iGrad - Developer Guide

Table of Contents

By: `AY1920S2-CS2103T-F09-3` Since: `Feb 2020` Licence: `MIT`

1. Setting up

Refer to the guide here.

2. Design

2.1. Architecture

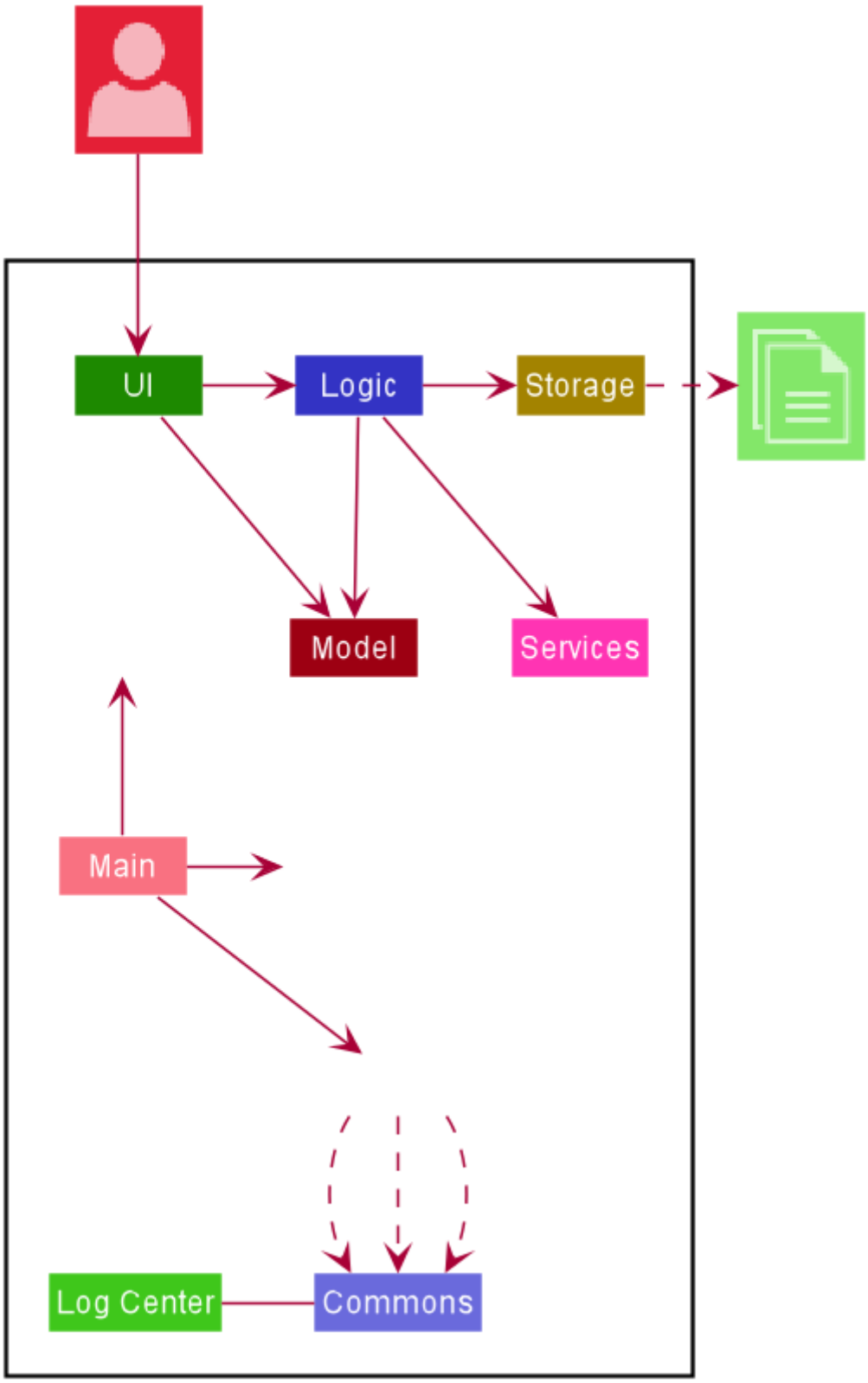


Figure 1. Architecture Diagram

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

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

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

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

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

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

The rest of the App consists of five components.

- `UI`: The UI of the App.
- `Logic`: The command executor.
- `Model`: Holds the data of the App in-memory.
- `Storage`: Reads data from, and writes data to, the hard disk.
- `Services`: Interacts with an external Application Programming Interface (API) to obtain data for the App.

Each of the first four components

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

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

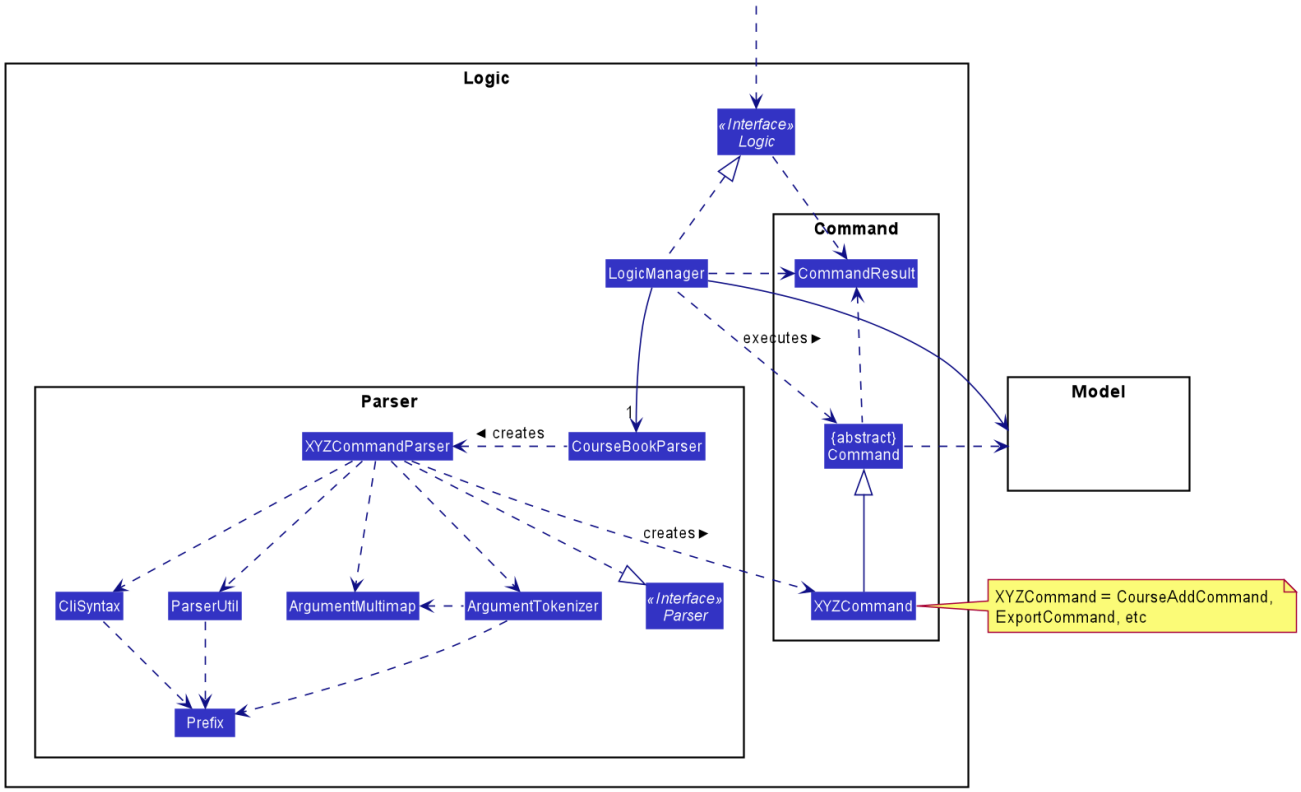


Figure 2. Class Diagram of the Logic Component

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`.

