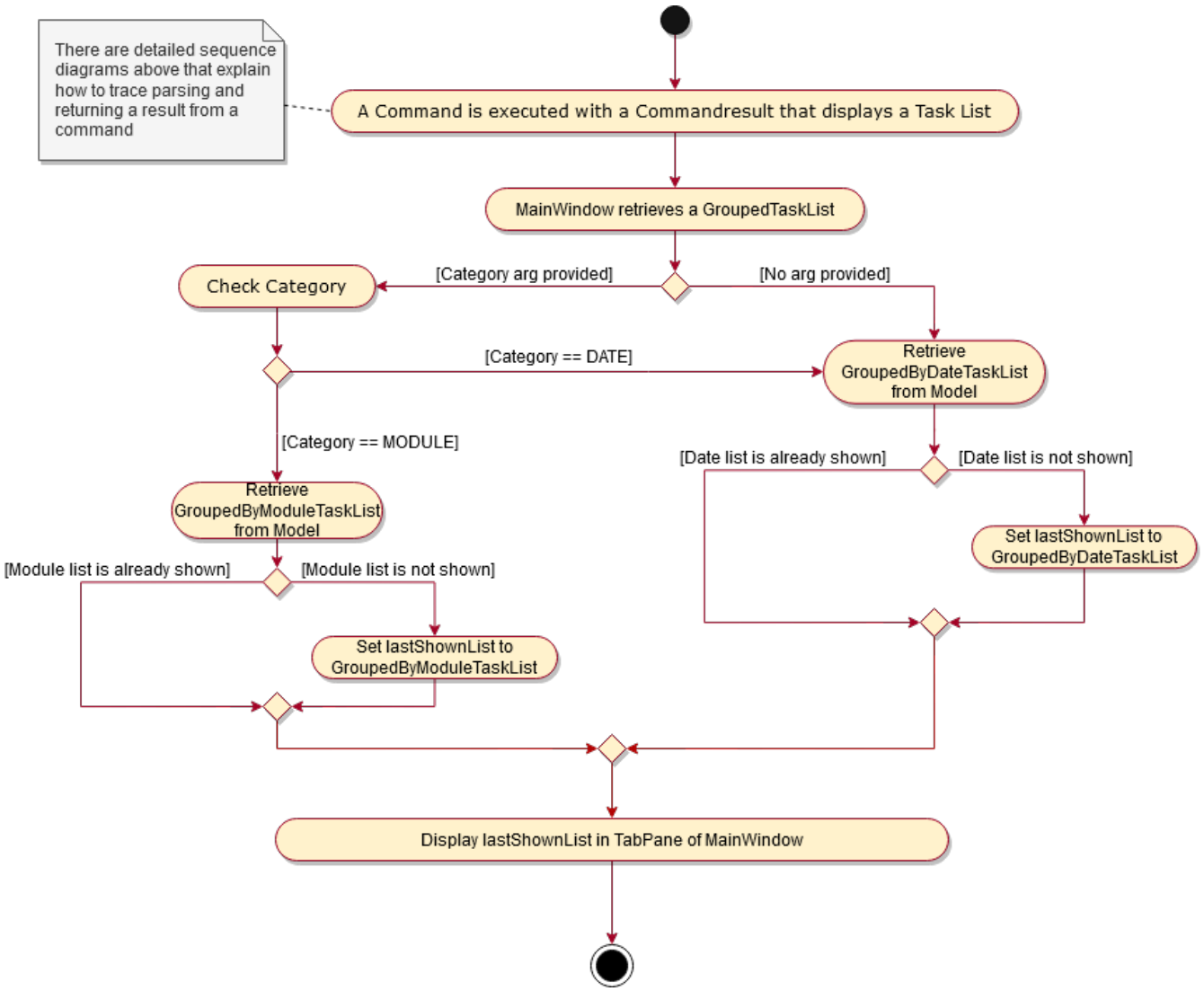

Figure 5. Activity Diagram showing the tab switch for ListCommand

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

Step 1. The user launches the application for the first time. The `MainWindow` will be initialized with `GroupedTaskListPanel` as a container for GroupedTaskList model objects. The panel is populated with sublists defined in `GroupedByDateTaskList`.

