GradTrak - Developer Guide

By: `Team W14-4` Since: `Feb 2019` Licence: `MIT`

1. Setting up

Go through the following sections to set up GradTrak.

1.1. Checking prerequisites

Ensure that each of the following software has been installed.

1. **JDK** `9` or later

   WARNING JDK `10` on Windows will fail to run tests in headless mode due to a JavaFX bug. Windows developers are highly recommended to use JDK `9`.

2. **IntelliJ** IDE

   NOTE IntelliJ by default has Gradle and JavaFx plugins installed.
   Do not disable them. If you have disabled them, go to `File` > `Settings` > `Plugins` to re-enable them.

1.2. Setting up the project

Follow the instructions below to set up the project on your computer.

1. Fork this repo, and clone the fork to your computer.
2. Open IntelliJ (if you are not in the welcome screen, click `File` > `Close Project` to close the existing project dialog first).
3. Set up the correct JDK version for Gradle.
   a. Click `Configure` > `Project Defaults` > `Project Structure`.
   b. Click `New…` and find the directory of the JDK.
4. Click `Import Project`.
5. Locate the `build.gradle` file and select it. Click `OK`.
6. Click `Open as Project`.
7. Click `OK` to accept the default settings.
8. Open a console and run the command `gradlew processResources` (Mac/Linux: `./gradlew processResources`). It should finish with the `BUILD SUCCESSFUL` message.
   This will generate all resources required by the application and tests.
9. Open `MainWindow.java` and check for any code errors.
   a. Due to an ongoing issue with some of the newer versions of IntelliJ, code errors may be detected even if the project can be built and run successfully.

   b. To resolve this, place your cursor over any of the code section highlighted in red. Press kbd:[ALT + ENTER], and select `Add '--add-modules=…' to module compiler options` for each error.

10. Repeat this for the test folder as well (e.g. check `HelpWindowTest.java` for code errors, and if so, resolve it the same way).

1.3. Verifying the setup

Follow the instructions below to verify that the setup is successful.

1. Run `seedu.address.MainApp` and try a few commands.
2. Run the tests to ensure all of them pass.

1.4. Configurating the project

Go through the following sections to configure the project.

1.4.1. Configuring the coding style

This project follows oss-generic coding standards. IntelliJ's default style is mostly compliant with ours but it uses a different import order. Follow the instructions below to rectify this issue.

1. Go to `File` > `Settings…` (Windows/Linux), or `IntelliJ IDEA` > `Preferences…` (macOS).
2. Select `Editor` > `Code Style` > `Java`.
3. Click on the `Imports` tab to set the order.
   - For `Class count to use import with '*'` and `Names count to use static import with '*'`: Set to `999` to prevent IntelliJ from contracting the import statements.
   - For `Import Layout`: The order is `import static all other imports`, `import java.*`, `import javax.*`, `import org.*`, `import com.*`, `import all other imports`. Add a `<blank line>` between each `import`.

Optionally, you can follow the UsingCheckstyle.adoc document to configure Intellij to check style-compliance as you write code.

1.4.2. Setting up CI

Set up Travis to perform Continuous Integration (CI) for your fork. See UsingTravis.adoc to learn how to set it up.

After setting up Travis, you can optionally set up coverage reporting for your team fork (see UsingCoveralls.adoc).

NOTE Coverage reporting could be useful for a team repository that hosts the final version but it is not that useful for your personal fork.

Optionally, you can set up AppVeyor as a second CI (see UsingAppVeyor.adoc).

NOTE — Having both Travis and AppVeyor ensures your App works on both Unix-based platforms and Windows-based platforms (Travis is Unix-based and AppVeyor is Windows-based)

1.4.3. Getting started with coding

Follow the instructions below when you are ready to start coding.

1. Get some sense of the overall design by reading [Design-Architecture].
2. Take a look at Appendix A, *Suggested Programming Tasks to Get Started*.

2. Design

The following sections explain the design of GradTrak.

2.1. Architecture

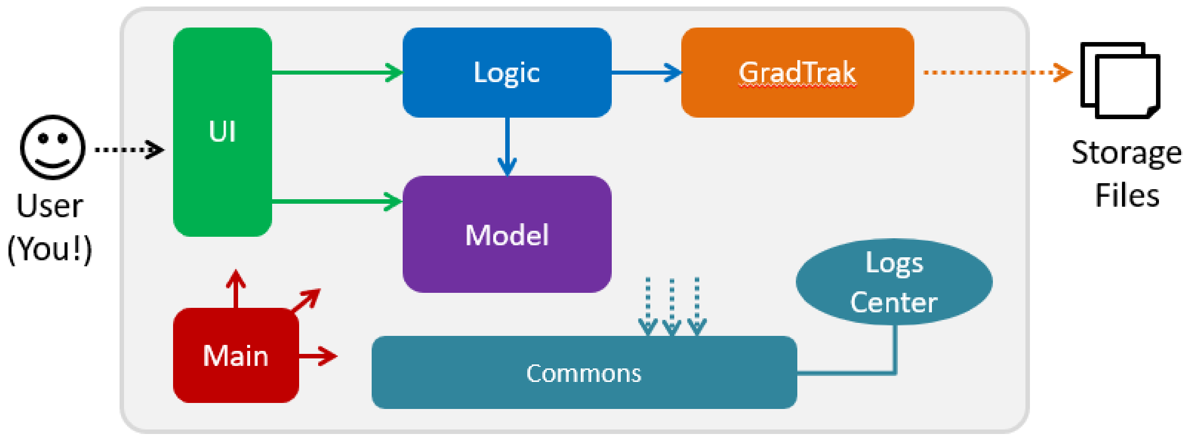


Figure 1. Architecture Diagram

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

TIP — The `.pptx` files used to create diagrams in this document can be found in the diagrams folder. To update a diagram, modify the diagram in the pptx file, select the objects of the diagram, and choose `Save as picture`.

`Main` has only one class called `MainApp`. It is responsible for:

- At app launch: Initializing the components in the correct sequence and connecting them with one another.
- At shut down: Shutting down the components and invoking cleanup methods 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 user interface (UI) of the App.
- **`Logic`**: The command executor.
- **`Model`**: The model holding the data of the App in-memory.
- **`Storage`**: The storage which reads data from and writes data to the hard disk.

Each of the four components above:

- Defines its Application Programming Interface (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 its API in the `Logic.java` interface and exposes its functionality using the `LogicManager.java` class.

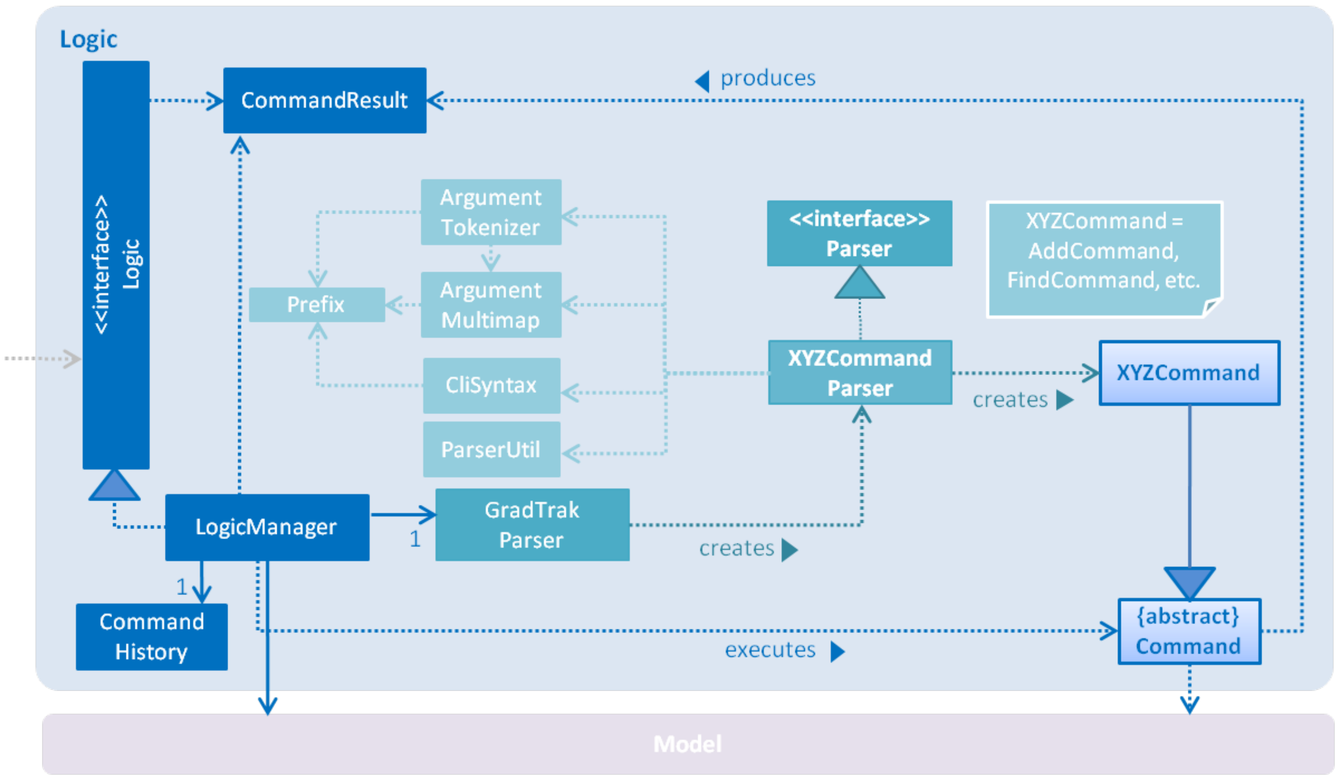


Figure 2. Class Diagram of the Logic Component

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 `delete 1`.