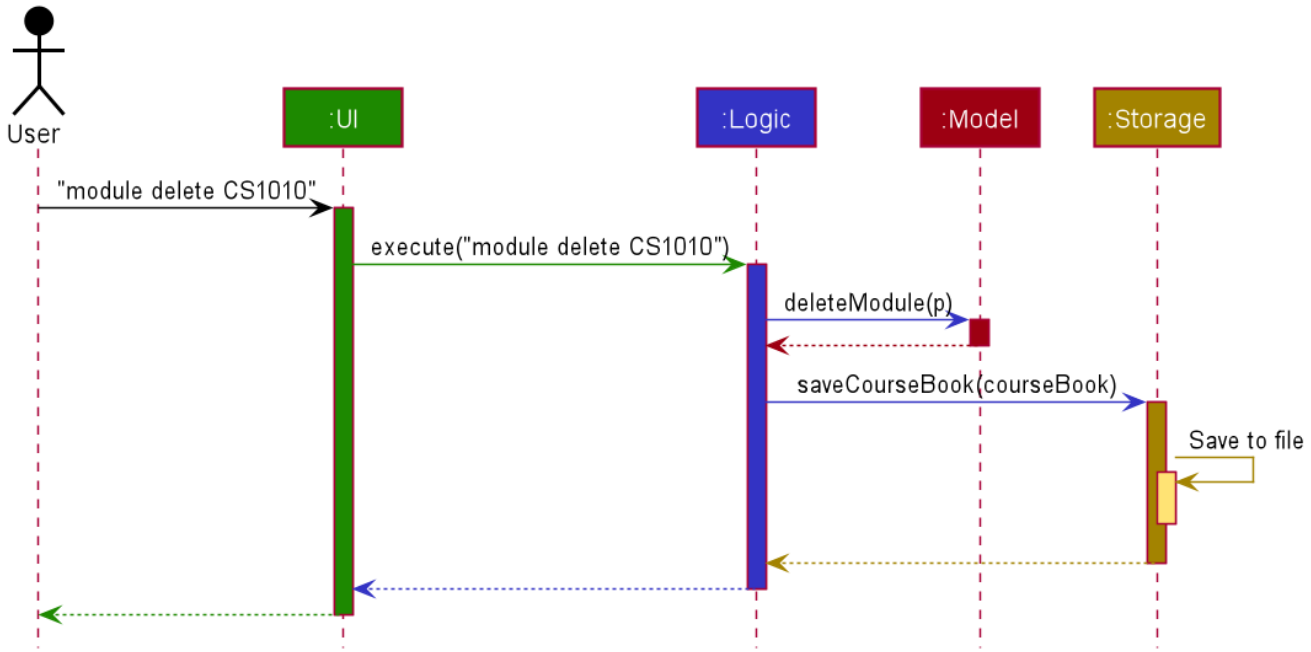


Figure 3. Component interactions for `module delete CS1010` command

The sections below give more details of each component.

2.2. UI component

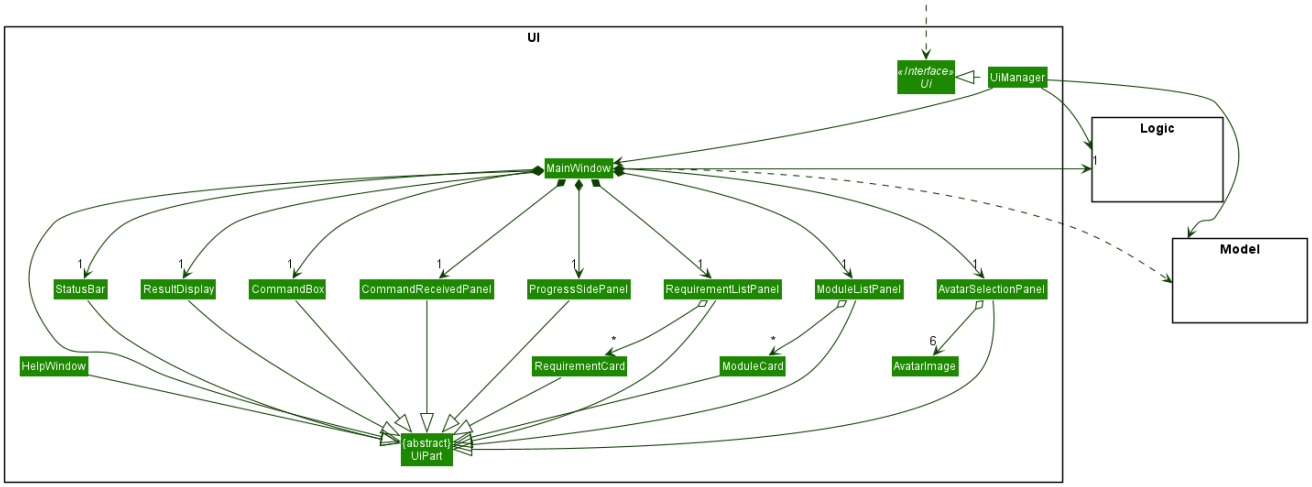


Figure 4. Structure of the UI Component

API : `Ui.java`

Section by: Daryl

The UI consists of a `MainWindow` that is made up of parts e.g.`CommandBox`, `ResultDisplay`, `ModuleListPanel`, `StatusBar` etc. All these, including the `MainWindow` (excluding `AvatarImage`), 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.

2.2.1. MainWindow

The `MainWindow` class serves as the hub for all the UI components, and contains the following UI classes:

- `AvatarSelectionPanel` - Avatar selection screen on first-time startup.
- `CommandBox` - Command box for user input.
- `CommandReceivedPanel` - Displays the last command entered.
- `ResultDisplay` - Displays the resultant message of the command entered. Also contains the avatar image.
- `ModuleListPanel` - Panel displaying the modules input into the system.
- `RequirementListPanel` - Panel displaying the requirements input into the system.
- `ProgressSidePanel` - Panel displaying the user's academic progress and fundamental details (Eg. CAP).

The `MainWindow` coordinates the development between the backend and frontend components to induce a visible change to the interface.

This is done through the `executeCommand(String commandText, Model model)` method. Upon user input, the `logic` class executes the command in `commandText`, and the model is updated to reflect the changes. Subsequently, after the model has been updated, the following UI classes `ResultDisplay` and `ProgressSidePanel` are refreshed as a result.

2.2.2. AvatarSelectionPanel

The `AvatarSelectionPanel` class displays the avatar selection screen upon first-time startup. Users will choose an that will act as a guide throughout their usage of the application.

2.2.3. CommandBox

The `CommandBox` class contains an editable `TextArea` JavaFX component which allows the user to enter input commands.

2.2.4. CommandReceivedPanel

The `CommandReceivedPanel` class contains a panel that shows the last command entered into the system.

Here is an example of how the `CommandReceivedPanel` works:

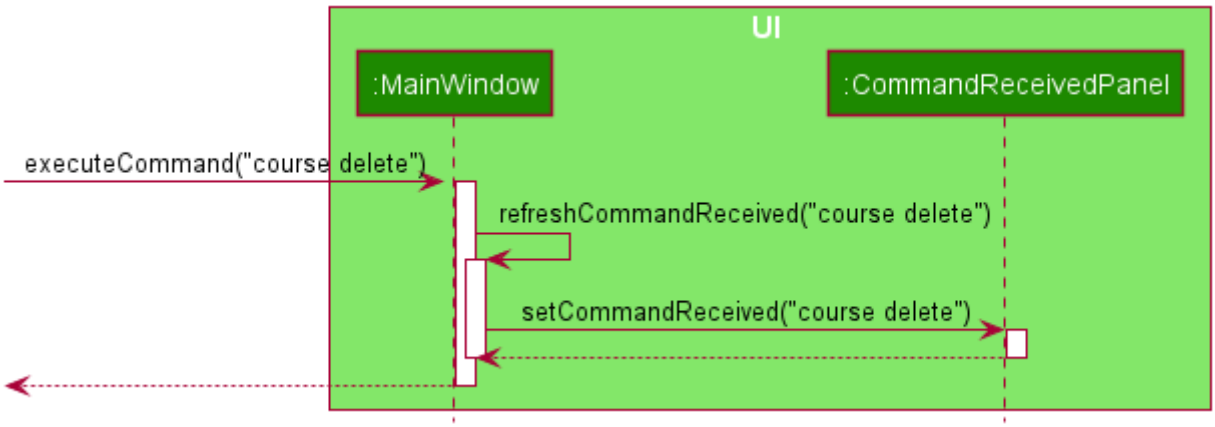


Figure 5. CommandReceivedPanel Updating from Received Command

1. Command executed on `MainWindow`.
2. `MainWindow` calls the method `refreshCommandReceivedPanel`, which refreshes the `CommandReceivedPanel`.
3. `CommandReceivedPanel` updates its JavaFX `Label` with the `String` of the command given.
4. `CommandReceivedPanel` displays visible change on the interface.
5. `refreshCommandReceivedPanel` ends execution.

2.2.5. ResultDisplay

The `ResultDisplay` class shows the resultant message generated from the user's input. The avatar will also showcase a different expression according to the success of the command given.