Step 2. The user executes ***list model*** to switch to category tasks by module code instead. `GroupedTaskListPanel` is repopulated with sublists defined in `GroupedByModuleTaskList`.

NOTE	If the user tries to switch to a `Cateory` which is already set, the command does not reinitialize the `GroupedTaskList` to prevent redundant filtering operations.

As `GroupedTaskList` has more than one underlying `ObservableList<Task>`, tasks cannot be retrieved the usual way. Thus, the `get()` function defined in the `ViewTaskList` interface must be implemented and used instead.
The following diagram shows the process of retrieving a `Task` from `ViewTaskList` when it is an instance of `GroupedTaskList`:

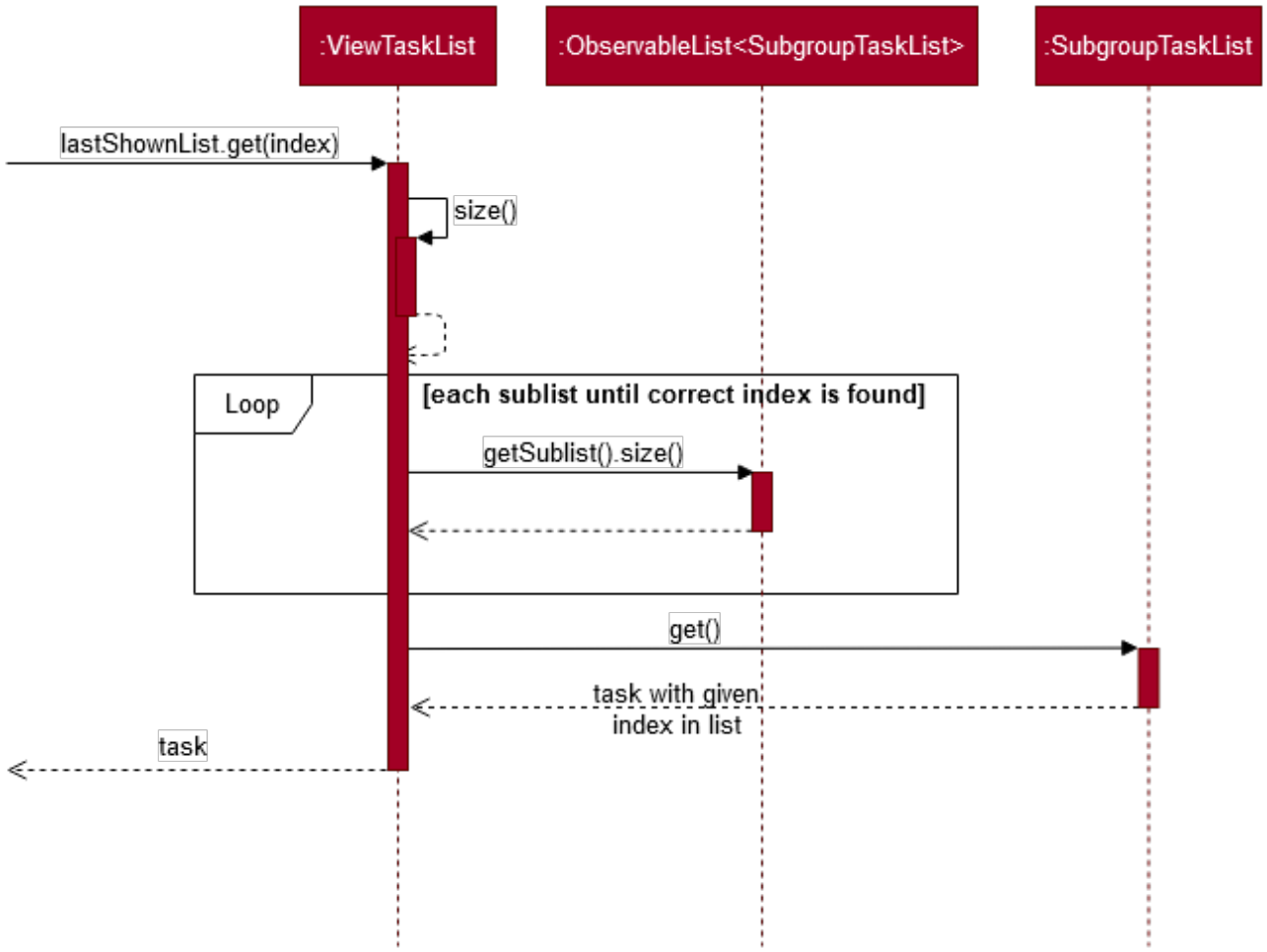

Figure 6. Sequence Diagram for `ViewTaskList.get()`

As the index passed as an argument to `lastShownList.get()` is a cumulative index, the implementation of `get()` in `ViewTaskList` has to iterate through each `SubgroupTaskList` stored within.