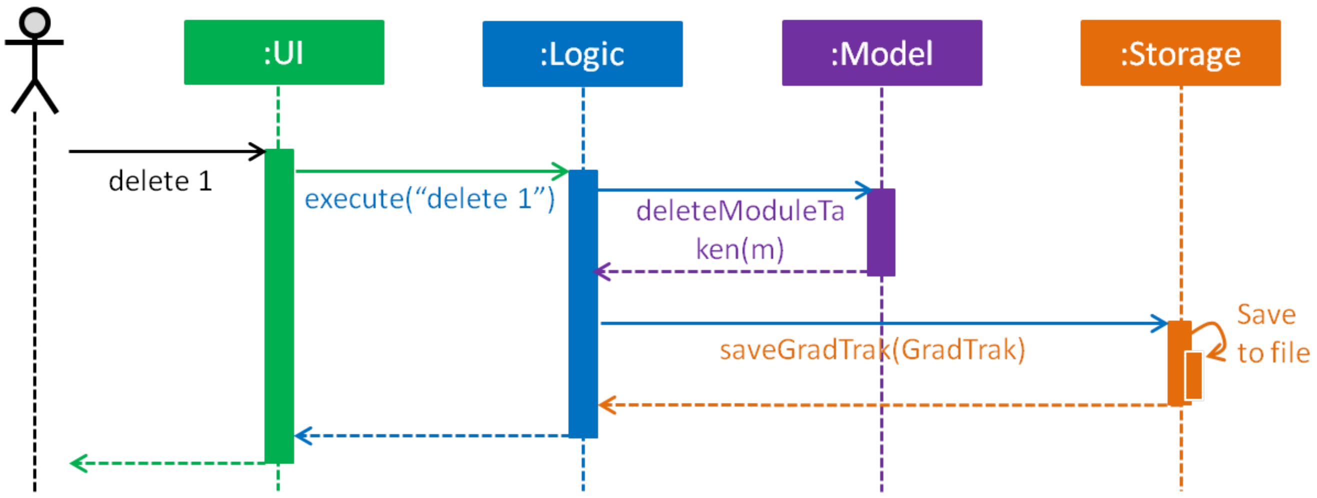


Figure 3. Component interactions for `delete 1` *command*

The sections below give more details of each component.

2.2. UI component

[UiClassDiagram] | *UiClassDiagram.png*

Figure 4. Structure of the UI Component

API : `Ui.java`

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

The `UI` component uses 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:

- Executes user commands using the `Logic` component.
- Listens for changes to `Model` data so that the UI can be updated with the modified data.

2.3. Logic component

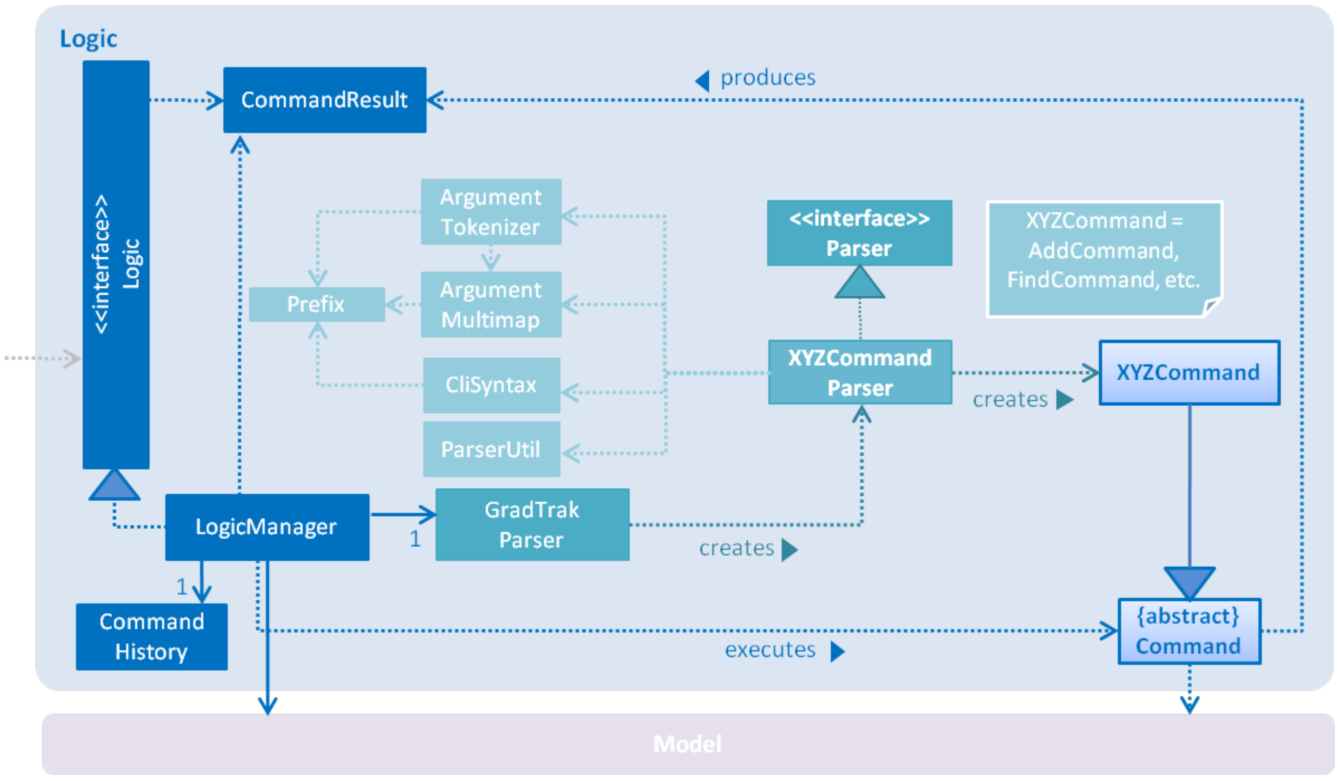


Figure 5. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `GradTrakParser` 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 module).
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.

[DeletePersonSdForLogic] | *DeletePersonSdForLogic.png*

Figure 6. Interactions Inside the Logic Component for the `delete 1` Command

2.4. Model component

[ModelClassDiagram] | *ModelClassDiagram.png*

Figure 7. Structure of the Model Component

API : `Model.java`

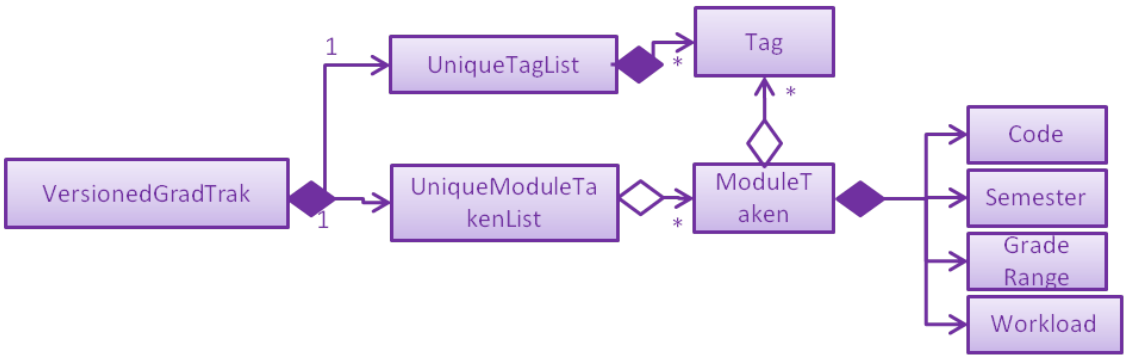
The `Model`:

- stores a `UserPref` object that represents the user's preferences.

- stores the GradTrak data.
- exposes an unmodifiable `ObservableList<ModuleTaken>` 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 `GradTrak`, which `ModuleTaken` can reference. This would allow `GradTrak` to only require one `Tag` object per unique `Tag`, instead of each `ModuleTaken` needing their own `Tag` object. An example of how such a model may look like is given below.



2.5. Storage component

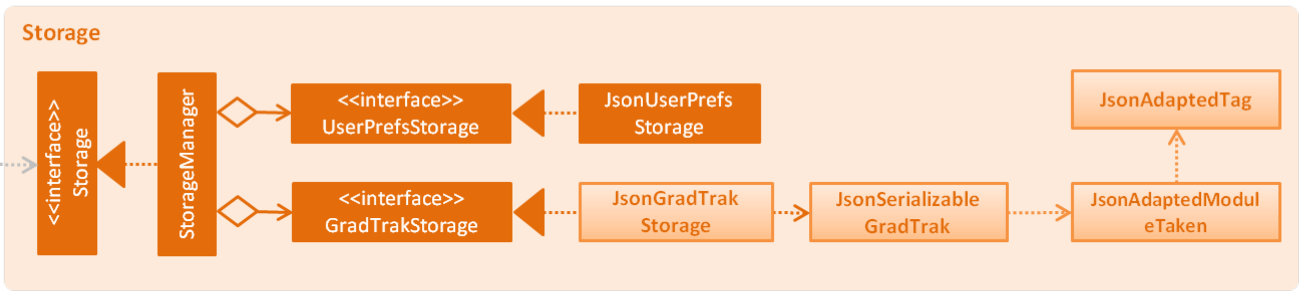


Figure 8. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

- can save `UserPref` objects in json format and read it back.
- can save the GradTrak data in json format and read it back.

2.6. Common classes

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

3. Implementation

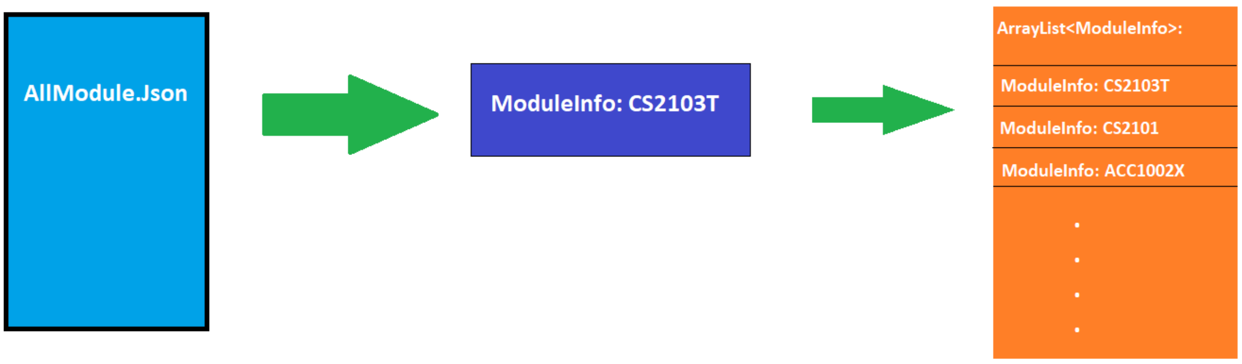
This section describes some noteworthy details on how certain features are implemented.