Here is an example of how the `ResultDisplay` works:

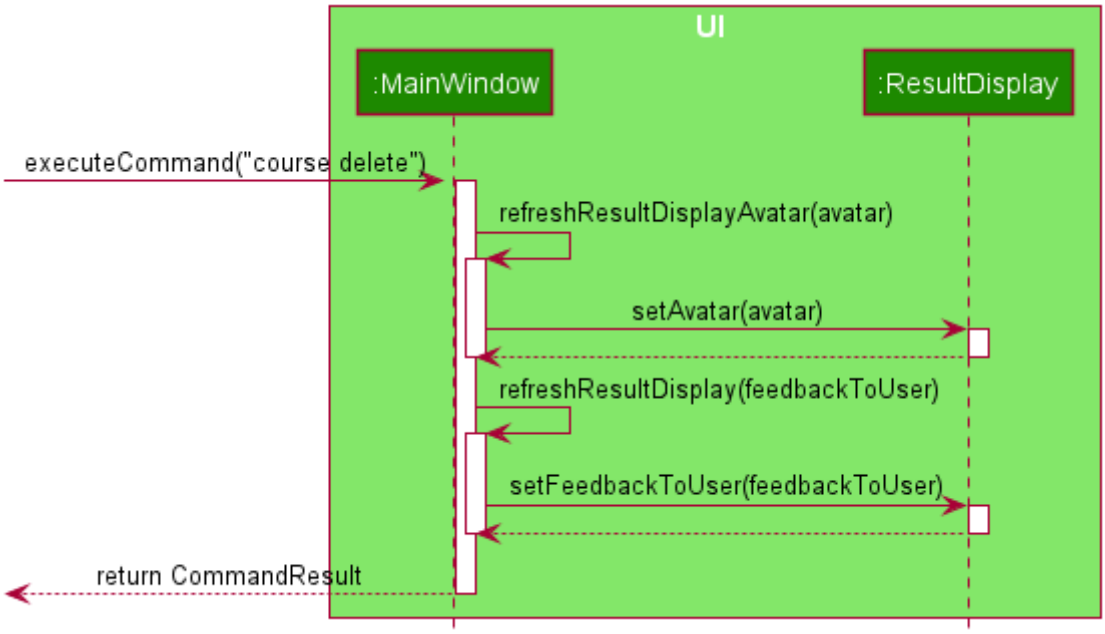


Figure 6. ResultDisplay Updating from Received Command

1. Command executed on `MainWindow`.

2. `MainWindow` calls the method `refreshResultDisplayAvatar`, which refreshes the `Avatar` in `ResultDisplay`.
3. `ResultDisplay` updates its JavaFX `ImageView` according to the `Avatar` of the command given. In this case, when no exception is thrown, the `Avatar` displays that of a positive expression.
4. `refreshResultDisplayAvatar` ends execution.
5. `ResultDisplay` displays visible change on the interface.
6. `MainWindow` calls the method `refreshResultDisplay`, which refreshes the resultant message displayed in `ResultDisplay`.
7. `ResultDisplay` updates its JavaFX `TextArea` according to the `CommandResult` of the command given. In this case, the TextArea will display the 'success' message generated as a result of the command.
8. `ResultDisplay` displays visible change on the interface.
9. `refreshResultDisplay` ends execution.

2.2.6. ModuleListPanel

The `ModuleListPanel` class contains the `ObservableList<Module>` JavaFX component allowing for a list view of the components inside it, in this case, a list of `ModuleCard` objects.

The contents of the list are dependent on the `modules` that the user has input into the system. Each module will be displayed as a `ModuleCard` object.

2.2.7. RequirementListPanel

The `RequirementListPanel` class contains the `ObservableList<Requirement` JavaFX component allowing for a list view of the components inside it, in this case, a list of `RequirementCard` objects.

The contents of the list are dependent on the `requirements` that the user has input into the system. Each requirement will be displayed as a `RequirementCard` object.

2.2.8. ProgressSidePanel

The `ProgressSidePanel` class contains the user's academic progress, as well as essential information. The following information is displayed on the `ProgressSidePanel`:

- `Course` name
- Inspiring quote from `QuoteGenerator`
- Modular Credits Progress Indicator
- `Semesters` left
- Current Cumulative Average Point (C.A.P)

Here is an example of how the `ProgressSidePanel` works:

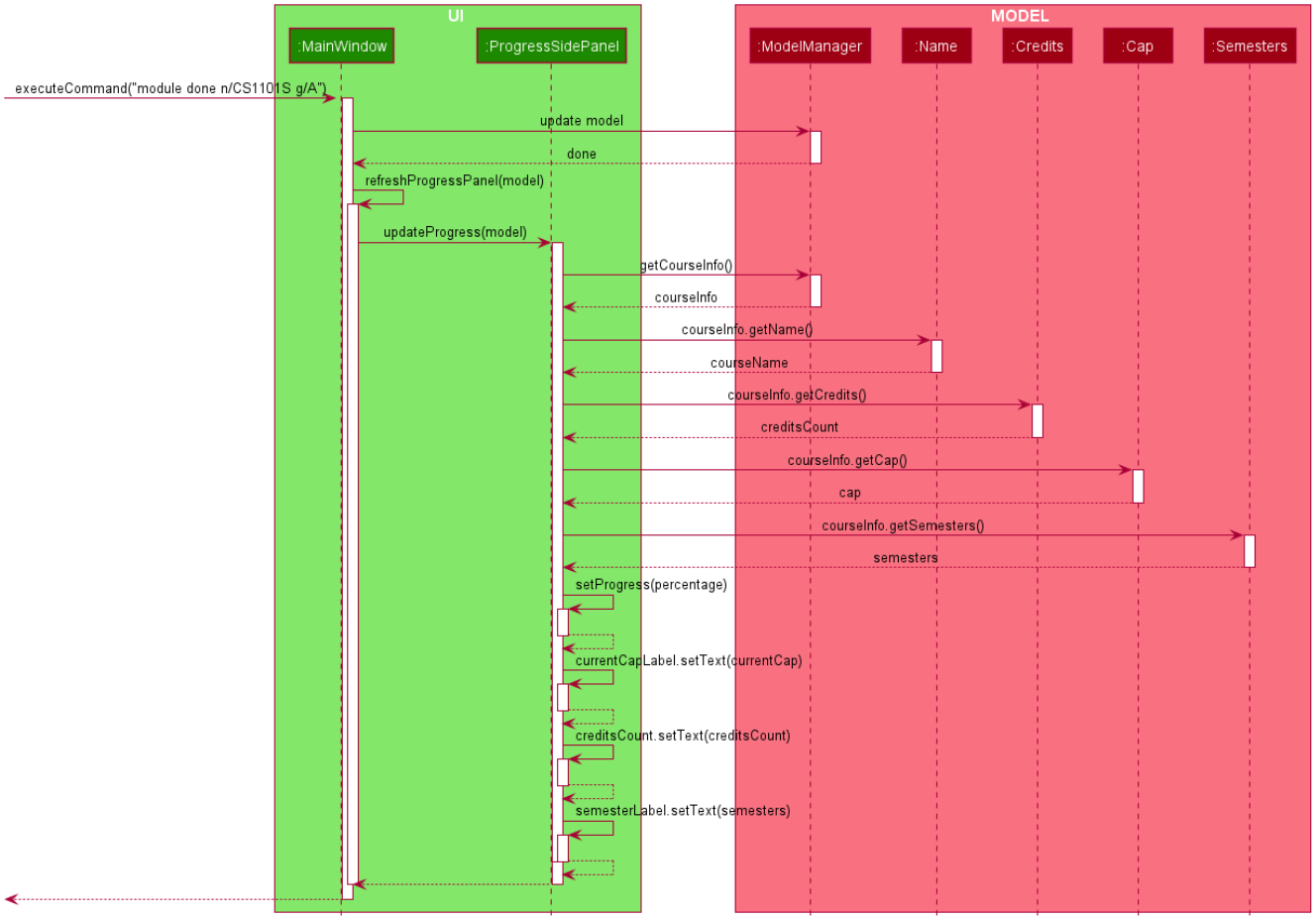


Figure 7. ProgressSidePanel Updating from Received Command

1. Command executed on `MainWindow`.
2. `Model` is updated.
3. `MainWindow` calls the method `refreshProgressPanel`, which refreshes the `ProgressSidePanel`.
4. `ProgressSidePanel` uses `Model` to obtain the corresponding `CourseInfo` information:
    a. `Name`
    b. `Credits`
    c. `Cap`
    d. `Semesters`
5. `ProgressSidePanel` executes corresponding JavaFX methods to update displayed information.
6. `ProgressSidePanel` shows visible change on the interface.
7. `refreshProgressPanel` ends execution.