Tasks are organized via a two-dimensional list. In this case, a `Task` is rendered into a `TaskCard`, and `TaskCard` elements are rendered within `SubGroupTaskListCell` elements which are listed in `SubgroupTaskListPanel`. A populated `SubgroupTaskListPanel` element is rendered as a `GroupedTaskListCell` which is listed in the top-level `GroupedTaskListCell`.
`SubgroupTaskListCell` and `GroupedTaskListCell` implement the `ListViewCell<T>` interface of the `ListView<T>` class provided by JavaFX.

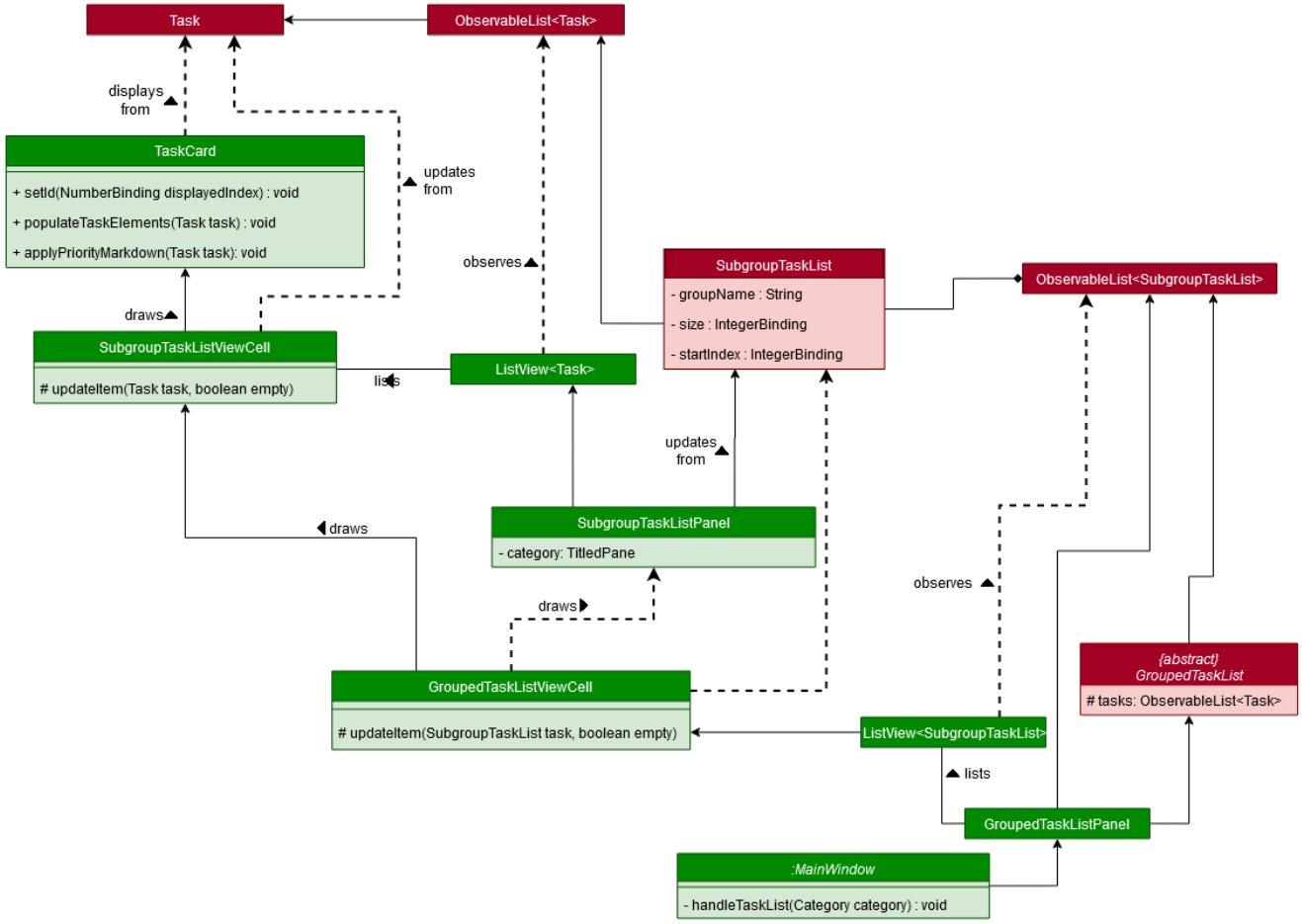

Figure 7. Class Diagram for UI classes displaying `GroupedTaskList`

The detailed interactions are described in the diagram shown above. As can be seen, the distribution of `ListViewCell` elements follows the way tasks are distributed within the model classes. Each `SubgroupTaskListPanel` is displaying a singular `SubgroupTaskList`, which further contains a list of `Task` entities.