3.1. Displaymod feature

3.1.1. Current Implementation

The `displaymod` is a search function that displays all available information with regards to the module codes the user types into the command box.

When the application is launched, a JSON file containing all of NUS module information is then converted into an Object Class called ModuleInfo individually and stored into an ObservableList<ModuleInfo> called ModuleInfoList.



The usage of ModuleInfoList is only temporary as it is passed into `ModelManager` and then converted into an ObservableList<>.

NOTE	The purpose of having ModuleInfoList is not only for temporary use; it also serves as form of Error handling if the application is unable to find the location of the JSON file containing all of the module information, a blank ModuleInfoList is handed to the ModelManager.

```
//Get an non Modifiable List of all modules and use a filtered list based on that to search for modules
this.allModules = allModules.getObservableList();
this.displayList = new FilteredList<>(this.allModules);
```

This is done as `ObservableList<>` comes with a `FilteredList<>` feature: which wraps an ObservableList and filters the contents based on `predicates`.

- `Predicates` — All search keywords i.e **Module Code** is saved as a list of `predicates`.

This allows for easier search throughout the list of all Module Information, as the User can search for multiple modules in a single search.

The FilteredList is then collected and the ModuleInfo Objects will be formatted into Strings so that the information required is displayed by the UI.

3.1.2. Design Considerations

Aspect: How Displaymod executes

- **Current Implementation :** Searches based on Module Codes Only
  - Pros: Searches is slightly faster since its only based on Module Codes
  - Cons: Limited search since it requires User to know Module Codes beforehand.
- **Future Implementation :** Search based on keywords
  - Pros: User can search for Modules based on keywords thus require no prior knowledge on a particular module code
  - Cons: Have to combine all module information into a single String and search for keywords; slower searches.

3.1.3. Aspect: Data Structure Used

- **Current Implementation :** `ObservableList<>` is used
  - Pros: Allows for `FilteredList<>` to be used based on predicates; easy implementation.
  - Cons: Requires additional classes to be implemented to handle the use of `Predicates`.
- **Alternative :** Sticking to the `ArrayList<>`
  - Pros: Easy to handle as it is a simple data structure.
  - Cons: Harder to search for specific Keywords(future implementation)

3.2. Undo/Redo feature

3.2.1. Current Implementation

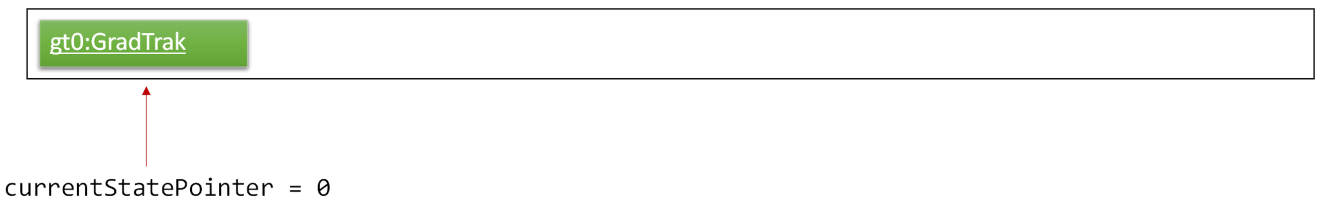
The undo/redo mechanism is facilitated by `VersionedGradTrak`. It extends `GradTrak` with an undo/redo history, stored internally as an `gradTrakStateList` and `currentStatePointer`. Additionally, it implements the following operations:

- `VersionedGradTrak#commit()` — Saves the current GradTrak state in its history.
- `VersionedGradTrak#undo()` — Restores the previous GradTrak state from its history.
- `VersionedGradTrak#redo()` — Restores a previously undone GradTrak state from its history.

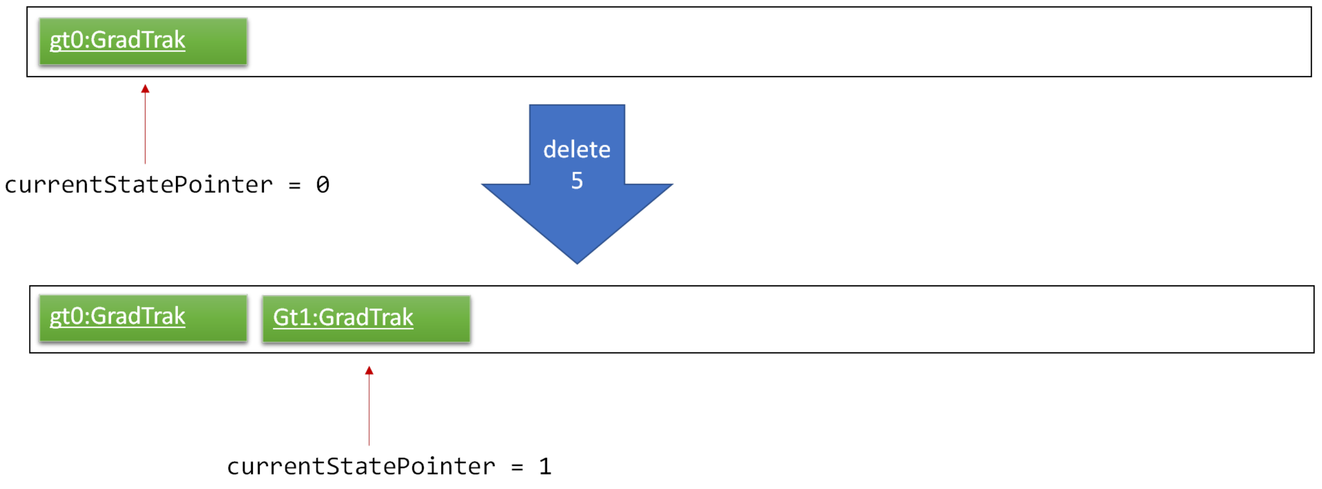
These operations are exposed in the `Model` interface as `Model#commitGradTrak()`, `Model#undoGradTrak()` and `Model#redoGradTrak()` respectively.

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

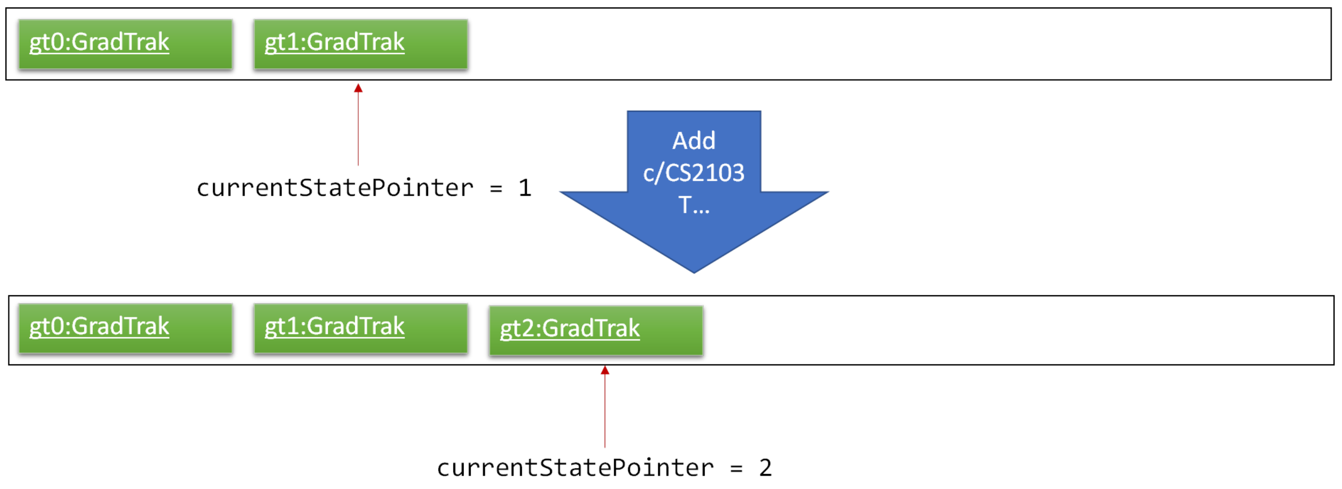
Step 1. The user launches the application for the first time. The `VersionedGradTrak` will be initialized with the initial GradTrak state, and the `currentStatePointer` pointing to that single GradTrak state.



Step 2. The user executes `delete 5` command to delete the 5th module in the GradTrak. The `delete` command calls `Model#commitGradTrak()`, causing the modified state of the GradTrak after the `delete 5` command executes to be saved in the `GradTrakStateList`, and the `currentStatePointer` is shifted to the newly inserted GradTrak state.

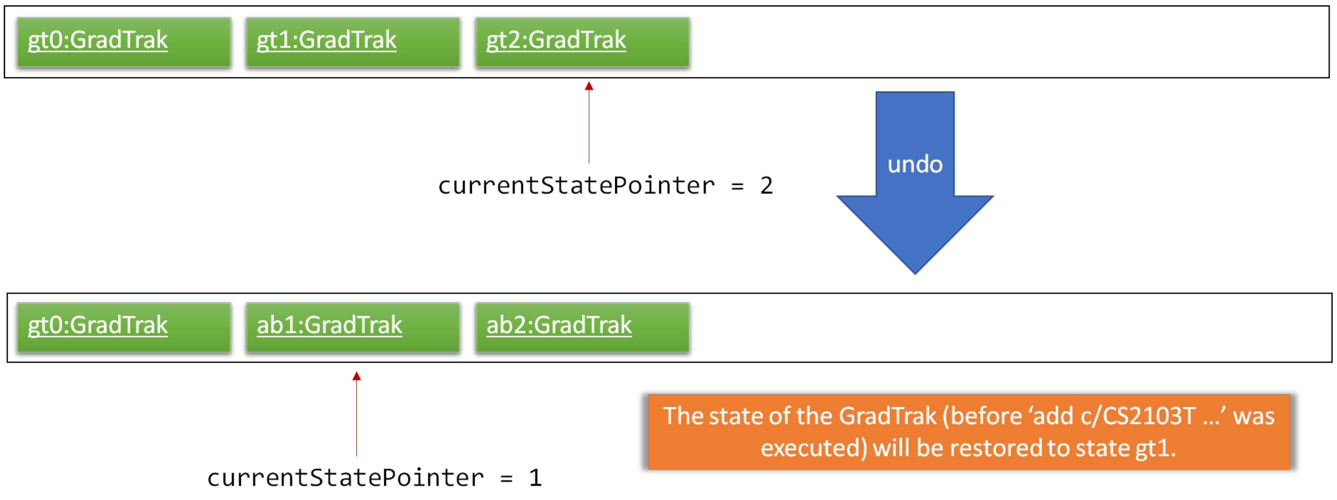


Step 3. The user executes `add c/CS2103T …` to add a new module. The `add` command also calls `Model#commitGradTrak()`, causing another modified GradTrak state to be saved into the `gradTrakStateList`.