2.2.9. Other Components

In addition to the main UI components grouped in the `MainWindow` class, these are the other UI components that are relevant to the interface:

- `AvatarImage` - Contains the image of the avatar.
- `ModuleCard` - Individual card containing the relevant information of the module. List of `ModuleCard` contained in the `ModuleListPanel`.

- `RequirementCard` - Individual card containing the relevant information of the requirement. List of `RequirementCards` contained in the `RequirementListPanel`.
- `HelpWindow` - Pop-up window containing the link the User Guide, as well as a list of all the commands in the application.

2.3. Logic component

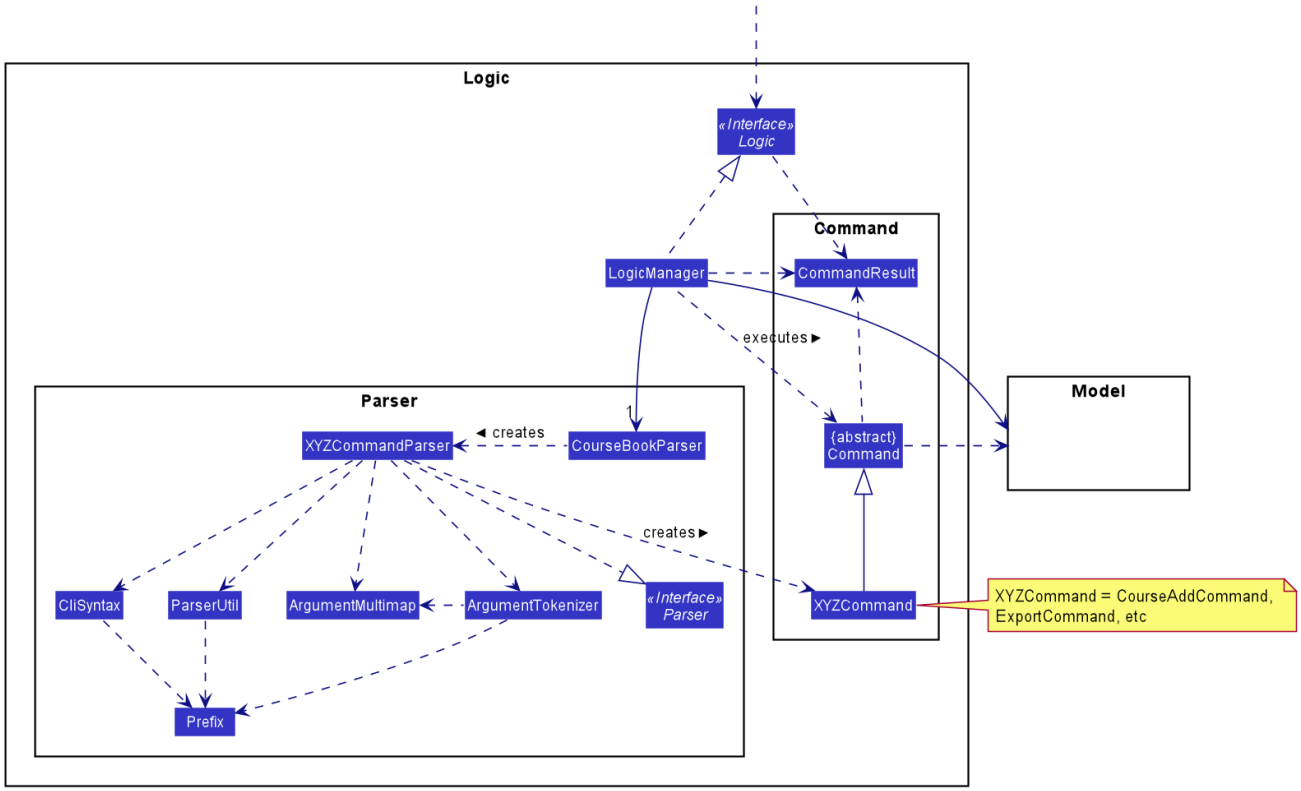


Figure 8. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `CourseBookParser` 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.