The indexes displayed in each `TaskCard` is dynamically computed from a `NumberBinding` which computes the index of that element in the list. The `NumberBinding` observes the place of the task within the current `SubgroupTaskList` as well as the number of elements in the preceding sublists. The sum of both numbers gives the index for the current element, which is set using `setId()`. `TaskCard` elements are updated with `populateTaskElements()`.

Each `TaskCard` will also have a different visual presentation depending on the value of the `Priority` of the task. The method which adjusts the visual presentation of a `Task` is `applyPriorityMarkdown()`.

The following images show how `Task` entities of different priorities are displayed:

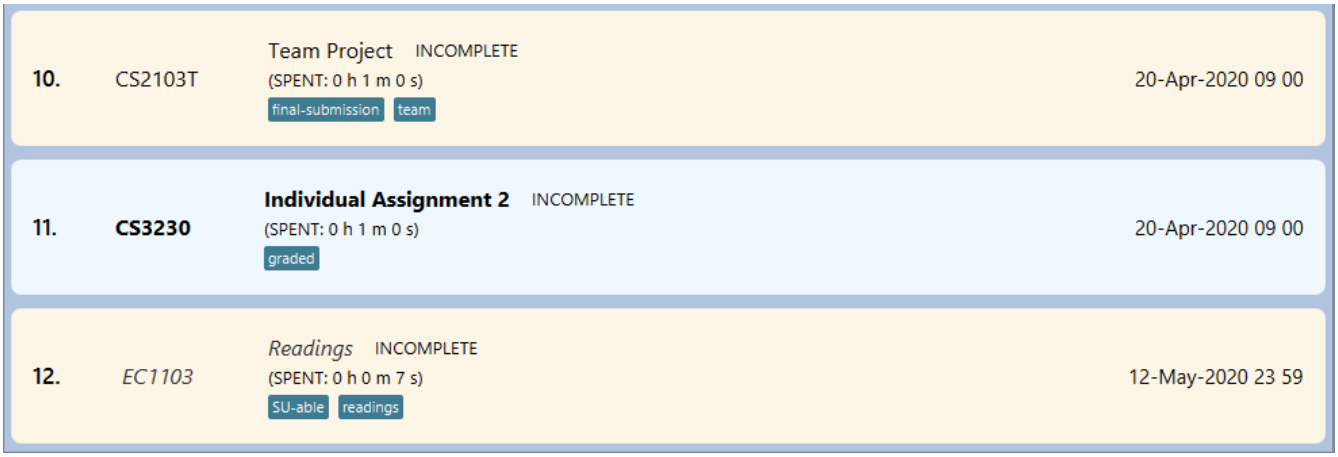

Figure 8. Markdown for Tasks of different priority. (Top to bottom: Normal, High, Low.)

Design Considerations

Aspect 1: `ListCommand` swaps to a different `ViewTaskList`

Refer to Activity Diagram showing the tab switch for ListCommand for the diagram describing this process.