NOTE If a command fails its execution, it will not call `Model#commitGradTrak()`, so the GradTrak state will not be saved into the `gradTrakStateList`.

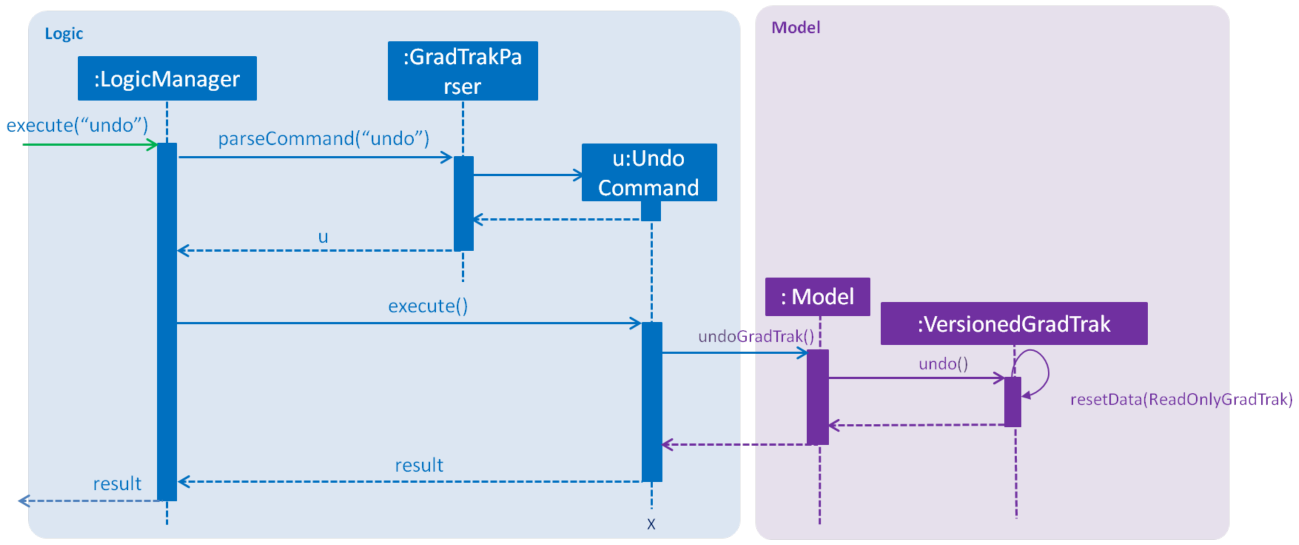
Step 4. The user now decides that adding the module was a mistake, and decides to undo that action by executing the `undo` command. The `undo` command will call `Model#undoGradTrak()`, which will shift the `currentStatePointer` once to the left, pointing it to the previous GradTrak state, and restores the GradTrak to that state.



NOTE

If the `currentStatePointer` is at index 0, pointing to the initial GradTrak state, then there are no previous GradTrak states to restore. The `undo` command uses `Model#canUndoGradTrak()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the undo.

The following sequence diagram shows how the undo operation works:

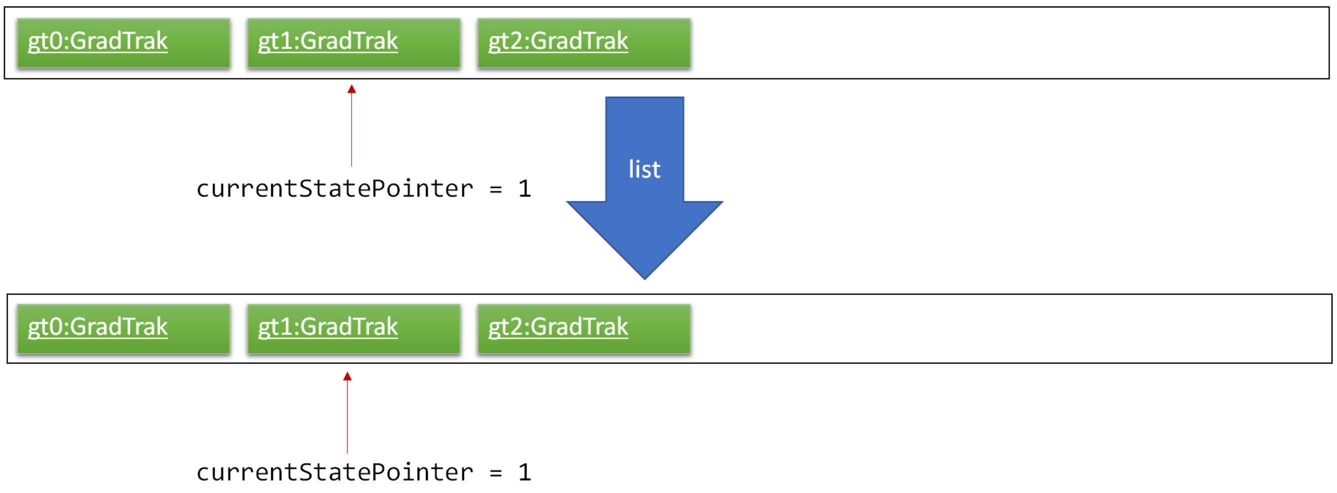


The `redo` command does the opposite — it calls `Model#redoGradTrak()`, which shifts the `currentStatePointer` once to the right, pointing to the previously undone state, and restores the GradTrak to that state.

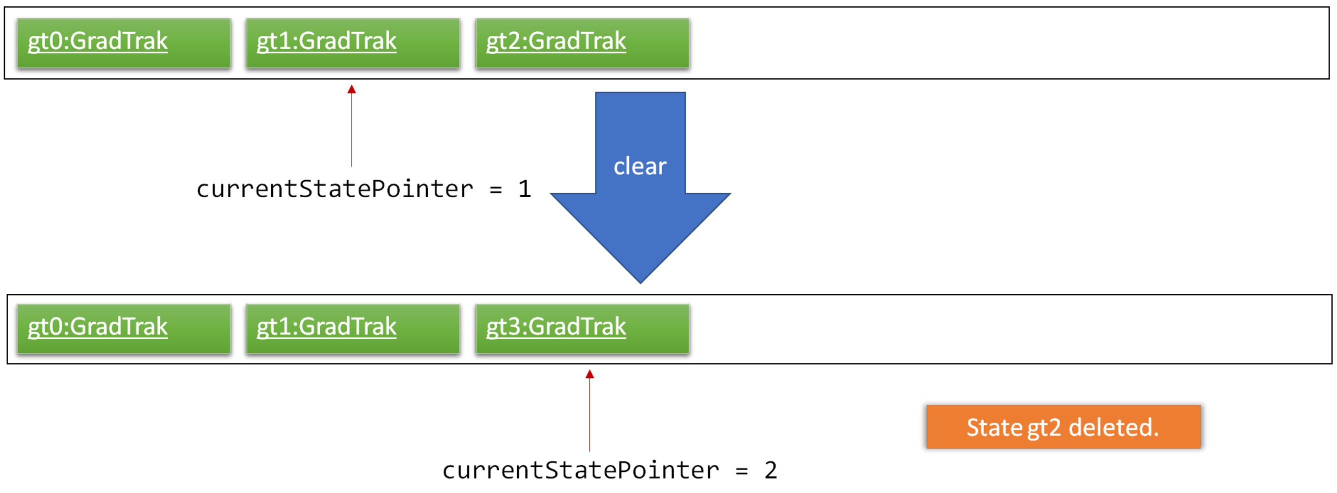
NOTE

If the `currentStatePointer` is at index `gradTrakStateList.size() - 1`, pointing to the latest GradTrak state, then there are no undone GradTrak states to restore. The `redo` command uses `Model#canRedoGradTrak()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the redo.

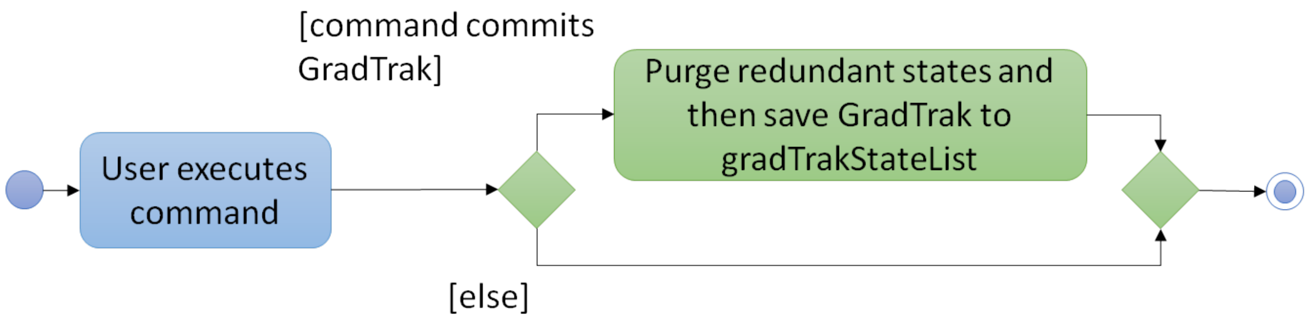
Step 5. The user then decides to execute the command `list`. Commands that do not modify the GradTrak, such as `list`, will usually not call `Model#commitGradTrak()`, `Model#undoGradTrak()` or `Model#redoGradTrak()`. Thus, the `gradTrakStateList` remains unchanged.