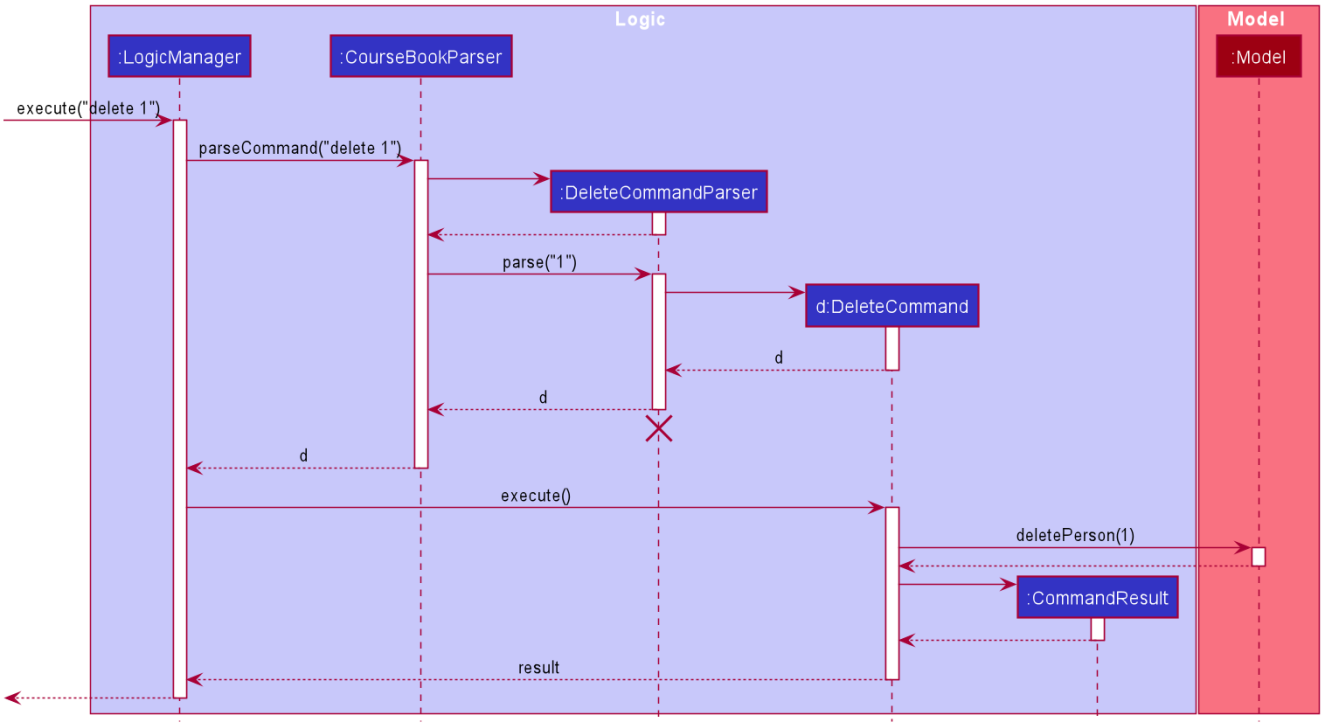


Figure 9. Interactions Inside the Logic Component for the `delete 1` Command

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

2.4. Model component

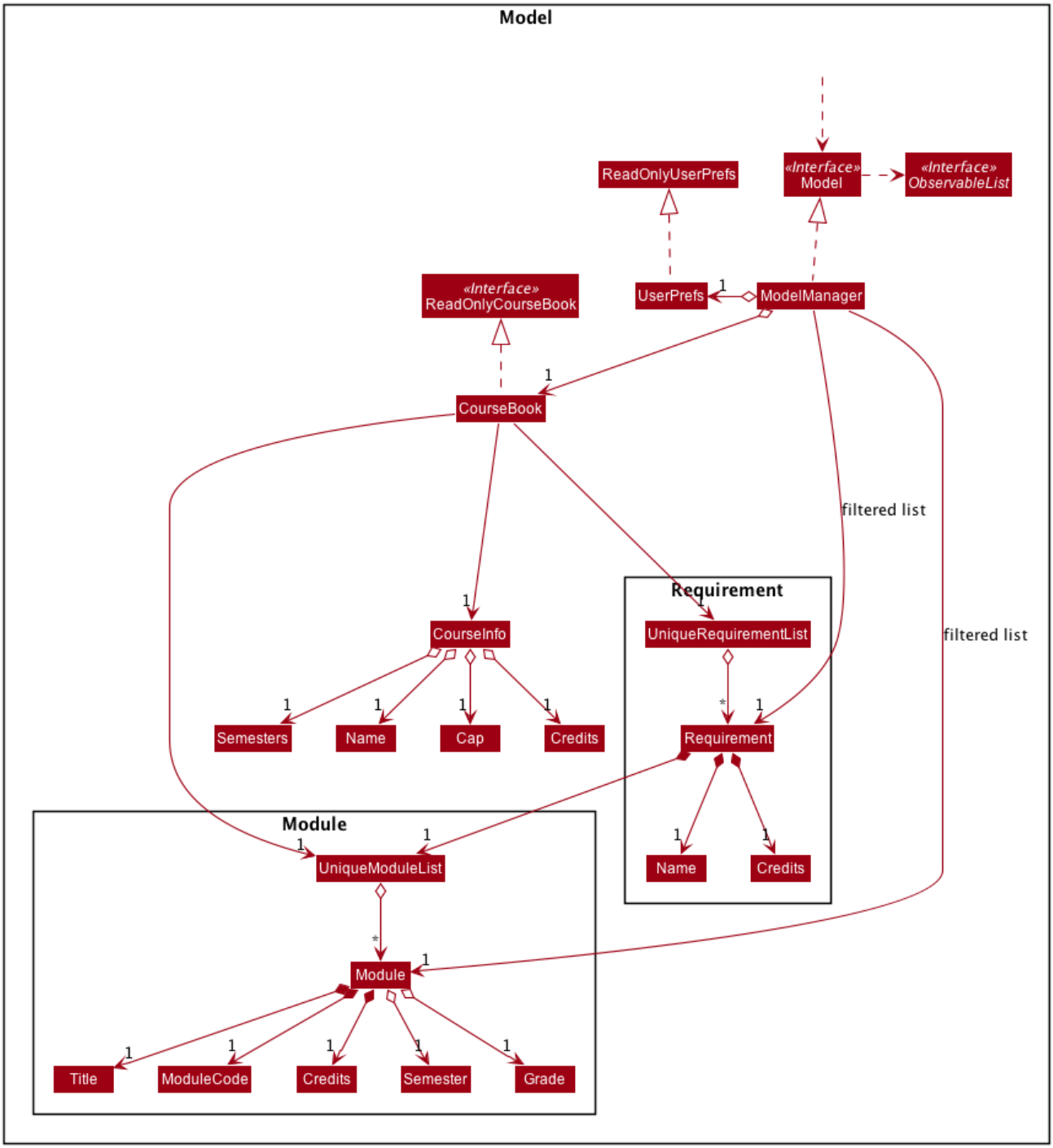


Figure 10. Structure of the Model Component

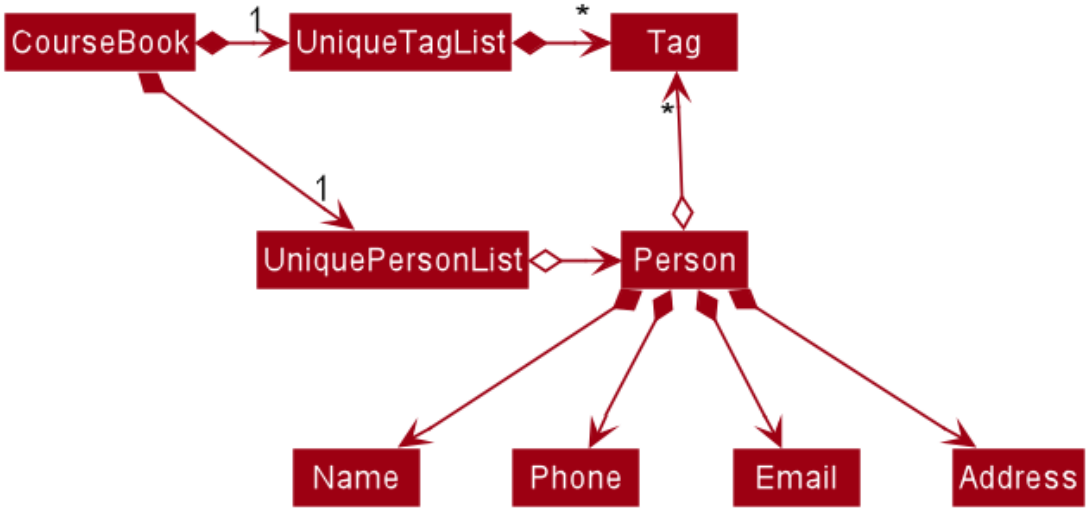
API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user's preferences.
- stores the Course Book data.
- exposes an unmodifiable `ObservableList<Requirement>` 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.
- exposes an unmodifiable `ObservableList<Module>` that can be 'observed'.
- does not depend on any of the other three components.

NOTE

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



2.5. Storage component

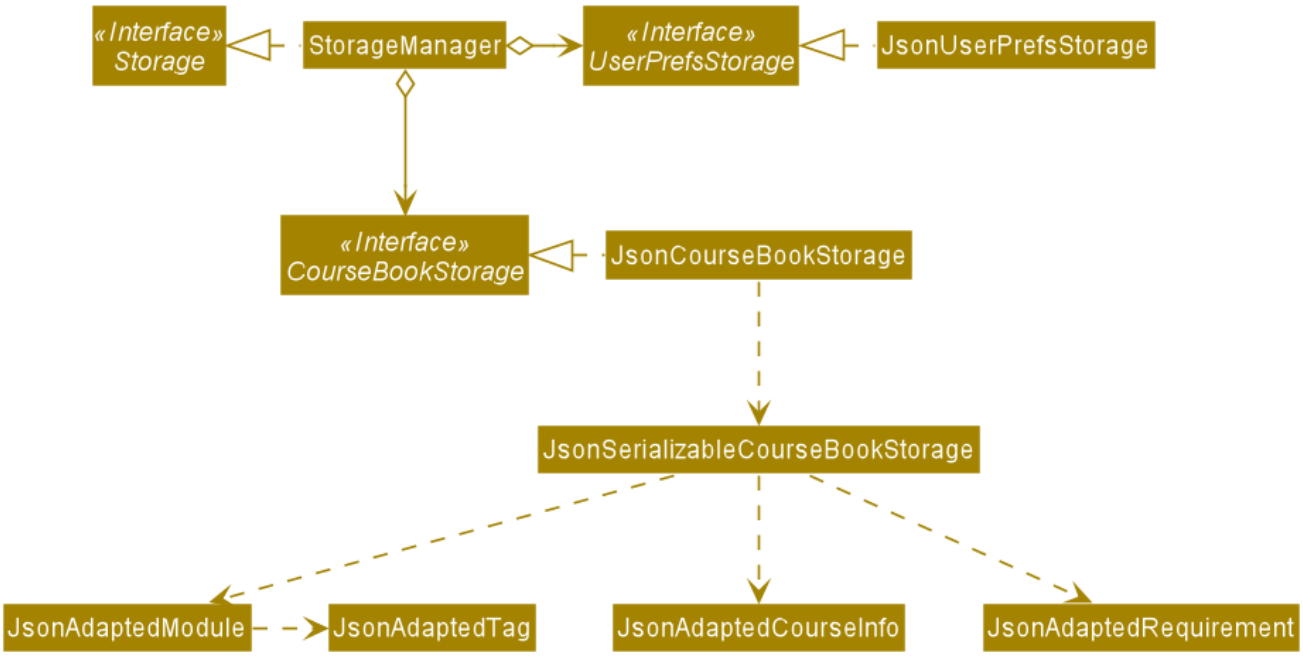


Figure 11. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

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

2.6. Common classes

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

3. Implementation

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

3.1. Course Feature

Section by: Nathanael Seen

As per the Model diagram above, there is only one `CourseBook` in the system.

A `CourseBook` represents all information related to helping a user track her graduation requirements, including the following:

- One `UniqueModuleList`, consisting of all `Modules` in the system, which may or may not be mapped to any (degree) `Requirement(s)`
- One `UniqueRequirementList`, consisting of all (degree) `Requirements`
- One `CourseInfo`, representing important information related to a degree course, which would be detailed more in this section

3.1.1. Implementation

Section by: Nathanael Seen

In the implementation of the course feature which 'houses' the various `Requirements` and the `Modules` mapped under those requirements, a `CourseInfo` class is necessary in order to represent overall course information, such as the name of the course, the current cap of the student (or user), the total credits (MCs) fulfilled/required, and semesters left before she could graduate.