- **Current solution**: Initializes each grouped list as each `ListCommand` is called and stores the latest list as `Model.lastShownList`.
  - Pros: Easy to implement. Scalable when more groupings are added.
  - Cons: Consecutive `ListCommand` operations which swap between different categories are expensive as the list is reinitalized each time.
  - Cons: It is hard to keep track of the exact type of list in `lastShownList`, which may lead to unexpected behavior.
- **Alternative 1:** Keep instances of all `GroupedTaskList` objects and update them as underlying Task List changes.
  - Pros: Consecutive `ListCommand` executions are less expensive.
  - Cons: All other commands that update the underlying list now have additional checks as each grouped list is updated.

Reason for chosen implementation:
The current solution was chosen in order to accomodate more `Category` implementations in the future. With only two classes inheriting `GroupedTaskList`, it is entirely feasible to implement both. However, since only one `GroupedTaskList` will be used at any time, and to take into account possible performance savings, only one `GroupedTaskList` implementation will exist at any one time.

Aspect 2: `get()` Task from `ViewTaskList` and iterate between Tasks.

Refer to Sequence Diagram for `ViewTaskList.get()` for the diagram describing this process.

- **Current solution**: Implement `get()` and `Iterator<Task>` in `ViewTaskList`.
  - Pros: Easy to implement. Scalable when more groupings are added.
  - Cons: Consecutive `ListCommand` operations are expensive as the list is reinitalized each time.
  - Cons: It is hard to keep track of the exact type of list in `lastShownList`, which may lead to unexpected behavior.
    - As a workaround, only operations defined in the `ViewTaskList` interface should be used.
- **Alternative 1:** Keep instances of all `GroupedTaskList` objects and update them as underlying Task List changes.
  - Pros: Consecutive `ListCommand` executions are less expensive.
  - Cons: All other commands that update the underlying `UniqueTaskList` will result in multiple update calls to `ViewTaskList`.

Reason for chosen implementation:
The current solution was chosen with integration with other tabs in mind. This implementation can easily be expanded to other tabs in a future version if other tabs also inherit from `ViewTaskList`. This allows ***add***, ***edit***, *delete and* ***_done*** commands to be executable from any tab, while still only requiring one `ViewTaskList` to be instantiated, which saves performance.

Aspect 3: Remove empty Categories in `GroupByModuleTaskList`

- **Current Solution**: UI displays problems from a `FilteredList<SubgroupTaskList>` and uses a `ListChangeListener<Task>` to maintain a set of unique module codes when the underlying task list is changed. The `ObservableSet<ModuleCode>` has a further `SetChangeListener<ModuleCode>` bound to it to remove categories that no longer contain any Tasks. This second listener directly removes unused categories from `GroupedByModuleTaskList`.
  - Pros: Consecutive changes to the underlying Task List are automatically reflected with a change in `SubgroupTaskList` categories.
  - Pros: The delegation of responsibilities between each `Listener` allows Single Responsibility Principle to be maintained.
  - Cons: Dependency between the two `Listener` classes has to be maintained.
- **Alternative 1:** Hide categories which are no longer used by adding a filter to the Task List returned.
  - Pros: Easy to implement and understand.
  - Cons: Not practical: as more Module Codes are added to the Task List, it might cause more and more hidden categories to be created which are expensive to filter through.
- **Alternative 2:** Abstract maintenance of the set of unique module codes to a `UniqueModuleCodeSet` class instanced in `UniqueTaskList`.
  - Pros: Easy to understand. Logic is further abstracted to a higher level and the new class is instanced together with the list that affects it.
  - Pros: The returned `ObservableSet<ModuleCode>` from `UniqueModuleCodeSet` can be made unmodifiable which would prevent unauthorized changes to the `ObservableSet` from other classes.
  - Cons: Implementation is challenging and prone to bugs, requiring significant testing. Due to the time of writing this Developer guide, the release is nearing V1.4 and time is spent fixing bugs for release instead.
  - This could be a proposed update in the future.

Reason for chosen implementation:
The best solution would be to create a `UniqueModuleCodeSet` instanced in `UniqueTaskList`, which would have the best scalability and abstraction. In addition, since such a set would be updated regularly, less mantainence is needed inside classes that require a list of unique `ModuleCode` entities. However, due to time constraints, such an implementation was not chosen. However, the current solution mimics the best solution as closely as possible by using `SetChangeListener` to update the `SubgroupTaskList` list. This means that a returned `UnmodifiableObservableSet` from `UniqueModuleCodeSet` can be substituted easily whenever such a refactoring is done.