Step 6. The user executes `clear`, which calls `Model#commitGradTrak()`. Since the `currentStatePointer` is not pointing at the end of the `gradTrakStateList`, all GradTrak states after the `currentStatePointer` will be purged. We designed it this way because it no longer makes sense to redo the `add c/CS2103T …` command. This is the behavior that most modern desktop applications follow.



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


[command commits GradTrak]
User executes command
Purge redundant states and then save GradTrak to gradTrakStateList
[else]

3.2.2. Design Considerations

Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Saves the entire GradTrak.
  - Pros: Easy to implement.
  - Cons: May have performance issues in terms of memory usage.

- **Alternative 2:** Individual command knows how to undo/redo by itself.
    - Pros: Will use less memory (e.g. for `delete`, just save the person being deleted).
    - Cons: We must ensure that the implementation of each individual command are correct.

Aspect: Data structure to support the undo/redo commands

- **Alternative 1 (current choice):** Use a list to store the history of GradTrak states.
    - Pros: Easy for less experienced developers to understand.
    - Cons: Logic is duplicated twice. For example, when a new command is executed, we must remember to update both `HistoryManager` and `VersionedGradTrak`.
- **Alternative 2:** Use `HistoryManager` for undo/redo
    - Pros: We do not need to maintain a separate list, and just reuse what is already in the codebase.
    - Cons: Requires dealing with commands that have already been undone: We must remember to skip these commands. Violates Single Responsibility Principle and Separation of Concerns as `HistoryManager` now needs to do two different things.

3.3. Checking whether Course Requirement is fulfilled Feature

3.3.1. Current Implementation

Users are able to check whether they have satisfied course requirements through a function `displayreq`. Similar to the implementation of the feature related to `displaymod`, when the application is launched, a JSON file containing pre-existing information on courses (either default or pre-defined by users) and their respective course requirement will loaded as `Course` objects and `CourseRequirement` interface (`Course` contains multiple `CourseRequirement` interface implementees) and stored in an `CourseList` inside of `ModelManager`. While the app does not support adding or removing courses and course requirements, users can still define their own course or course requirements by modifying the JSON file.

We employ the "Composite" design pattern for this class as there are desirable boolean binary operations such as "and", "or", that we would like to apply on two different requirements and we would want to regard the "simple" CourseRequirement, (one that can be achieved without "and", "or") and a more complicated one (one that can only be achieved with "and", "or") to be the same. As such the logical choice would be to make `CourseRequirement` an interface and its implementation is restricted to the `PrimitiveRequirement` and `CompositeRequirement` classes.

[CompositeDesignPattern] | *CompositeDesignPattern.png*

As the name suggests, `CompositeRequirement` is made up of two other `CourseRequirements`, a `LogicalConnector` enumerations (which is used to represent logical binary operators such conjunctions and disjunctions). On the other hand, a `PrimitiveRequirement` is a standalone `CourseRequirement` implementation that contains a `Condition` class, which is really a helper class that is used to check whether a list of `ModuleInfoCode` can satisfy the `CourseRequirement`. The `Condition` class has the following attribute to decide whether the a condition of `PrimitiveRequirement` is fulfilled, namely one String, Regular Expression, and an integer `minToSatisfy`. We will further elaborate the details below.

There currently 3 ways that `CourseRequirement` provides information to the user:

- `isFulfilled()` — a method that accepts a list of `ModuleInfoCode` and returns a `boolean` to indicate whether the list of `ModuleInfoCode` can satisfy the requirement.
  - For `PrimitiveRequirement`, if the number of distinct `ModuleInfoCode` satisfies the list of Regular Expression pre-defined in `Condition` class exceeds `minToSatisfy`, we define the `PrimitiveRequirement` to be satisfied.
  - Similarly, for `CompositeRequirement`, it depends on the `LogicalConnector`; if we have "and", then both `CourseRequirement` must be satisfied in order for `CompositeRequirement` to be satisfied and likewise for "or".
- `percentageFulfilled()` — a method that also accepts a list of ModuleInfoCode returns a `double` value on the percentage of completion of the `CourseRequirement`.
  - For `PrimitiveRequirement` this will be the proportion of distinct modules that satisfy at least one of the regular expression divided by `minToSatisfy`.
  - For `CompositeRequirement`, a `LogicalConnector` of "or" wil result in the max of the two requirements whereas for "and", we take average of both classes to approximate the degree of completion.
- `getUnfulfilled()` — a method that accepts a list of `ModuleInfoCode` and returns a list of Strings that matches regular expressions (defined under Condition) of the `ModuleInfoCode` that can be used to satisfy the `CourseRequirement`.

3.3.2. Design Considerations

Aspect: How condition checks whether Course Requirement is fulfilled

- **Alternative 1 (current choice): Checking Requirement fulfilled by only using `ModuleInfoCode` of `ModuleTaken`**
  - Pros: Easier to implement since we are restricting scope to only checking of strings
  - Cons: There could be 'corner cases' that we left out; actual NUS Course Requirement that we cannot represent by merely checking the module code.
- **Alternative 2: Checking Requirement fulfilled by accessing any attribute of `ModuleTaken`**
  - Pros: Increased flexibility allows for more powerful expressions and increased usability for users side
  - Cons: More Java classes/coding required (to Separate Responsibilities and Concerns) also means that more tests and storage components for different classes.

Alternative 1 is chosen over alternative 2 because it is much easier to implement in terms of scope. Another reason why we chose alternative 1 over alternative 2 is due to the lack of time. For most cases, alternative 1 seems to be sufficient. However, in the future, we might extend the Condition class to check for other attributes of `ModuleTaken`.

3.4. Module recommendation feature

The module recommendation feature displays modules which the user the recommended to take based on existing GradTrak modules and specific course requirements. It generates a list of module codes together with its corresponding title and requirement type satisfied. The entire list is displayed on the Result Panel upon entering the `rec` command.

3.4.1. Current implementation

Each recommended module is represented by a `RecModule` which contains a unique `ModuleInfoCode`, `ModuleInfoTitle` and its corresponding `CourseReqType` satisfied. When `ModelManager` is initialised, `Model#getObservableRecModuleList` is called which generates an `ObservableList` of `RecModule` , one for each module in the entire `ModuleInfoList`. This list is wrapped in a `FilteredList`, which is further wrapped in a `SortedList`, both stored in `ModelManager`. At this point, all `RecModule` in the list contain an empty `CourseReqType` field.

When the `rec` command is entered, the sequence of execution is as follows:

1. `Model#updateRecModuleList` is called, which creates a `RecModulePredicate` given the user's `Course` and `GradTrak`, and a `RecModuleComparator`.
2. The `RecModulePredicate` is applied to the `FilteredList` of `RecModule`.
    a. It first tests if the `ModuleInfoCode` in a `RecModule` (call it `codeToTest`) is eligible to be taken, using an `EligibleModulePredicate` which takes in `GradTrak`. Those `RecModule` corresponding to `ModuleTaken` in `GradTrak` which are already passed or to be taken in a future semester (non-failed modules) are filtered out at this stage.
    b. The `codeToTest` is then passed into `Course#getCourseReqTypeOf`, which in turn calls `CourseRequirement#canFulfill` for each `CourseRequirement` listed in `Course`. A list of `CourseReqType` that the `codeToTest` can satisfy is returned. This `courseReqTypeList` is sorted by the priority of `CourseReqType` as defined in the `enum` class: CORE, TE, IE, FAC, GE, UE.
    c. A `nonFailedCodeList` of `ModuleInfoCode` corresponding to non-failed `ModuleTaken` is retrieved from `GradTrak`. For each `CourseReqType` in the `courseReqTypeList` (highest priority first):
        i. `Course#isCodeContributing` is called, which takes in the `CourseReqType`, `nonFailedCodeList` and `codeToTest`.
        ii. For each `CourseRequirement` listed in `Course` corresponding to the given `CourseReqType`, `CourseRequirement#getUnfulfilled` is called which takes in the `nonFailedCodeList` and returns an `unfulfilledRegexList` of RegExes not satisfied.
        iii. If the `codeToTest` matches any of the RegExes in the `unfulfilledRegexList`, `Course#isCodeContributing` returns `true` and the loop for `courseReqTypeList` terminates.
    d. The `CourseReqType` of highest priority satisfied by `codeToTest` is then set into the `RecModule`. However, if the `codeToTest` does not contribute to any of the `CourseRequirement` listed in `Course`, the `RecModule` is filtered out.
3. The `RecModuleComparator` is applied to the `SortedList` of `RecModule`. It sorts the list in decreasing priority of the `CourseReqType` satisfied by the `RecModule`. Those `RecModule` with equal priority are sorted by module level (the first numerical digit of its `ModuleInfoCode`), considering the fact that lower level modules are usually taken first. In the case of equal priority and module level,

lexicographical sorting is used.

4. The `SortedList` of `RecModule` is retrieved from the `ModelManager`. If it is empty, the user will be notified that all course requirements have been satisfied by the current GradTrak. Otherwise, it will be displayed to the user in the Result Panel.

If there are changes to `GradTrak` (adding, editing or deleting modules) or `Course` (changing the Focus Area), the `rec` command must be run again to reflect the updated recommendation list.

3.4.2. Design Considerations

Aspect: Sorting of recommendation list

- **Alternative 1 (current choice): Recommendation list is sorted by a fixed order of `CourseReqType` priority as defined in the `enum` class**
  - Pros: Easy to implement and modify
  - Cons: User may have his own order of priority that differs from the default one
- **Alternative 2: Recommendation list can be sorted by a custom order defined by the user**
  - Pros: User can sort the list according to his own preferences
  - Cons: Difficult to implement if several parameters for sorting is allowed; input method for the custom order is problematic

Aspect: How `CourseReqType` for each tested module is stored

- **Alternative 1 (current choice): `CourseReqType` is stored in `RecModule` together with `ModuleInfoCode` and `ModuleInfoTitle`**
  - Pros: More organised; any additional information the user wishes to be displayed can also be stored in `RecModule`
  - Cons: More space required for creating `RecModule` for all modules
- **Alternative 2 (previous choice): A `HashMap` of `ModuleInfoCode` to `CourseReqType` is stored; the map is reset before each `rec` command**
  - Pros: Only need to store `ModuleInfoCode` instead of `RecModule`
  - Cons: Retrieving other information of the module for displaying in the Result Panel is slightly problematic

3.4.3. Possible Improvements

1. Allow the user to display a module’s information (from displaymod command) using its index in the recommendation list
2. Allow the user to add a module to GradTrak using its index in the recommendation list

3.5. Semester CAP and workload management feature

3.5.1. Current Implementation

Users are able to set their preferred minimum and maximum CAP limits for each semester. They are also able to set the minimum and maximum workload limits for each semester in terms of the number of hours per week in terms of lectures, tutorials, labs, projects, and preparation.