These important information are encapsulated in the `CourseInfo` class which should only have one `Name`, one `Cap`, one `Credits` and one `Semesters` object(s), at any one time:

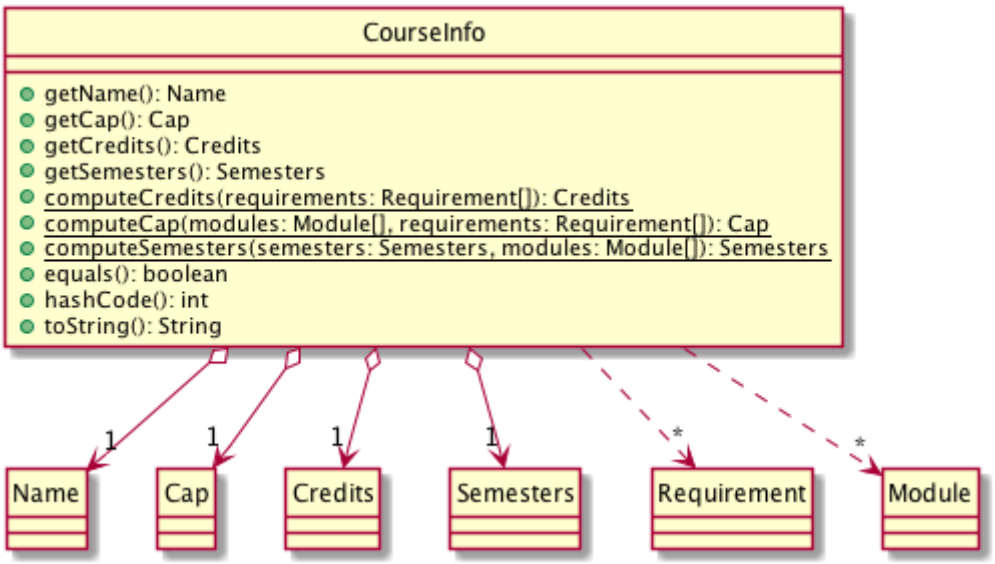


Figure 12. Structure of CourseInfo Class

Also, as per the diagram above, we note that these fields (in a `CourseInfo`) are optional, because a user might not even have a course set in the first place. This occurs when the application is started out in a 'blank' state, with no initial or sample data.

Now, to describe more of this `CourseInfo` class, its fields, the following two sections would detail the `Credits` and `Semesters` classes and their design.

Thereafter, the next two sections would attempt to explain the mechanics of the two crucial static methods `computeCredits(…)` and `computeCap(…)`, which is used throughout the application.

Finally, the last few sections would be dedicated to elaborating how the various course commands; `course set`, `course edit`, `course achieve`, and `course delete` works.

Credits

Section by: Nathanael Seen

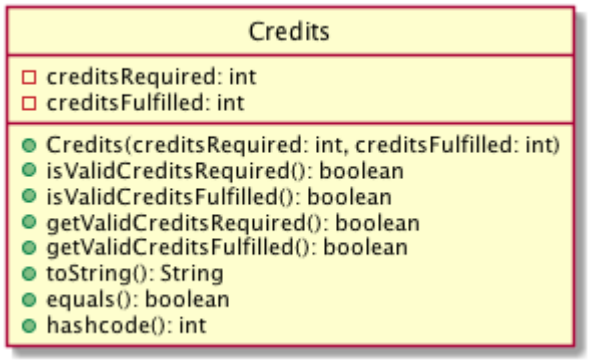


Figure 13. Course Credits Class Diagram

The `Credits` class maintains 2 integers; `creditsRequired` and `creditsFulfilled`, for storing both the total number of course credits (MCs) required for graduation, and also the number credits the user has fulfilled thus far, respectively.

Also, it some `public` validation methods (`isValidCreditsFulfilled()` and `isValidCreditsFulfilled()`) which is used in the constructor for constructing a 'valid' `Credits` object.

The following constraints defines a 'valid' `Credits` object:

- `creditsRequired` > 0
- `creditsFulfilled` >= 0

Note that `creditsFulfilled` can be **more than or equals** to `creditsRequired`, as it is possible that a student 'over-fulfills' the graduation requirements in her course.

The following are some noteworthy details on the `Credits` class/object:

- `Credits` is recomputed through the `computeCredits(…)` method in `CourseInfo` (whenever there is a possible change). This newly recomputed `Credits` object would be subsequently updated in the `CourseInfo`

- `creditsRequired` is recomputed by summing all the `creditsRequired` of the individual `Requirements` in the `UniqueRequirementList`
- `creditsFulfilled` is recomputed in the same way as `creditsFulfilled`
- In the `CourseInfo` class, `Credits` is first initialized (to some non-empty value) only when the following conditions have been met:
  - The user has already set a course, through the `course set` command.
  - There is at least one `Requirement` in the `UniqueRequirementList`, by which `creditsRequired` and `creditsFulfilled` could be re/computed.

Semesters

Section by: Teri

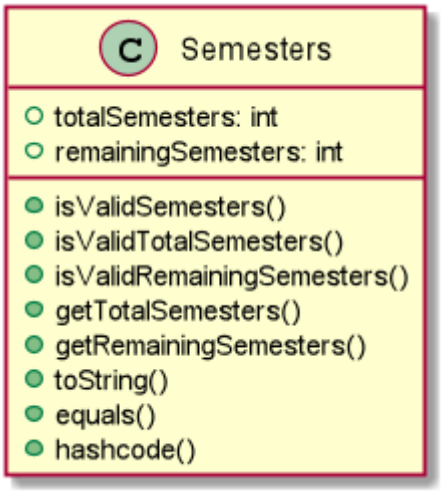


Figure 14. Course Semesters Class Diagram

`Semesters` stores the total semesters and remaining semesters that a user has in the course.

The following are some noteworthy details on the `Semesters` class/object:

- `Semesters` is first initialized when user does command `course set`. `totalSemesters` will be equal to `remainingSemesters` as user has not entered any other data to indicate completion of semesters.
- `Semesters` is updated through method `computeSemesters` in `CourseInfo`.
- `totalSemesters` is changed by user through the command `course edit`.
- `remainingSemesters` is computed by method `computeRemainingSemesters`. This method uses the `moduleList` to check for `module` that has `Semester` and `Grade`. The `module` which fulfils the mentioned and has the latest `Semester` will be taken as the latest completed semester.

Compute Credits

In this section, we describe how `computeCredits(requirement: Requirement[])` works to recompute the latest `Credits`.

As previously mentioned, this method is invoked everytime there is a possible change in the total course `Credits`.

This might be caused through the following commands:

- `module done` where a `Module` is attributed a grade and marked done. Resultantly, all `Requirements` in the `UniqueRequirementList`, consisting of that `Module` would have to be updated. Also, but most importantly, the `creditsFulfilled` attribute of those `Requirements` would have to be updated, causing an eventual change in the total course `creditsFulfilled` of `Credits` (in `CourseInfo`).
- `requirement un/assign` where `Module(s)` are assigned to that particular requirement, where some `Modules` might have already been marked as done and given a grade, hence falling back to the first scenario, where `creditsFulfilled` of a course would have to be updated.
- `requirement edit` where the `creditsRequired` attribute of that requirement might be updated, resulting in the need to update the overall course `creditsRequired` as well
- And many others such as; `requirement add`, `requirement delete`, `module edit`, `module delete`

Now, we have specified the possible scenarios where `computeCredits(…)` might have to be invoked to update `Credits` of `CourseInfo`, however we have not described how it actually works.

As from our previous discussion, we note that `creditsFulfilled` and `creditsRequired` is computed through summing up the `creditsFulfilled` and `creditsRequired` for the individual `Requirements`.