The current stored semester limits will be loaded from the GradTrak.json file into a list of `SemLimit` Objects for each semester on app startup. If the user has not set any limits, the json file will contain a list with default limit values.

Users able to set the expected minimum and maximum grade for each module they take, as well as the number of workload hours. The `EditCommand` is used to set the grades and workload expected using an single argument for each variable.

Users can check if their expected grades and workload per semester falls within their preferred limits. A class `LimitChecker` handles the computation of CAP and total workload of the semesters and generates a table in HTML with the information computed.

3.6. 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 3.7, “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

3.7. Configuration

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

4. Documentation

We use asciidoc for writing documentation.

NOTE We chose asciidoc over Markdown because asciidoc, although a bit more complex than Markdown, provides more flexibility in formatting.

4.1. Editing Documentation

See UsingGradle.adoc to learn how to render `.adoc` files locally to preview the end result of your edits. Alternatively, you can download the AsciiDoc plugin for IntelliJ, which allows you to preview the changes you have made to your `.adoc` files in real-time.

4.2. Publishing Documentation

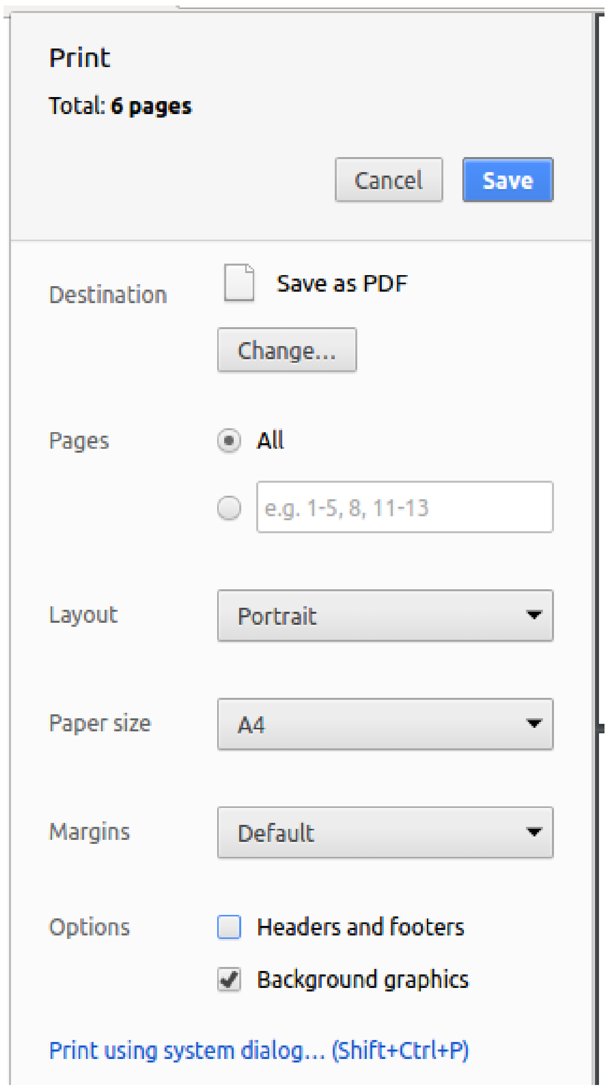
See UsingTravis.adoc to learn how to deploy GitHub Pages using Travis.

4.3. Converting Documentation to PDF format

We use Google Chrome for converting documentation to PDF format, as Chrome’s PDF engine preserves hyperlinks used in webpages.

Here are the steps to convert the project documentation files to PDF format.

1. Follow the instructions in UsingGradle.adoc to convert the AsciiDoc files in the `docs/` directory to HTML format.
2. Go to your generated HTML files in the `build/docs` folder, right click on them and select `Open with` → `Google Chrome`.
3. Within Chrome, click on the `Print` option in Chrome’s menu.
4. Set the destination to `Save as PDF`, then click `Save` to save a copy of the file in PDF format. For best results, use the settings indicated in the screenshot below.


Print

Total: **6 pages**

Cancel | Save

Destination: Save as PDF (Change...)

Pages: [✓ All] e.g. 1-5, 8, 11-13

Layout: Portrait

Paper size: A4

Margins: Default

Options: ☐ Headers and footers ☑ Background graphics

Print using system dialog... (Shift+Ctrl+P)

Figure 9. Saving documentation as PDF files in Chrome

4.4. Site-wide Documentation Settings

The `build.gradle` file specifies some project-specific asciidoc attributes which affects how all documentation files within this project are rendered.

TIP Attributes left unset in the `build.gradle` file will use their **default value**, if any.

Table 1. List of site-wide attributes

Attribute name	Description	Default value
`site-name`	The name of the website. If set, the name will be displayed near the top of the page.	*not set*
`site-githuburl`	URL to the site's repository on GitHub. Setting this will add a "View on GitHub" link in the navigation bar.	*not set*

Attribute name	Description	Default value
`site-seedu`	Define this attribute if the project is an official SE-EDU project. This will render the SE-EDU navigation bar at the top of the page, and add some SE-EDU-specific navigation items.	*not set*

4.5. Per-file Documentation Settings

Each `.adoc` file may also specify some file-specific asciidoc attributes which affects how the file is rendered.

Asciidoctor's built-in attributes may be specified and used as well.

TIP Attributes left unset in `.adoc` files will use their **default value**, if any.

Table 2. List of per-file attributes, excluding Asciidoctor's built-in attributes

Attribute name	Description	Default value
`site-section`	Site section that the document belongs to. This will cause the associated item in the navigation bar to be highlighted. One of: `UserGuide`, `DeveloperGuide`, `LearningOutcomes`*, `AboutUs`, `ContactUs` ** Official SE-EDU projects only*	*not set*
`no-site-header`	Set this attribute to remove the site navigation bar.	*not set*

4.6. Site Template

The files in `docs/stylesheets` are the CSS stylesheets of the site. You can modify them to change some properties of the site's design.

The files in `docs/templates` controls the rendering of `.adoc` files into HTML5. These template files are written in a mixture of Ruby and Slim.

WARNING Modifying the template files in `docs/templates` requires some knowledge and experience with Ruby and Asciidoctor's API. You should only modify them if you need greater control over the site's layout than what stylesheets can provide. The SE-EDU team does not provide support for modified template files.

5. Testing

5.1. Running Tests

There are three ways to run tests.

TIP The most reliable way to run tests is the 3rd one. The first two methods might fail some GUI tests due to platform/resolution-specific idiosyncrasies.

Method 1: Using IntelliJ JUnit test runner

- To run all tests, right-click on the `src/test/java` folder and choose `Run 'All Tests'`
- To run a subset of tests, you can right-click on a test package, test class, or a test and choose `Run 'ABC'`

Method 2: Using Gradle

- Open a console and run the command `gradlew clean allTests` (Mac/Linux: `./gradlew clean allTests`)

NOTE See UsingGradle.adoc for more info on how to run tests using Gradle.

Method 3: Using Gradle (headless)

Thanks to the TestFX library we use, our GUI tests can be run in the *headless* mode. In the headless mode, GUI tests do not show up on the screen. That means the developer can do other things on the Computer while the tests are running.

To run tests in headless mode, open a console and run the command `gradlew clean headless allTests` (Mac/Linux: `./gradlew clean headless allTests`)

5.2. Types of tests

We have two types of tests:

1. **GUI Tests** - These are tests involving the GUI. They include,
   a. *System Tests* that test the entire App by simulating user actions on the GUI. These are in the `systemtests` package.
   b. *Unit tests* that test the individual components. These are in `seedu.address.ui` package.
2. **Non-GUI Tests** - These are tests not involving the GUI. They include,
   a. *Unit tests* targeting the lowest level methods/classes.
   e.g. `seedu.address.commons.StringUtilTest`
   b. *Integration tests* that are checking the integration of multiple code units (those code units are assumed to be working).
   e.g. `seedu.address.storage.StorageManagerTest`

c. Hybrids of unit and integration tests. These test are checking multiple code units as well as how the are connected together.
   e.g. `seedu.address.logic.LogicManagerTest`

5.3. Troubleshooting Testing

Problem: `HelpWindowTest` fails with a `NullPointerException`.

- Reason: One of its dependencies, `HelpWindow.html` in `src/main/resources/docs` is missing.
- Solution: Execute Gradle task `processResources`.

6. Dev Ops

6.1. Build Automation

See UsingGradle.adoc to learn how to use Gradle for build automation.

6.2. Continuous Integration

We use Travis CI and AppVeyor to perform *Continuous Integration* on our projects. See UsingTravis.adoc and UsingAppVeyor.adoc for more details.

6.3. Coverage Reporting

We use Coveralls to track the code coverage of our projects. See UsingCoveralls.adoc for more details.

6.4. Documentation Previews

When a pull request has changes to asciidoc files, you can use Netlify to see a preview of how the HTML version of those asciidoc files will look like when the pull request is merged. See UsingNetlify.adoc for more details.

6.5. Making a Release

Here are the steps to create a new release.

1. Update the version courseReqCredits in `MainApp.java`.
2. Generate a JAR file using Gradle.
3. Tag the repo with the version courseReqCredits. e.g. `v0.1`
4. Create a new release using GitHub and upload the JAR file you created.

6.6. Managing Dependencies

A project often depends on third-party libraries. For example, GradTrak depends on the Jackson library for JSON parsing. Managing these *dependencies* can be automated using Gradle. For example, Gradle can download the dependencies automatically, which is better than these alternatives:

a. Include those libraries in the repo (this bloats the repo size)

b. Require developers to download those libraries manually (this creates extra work for developers)

Appendix A: Suggested Programming Tasks to Get Started

Suggested path for new programmers:

1. First, add small local-impact (i.e. the impact of the change does not go beyond the component) enhancements to one component at a time. Some suggestions are given in Section A.1, "Improving each component".
2. Next, add a feature that touches multiple components to learn how to implement an end-to-end feature across all components. Section A.2, "Creating a new command: `remark`" explains how to go about adding such a feature.

A.1. Improving each component

Each individual exercise in this section is component-based (i.e. you would not need to modify the other components to get it to work).

`Logic` component

Scenario: You are in charge of `logic`. During dog-fooding, your team realize that it is troublesome for the user to type the whole command in order to execute a command. Your team devise some strategies to help cut down the amount of typing necessary, and one of the suggestions was to implement aliases for the command words. Your job is to implement such aliases.

TIP Do take a look at Section 2.3, "Logic component" before attempting to modify the `Logic` component.

1. Add a shorthand equivalent alias for each of the individual commands. For example, besides typing `clear`, the user can also type `c` to remove all modulesTaken in the list.

- Hints
  - Just like we store each individual command word constant `COMMAND_WORD` inside `*Command.java` (e.g. `FindCommand#COMMAND_WORD`, `DeleteCommand#COMMAND_WORD`), you need a new constant for aliases as well (e.g. `FindCommand#COMMAND_ALIAS`).
  - `AddressBookParser` is responsible for analyzing command words.
- Solution
  - Modify the switch statement in `AddressBookParser#parseCommand(String)` such that both the proper command word and alias can be used to execute the same intended command.
  - Add new tests for each of the aliases that you have added.
  - Update the user guide to document the new aliases.
  - See this PR for the full solution.

`Model` component

Scenario: You are in charge of `model`. One day, the `logic`-in-charge approaches you for help. He wants to implement a command such that the user is able to remove a particular tag from everyone in the address book, but the model API does not support such a functionality at the moment. Your job is to implement an API method, so that your teammate can use your API to implement his command.

TIP Do take a look at Section 2.4, “Model component” before attempting to modify the `Model` component.

1. Add a `removeTag(Tag)` method. The specified tag will be removed from everyone in the address book.

- Hints
  - The `Model` and the `GradTrak` API need to be updated.
  - Think about how you can use SLAP to design the method. Where should we place the main logic of deleting tags?
  - Find out which of the existing API methods in `GradTrak` and `ModuleTaken` classes can be used to implement the tag removal logic. `GradTrak` allows you to update a module, and `ModuleTaken` allows you to update the tags.
- Solution
  - Implement a `removeTag(Tag)` method in `GradTrak`. Loop through each module, and remove the `tag` from each module.
  - Add a new API method `deleteTag(Tag)` in `ModelManager`. Your `ModelManager` should call `GradTrak#removeTag(Tag)`.
  - Add new tests for each of the new public methods that you have added.
  - See this PR for the full solution.

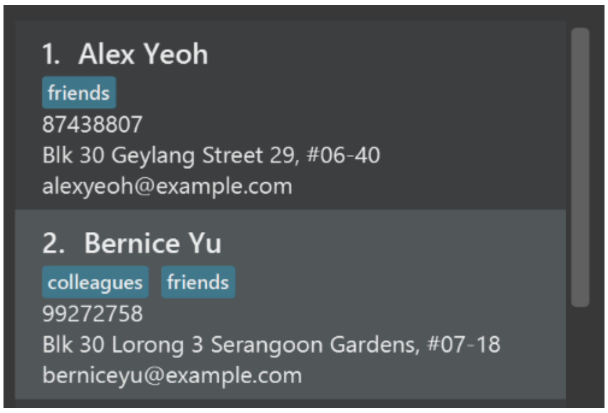
`Ui` component

Scenario: You are in charge of `ui`. During a beta testing session, your team is observing how the users use your address book application. You realize that one of the users occasionally tries to delete non-existent tags from a contact, because the tags all look the same visually, and the user got confused. Another user made a typing mistake in his command, but did not realize he had done so because the error message wasn't prominent enough. A third user keeps scrolling down the list, because he keeps forgetting the index of the last person in the list. Your job is to implement improvements to the UI to solve all these problems.

TIP Do take a look at Section 2.2, "UI component" before attempting to modify the `UI` component.

1. Use different colors for different tags inside person cards. For example, `friends` tags can be all in brown, and `colleagues` tags can be all in yellow.

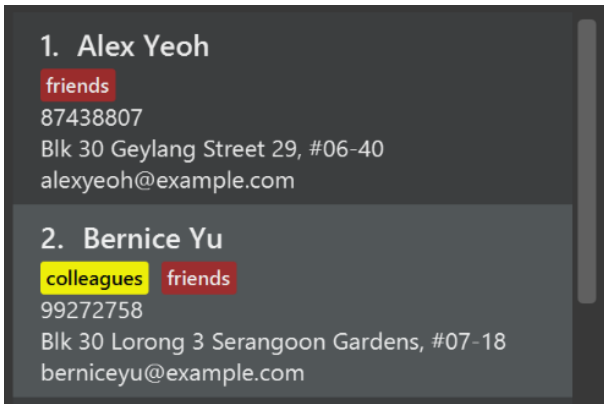
   Before



1. Alex Yeoh
friends
87438807
Blk 30 Geylang Street 29, #06-40
alexyeoh@example.com

2. Bernice Yu
colleagues friends
99272758
Blk 30 Lorong 3 Serangoon Gardens, #07-18
berniceyu@example.com

After



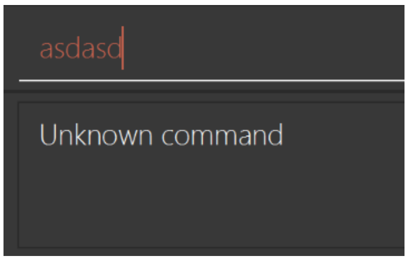
1. Alex Yeoh
friends
87438807
Blk 30 Geylang Street 29, #06-40
alexyeoh@example.com

2. Bernice Yu
colleagues friends
99272758
Blk 30 Lorong 3 Serangoon Gardens, #07-18
berniceyu@example.com

- Hints
  - The tag labels are created inside the `PersonCard` constructor (`new Label(tag.tagName)`). JavaFX's `Label` class allows you to modify the style of each Label, such as changing its color.
  - Use the .css attribute `-fx-background-color` to add a color.
  - You may wish to modify `DarkTheme.css` to include some pre-defined colors using css, especially if you have experience with web-based css.
- Solution
  - You can modify the existing test methods for `PersonCard` 's to include testing the tag's color as well.
  - See this PR for the full solution.
    - The PR uses the hash code of the tag names to generate a color. This is deliberately designed to ensure consistent colors each time the application runs. You may wish to expand on this design to include additional features, such as allowing users to set their own tag colors, and directly saving the colors to storage, so that tags retain their colors even if the hash code algorithm changes.

2. Modify `NewResultAvailableEvent` such that `ResultDisplay` can show a different style on error (currently it shows the same regardless of errors).

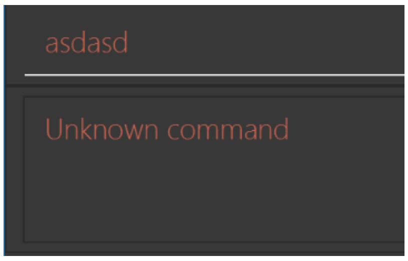
Before



asdasd

Unknown command

After

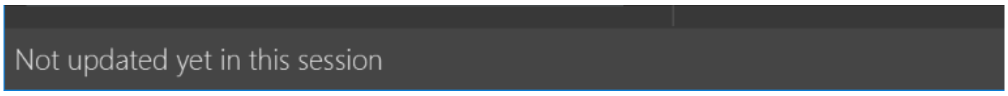

asdasd

Unknown command

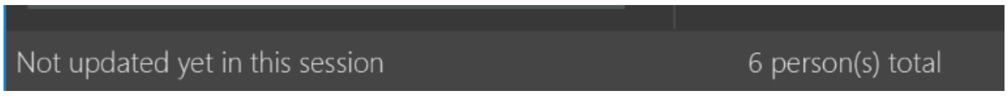
- Hints
  - NewResultAvailableEvent is raised by CommandBox which also knows whether the result is a success or failure, and is caught by ResultDisplay which is where we want to change the style to.
  - Refer to CommandBox for an example on how to display an error.
- Solution
  - Modify NewResultAvailableEvent 's constructor so that users of the event can indicate whether an error has occurred.
  - Modify ResultDisplay#handleNewResultAvailableEvent(NewResultAvailableEvent) to react to this event appropriately.
  - You can write two different kinds of tests to ensure that the functionality works:
    - The unit tests for ResultDisplay can be modified to include verification of the color.
    - The system tests AddressBookSystemTest#assertCommandBoxShowsDefaultStyle() and AddressBookSystemTest#assertCommandBoxShowsErrorStyle() to include verification for ResultDisplay as well.
  - See this PR for the full solution.
    - Do read the commits one at a time if you feel overwhelmed.

3. Modify the StatusBarFooter to show the total courseReqCredits of people in the address book.

Before


Not updated yet in this session

After


Not updated yet in this session 6 person(s) total

- Hints
  - `StatusBarFooter.fxml` will need a new `StatusBar`. Be sure to set the `GridPane.columnIndex` properly for each `StatusBar` to avoid misalignment!
  - `StatusBarFooter` needs to initialize the status bar on application start, and to update it accordingly whenever the address book is updated.
- Solution
  - Modify the constructor of `StatusBarFooter` to take in the courseReqCredits of modulesTaken when the application just started.
  - Use `StatusBarFooter#handleGradTrakChangedEvent(GradTrakChangedEvent)` to update the courseReqCredits of modulesTaken whenever there are new changes to the GradTrak.
  - For tests, modify `StatusBarFooterHandle` by adding a state-saving functionality for the total courseReqCredits of people status, just like what we did for save location and sync status.
  - For system tests, modify `GradTrakSystemTest` to also verify the new total courseReqCredits of modulesTaken status bar.
  - See this PR for the full solution.

`Storage` component

Scenario: You are in charge of `storage`. For your next project milestone, your team plans to implement a new feature of saving the GradTrak to the cloud. However, the current implementation of the application constantly saves the GradTrak after the execution of each command, which is not ideal if the user is working on limited internet connection. Your team decided that the application should instead save the changes to a temporary local backup file first, and only upload to the cloud after the user closes the application. Your job is to implement a backup API for the GradTrak storage.

TIP Do take a look at Section 2.5, “Storage component” before attempting to modify the `Storage` component.