More formally, the interactions between the various classes, for the computation to be performed are as such:

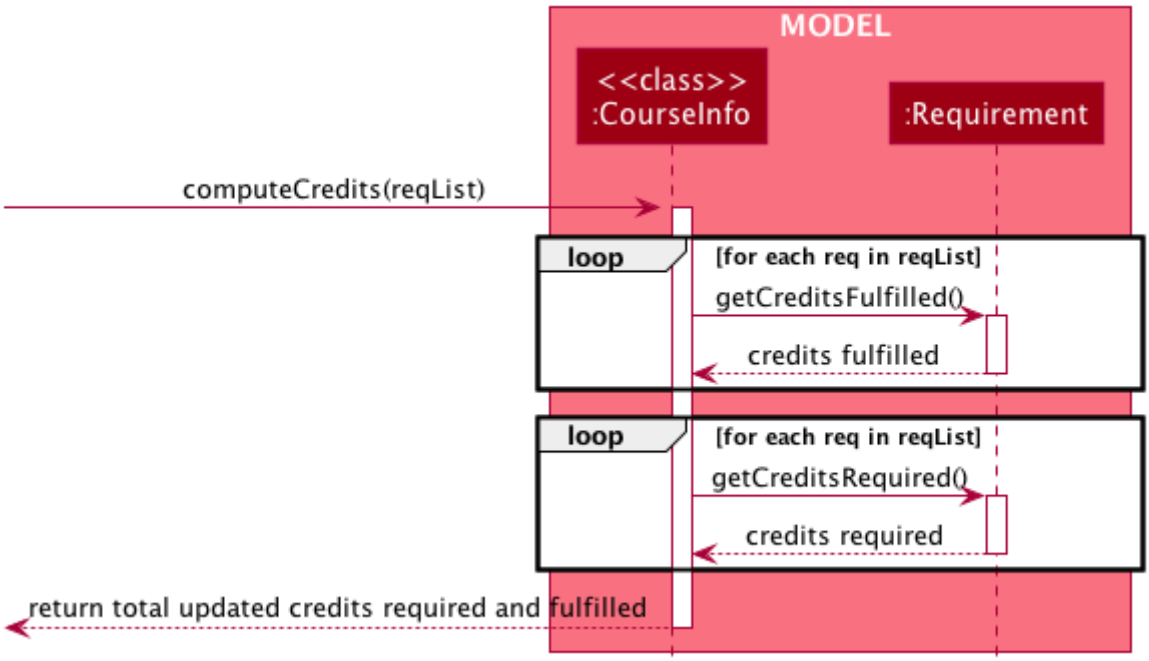


Figure 15. Sequence Diagram for computing updated credits

Compute Cap

Similar to Section 3.1.1.3, "Compute Credits", `Cap` has to be updated frequently, each time module information in `coursebook` changes, and the `computeCap(…)` method facilitates the recomputation of the updated `Cap`.

The diagram below describes the interactions between the various classes, as the computation is performed:

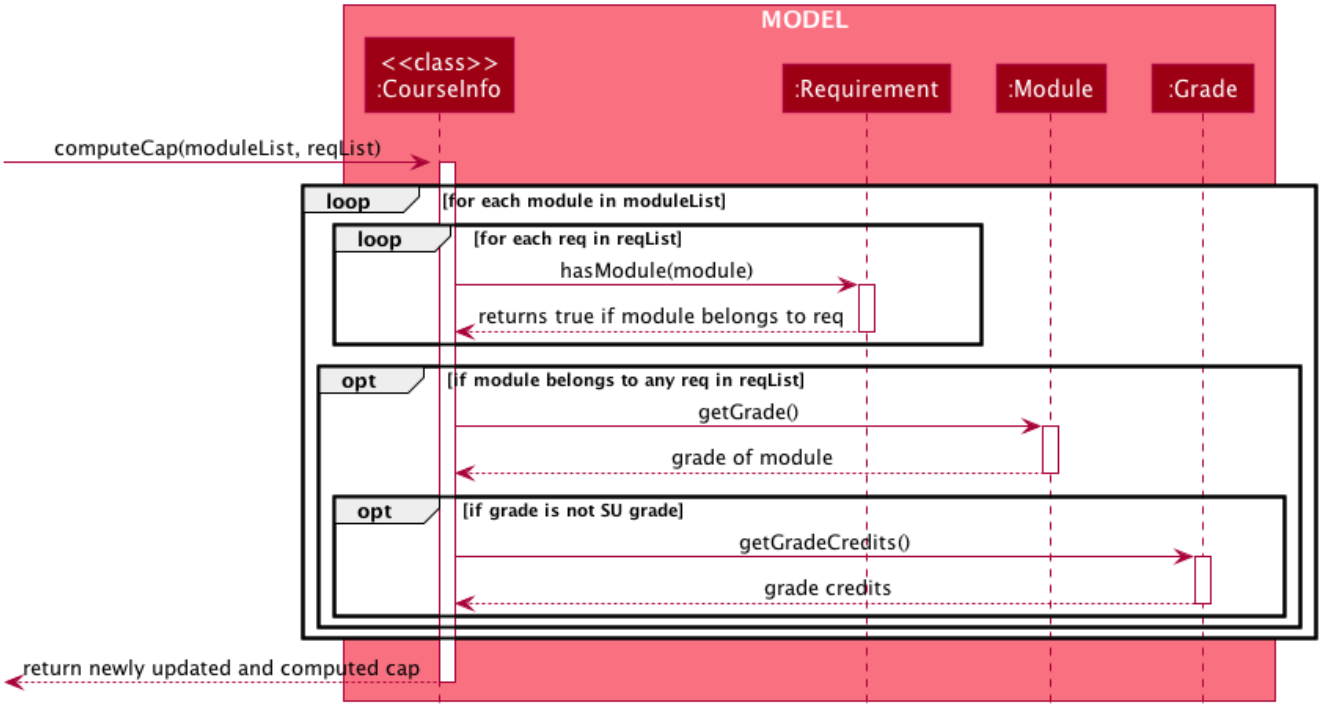


Figure 16. Sequence Diagram for computing updated cap

We note that from above, `CourseInfo` does most of the interfacing with other classes, and the rest of the classes don't interact with each another.

In summary, the following steps are performed as `computeCap(…)` is invoked:

(For each `Module` in the `moduleList`):

1. `CourseInfo` first iterates through `reqList` to determine if a `Module` belongs to any `Requirement`
2. If it does, `CourseInfo` again interfaces with `Module` to extract its `Grade`.
3. Finally, `CourseInfo` interacts with `Grade` to determine if the `Grade` is a non-SU grade.
4. If the `Grade` is non-SU, the `Module` is factored into Cap computation.

3.1.2. Course Edit

Section by: Teri

Overview

Users can edit their course info, which are `Name` and `Semesters` by using the `course edit` command.

Implementation

Here is how the courseInfo class updates when name of course is edited.

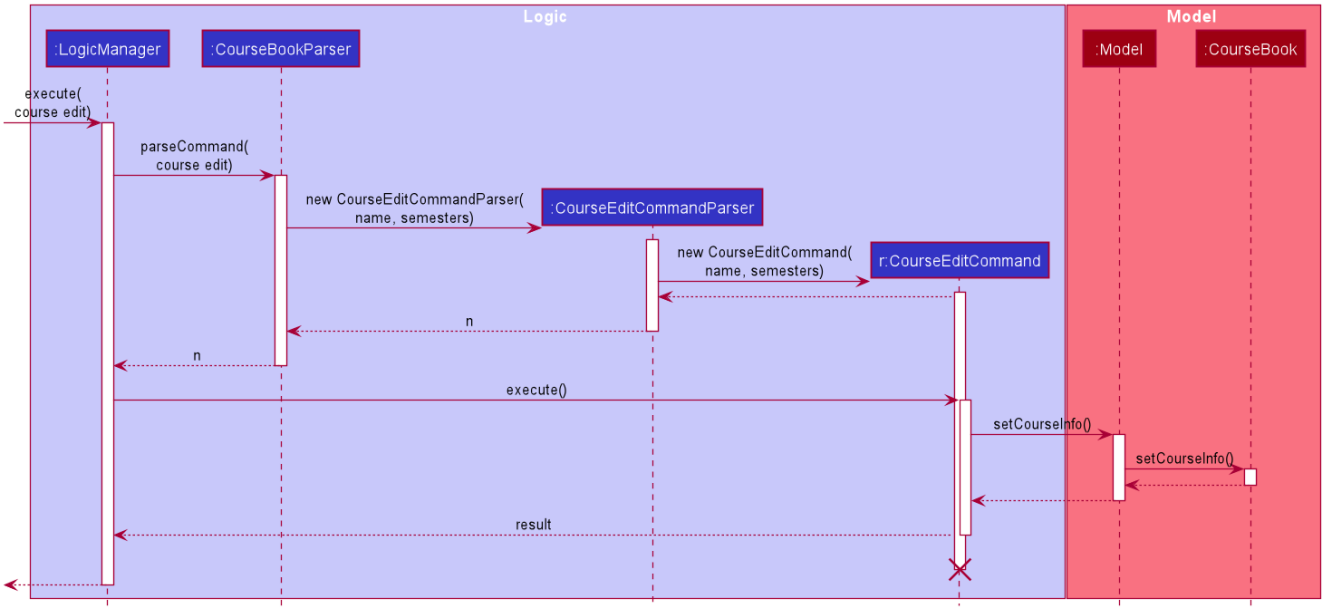


Figure 17. Sequence Diagram when editing course name and course semesters.