1. Add a new method `backupGradTrak(ReadOnlyGradTrak)`, so that the address book can be saved in a fixed temporary location.

   - Hint
     - Add the API method in `GradTrakStorage` interface.
     - Implement the logic in `StorageManager` and `JsonGradTrakStorage` class.
   - Solution
     - See this PR for the full solution.

A.2. Creating a new command: `remark`

By creating this command, you will get a chance to learn how to implement a feature end-to-end, touching all major components of the app.

Scenario: You are a software maintainer for `GradTrak`, as the former developer team has moved on to new projects. The current users of your application have a list of new feature requests that they hope the software will eventually have. The most popular request is to allow adding additional comments/notes about a particular module, by providing a flexible `remark` field for each module, rather than relying on tags alone. After designing the specification for the `remark` command, you are convinced that this feature is worth implementing. Your job is to implement the `remark` command.

A.2.1. Description

Edits the remark for a person specified in the `INDEX`.
Format: `remark INDEX r/[REMARK]`

Examples:

- `remark 1 r/Very difficult!`
  Edits the remark for the first module to `Very difficult!`
- `remark 1 r/`
  Removes the remark for the first module.

A.2.2. Step-by-step Instructions

[Step 1] Logic: Teach the app to accept 'remark' which does nothing

Let’s start by teaching the application how to parse a `remark` command. We will add the logic of `remark` later.

Main:

1. Add a `RemarkCommand` that extends `Command`. Upon execution, it should just throw an `Exception`.
2. Modify `GradTrakParser` to accept a `RemarkCommand`.

Tests:

1. Add `RemarkCommandTest` that tests that `execute()` throws an Exception.
2. Add new test method to `GradTrakTest`, which tests that typing "remark" returns an instance of `RemarkCommand`.

[Step 2] Logic: Teach the app to accept 'remark' arguments

Let’s teach the application to parse arguments that our `remark` command will accept. E.g. `1 r/Very Difficult!`

Main:

1. Modify `RemarkCommand` to take in an `Index` and `String` and print those two parameters as the error message.
2. Add `RemarkCommandParser` that knows how to parse two arguments, one index and one with prefix 'r/'.
3. Modify `GradTrakParser` to use the newly implemented `RemarkCommandParser`.

Tests:

1. Modify `RemarkCommandTest` to test the `RemarkCommand#equals()` method.
2. Add `RemarkCommandParserTest` that tests different boundary values for `RemarkCommandParser`.
3. Modify `GradTrakParserTest` to test that the correct command is generated according to the user input.

[Step 3] Ui: Add a placeholder for remark in `PersonCard`

Let's add a placeholder on all our `PersonCard` s to display a remark for each person later.

Main:

1. Add a `Label` with any random text inside `PersonListCard.fxml`.
2. Add FXML annotation in `PersonCard` to tie the variable to the actual label.

Tests:

1. Modify `PersonCardHandle` so that future tests can read the contents of the remark label.

[Step 4] Model: Add `Remark` class

We have to properly encapsulate the remark in our `Person` class. Instead of just using a `String`, let's follow the conventional class structure that the codebase already uses by adding a `Remark` class.

Main:

1. Add `Remark` to model component (you can copy from `Address`, remove the regex and change the names accordingly).
2. Modify `RemarkCommand` to now take in a `Remark` instead of a `String`.

Tests:

1. Add test for `Remark`, to test the `Remark#equals()` method.

[Step 5] Model: Modify `ModuleTaken` to support a `Remark` field

Now we have the `Remark` class, we need to actually use it inside `ModuleTaken`.

Main:

1. Add `getRemark()` in `ModuleTaken`.
2. You may assume that the user will not be able to use the `add` and `edit` commands to modify the

remarks field (i.e. the module will be created without a remark).

3. Modify `SampleDataUtil` to add remarks for the sample data (delete your `data/gradtrak.json` so that the application will load the sample data when you launch it.)

[Step 6] Storage: Add `Remark` field to `JsonAdaptedModuleTaken` class

We now have `Remark` s for `ModuleTaken` s, but they will be gone when we exit the application. Let’s modify `JsonModuleTaken` to include a `Remark` field so that it will be saved.

Main:

1. Add a new JSON field for `Remark`.

Tests:

1. Fix `invalidAndValidModuleTakenGradTrak.json`, `typicalModuleTakenGradTrak.json`, `validGradTrak.json` etc., such that the JSON tests will not fail due to a missing `remark` field.

[Step 6b] Test: Add withRemark() for `ModuleTaken`

Since `ModuleTaken` can now have a `Remark`, we should add a helper method to `ModuleTakenBuilder`, so that users are able to create remarks when building a `ModuleTaken`.

Tests:

1. Add a new method `withRemark()` for `ModuleTakenBuilder`. This method will create a new `Remark` for the module that it is currently building.
2. Try and use the method on any sample `ModuleTaken` in `TypicalModuleTaken`.

[Step 7] Ui: Connect `Remark` field to `PersonCard`

Our remark label in `PersonCard` is still a placeholder. Let’s bring it to life by binding it with the actual `remark` field.

Main:

1. Modify `PersonCard`'s constructor to bind the `Remark` field to the `ModuleTaken` 's remark.

Tests:

1. Modify `GuiTestAssert#assertCardDisplaysPerson(…)` so that it will compare the now-functioning remark label.

[Step 8] Logic: Implement `RemarkCommand#execute()` logic

We now have everything set up… but we still can’t modify the remarks. Let’s finish it up by adding in actual logic for our `remark` command.

Main:

1. Replace the logic in `RemarkCommand#execute()` (that currently just throws an `Exception`), with the

actual logic to modify the remarks of a person.

Tests:

1. Update `RemarkCommandTest` to test that the `execute()` logic works.

A.2.3. Full Solution

See this PR for the step-by-step solution.

Appendix B: Product Scope

Target user profile:

- has a need to manage a significant courseReqCredits of contacts
- prefer desktop apps over other types
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition: manage contacts faster than a typical mouse/GUI driven app

Appendix C: User Stories

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

Priority	As a …	I want to …	So that I can…
`* * *`	student	track the modules I am taking	know what I need to complete my graduation requirement
`* * *`	new user	see usage instructions	refer to instructions when I forget how to use the App
`* * *`	student	add a module for the current semester	
`* * *`	student	delete a module	remove modules that I am not taking

Priority	As a …	I want to …	So that I can…
* * *	student	find a module by code, semester, grade or finished status	locate details of modules without having to go through the entire list
* *	student	hide private contact details by default	minimize chance of someone else seeing them by accident
* *	student	view pre-requisites for a module	take the pre-requisites ahead of time
* *	student	add my modules in future semesters	plan ahead
* *	student	know my CAP	pull up my CAP
* *	student	see a recommended list of modules I can take in order of priority	fulfil the graduation requirements on time
*	student with many modules	sort module by name	locate a module easily

{More to be added}

Appendix D: Use Cases

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

Use case: Initialising

Precondition: User uses GT for the first time

MSS

1. User starts GT.

2. GT displays courses.
3. GT prompts for user to set choice of course.
4. User selects course.
5. GT sets choice of course by user.

   Use case ends.

Extensions

   4a. User selects course that has specialisation.

      4a1. GT displays specialisation.

      4a2. GT prompts user to select specialisation.

      4a3. GT sets choice of course and specialisation by user.

      Use case ends.

Use case: Adding to completed list of modules

Precondition: Student has already initialised GT

MSS

1. User enters command to add modules with module code, semester completed and grades.
2. System records module code, semester completed and grades.

   Use case ends.

Extensions

   1a. Student enters invalid grade.

      1a1. GT prompts user that grade is invalid.

      Use case resumes at step 1.

   1b. Student enters invalid module code.

      1b1. GT prompts user that module code is invalid.

      Use case resumes at step 1.

   1c. Student enters invalid semester.

      1c1. GT prompts user that semester is invalid.

      Use case resumes at step 1.

   1d. Student enters module that is already in list of completed modules.

1d1. GT prompts user that there is a repeat of module code.

Use case ends.

Use case: Delete module

Precondition: Student has already intialised GT

MSS

1. User enters command to remove modules by giving module code.
2. System removes module from list of completed / planned modules.

   Use case ends.

Extensions

1a. Student enters module code that is correct but not in list of modules completed or planned.

Use case ends.

1b. Student enters module code that is wrong.

1b1. GT shows an error message.

Use case resumes at step 1.

Use case: Modify details of modules taken

MSS

1. User enters command to edit modules indicated by module code and gives grades and semester taken / planning to take.
2. System edits relevant details.

   Use case ends.

Extensions

1a. Module code correct but not in the list of modules completed or planned.

1a1. GT prompts user that module is not being taken.

Use case ends.

1b. Student enters invalid module code.

1b1. GT shows an error message.

Use case ends.

Use case: Display module info

MSS

1. Student keys command to find module code or keywords in the title of module.
2. System returns module given module code or keywords.

   Use case ends.

{More to be added}

Appendix E: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java `9` or higher installed.
2. Should be able to hold up to 1000 modulesTaken 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.

{More to be added}

Appendix F: Glossary

Mainstream OS
: Windows, Linux, Unix, OS-X

Private contact detail
: A contact detail that is not meant to be shared with others

Appendix G: Product Survey

Product Name

Author: …

Pros:

- …
- …

Cons:

- …
- …

Appendix H: 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.

H.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 a set of sample contacts. The window size may not be optimum.
2. Saving 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: The most recent window size and location is retained.

{ more test cases … }

H.2. Deleting a person

1. Deleting a person while all modulesTaken are listed
   a. Prerequisites: List all modulesTaken using the `list` command. Multiple modulesTaken in the list.
   b. Test case: `delete 1`
      Expected: First contact is deleted from the list. Details of the deleted contact shown in the status message. Timestamp in the status bar is updated.
   c. Test case: `delete 0`
      Expected: No person is deleted. Error details shown in the status message. Status bar remains the same.
   d. Other incorrect delete commands to try: `delete`, `delete x` (where x is larger than the list size) *{give more}*
      Expected: Similar to previous.

{ more test cases … }

H.3. Saving data

1. Dealing with missing/corrupted data files

a. *{explain how to simulate a missing/corrupted file and the expected behavior}*

{ more test cases … }