When a user edits name of a course, the user has to specify the prefix `n/` for `Name` and/or `s/` prefix for `Semesters`.

Then the application proceeds to do the following steps:

1. The `CourseEditCommandParser` is called to parse the `CourseEditCommand` with the `n/` and `s/` prefix.
2. The `CourseEditCommand` is executed and calls `setCourseInfo` to `Model`.
3. `Model` calls the same method `setCourseInfo` to `CourseBook`.

4. The new course `Name` and course `Semesters` is set in the `CourseBook`.

3.1.3. Course Achieve

Section by: Teri

Overview

Users can get an automatic calculation of their desired C.A.P. by using the `course achieve` command and entering their desired `Cap`.

Implementation

The computation of C.A.P. is done through `computeEstimatedCap` in `courseInfo` which uses `Semesters` and `Cap` of `courseInfo`.

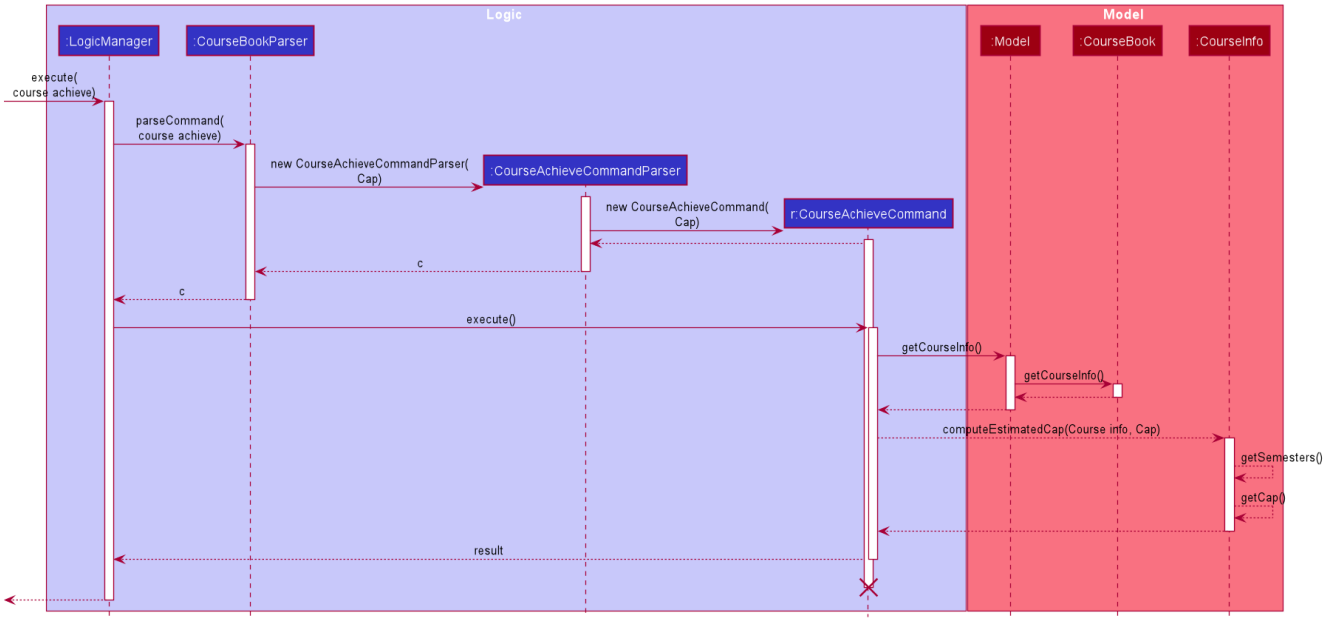


Figure 18. Course Achieve Sequence Diagram

When a user wants to calculate achievable C.A.P., the user has to specify the prefix `c/` for `Cap`. Then the application proceeds to do the following steps:

1. The `CourseAchieveCommandParser` is called to parse the `CourseAchieveCommand` with the `c/` prefix.
2. The `CourseAchieveCommand` is executed and it calls method `getCourseInfo` in `Model` to get `CourseInfo`.
3. With the `CourseInfo` and `Cap`, `CourseAchieveCommand` calls method `calculateEstimatedCap` in `CourseInfo`.
4. `CourseInfo` calls method `getSemesters` and `getCap` to itself to get the following information:
    i. `Semesters`
    ii. `Cap`
5. `computeEstimmatedCap` computes and returns estimate `Cap`.
6. The result is passed back to the user.

Design Considerations

Invalid and Unachievable C.A.P.

It is possible that a calculated `Cap` to achieve is not a valid `Cap`. In such situations, an exception is thrown within the `computeEstimatedCap` command and it is caught in the `CourseAchieveCommand`. User will be given feedback that the desired C.A.P. is not achievable.

The figure below illustrates this:

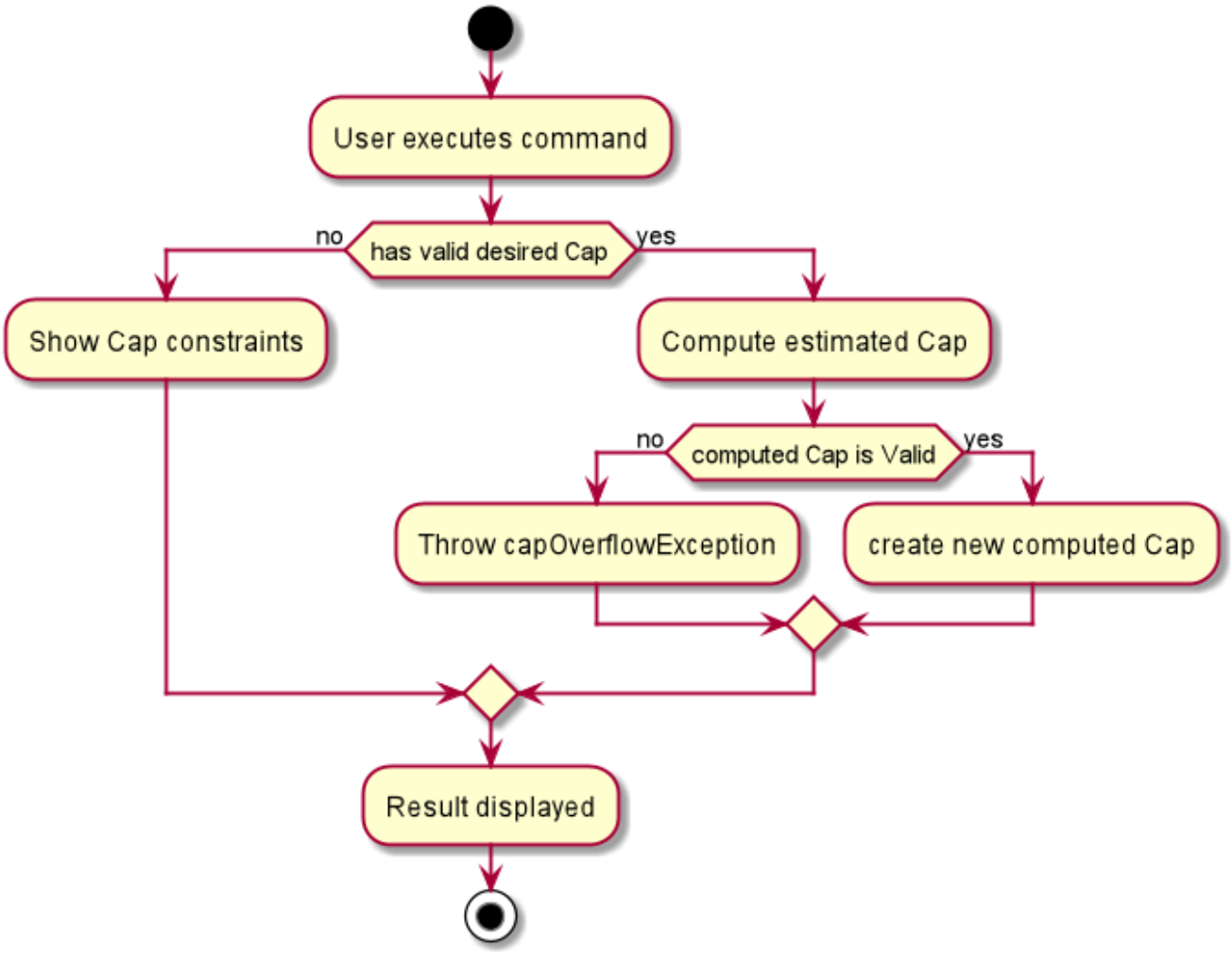


Figure 19. Course Achieve Activity Diagram

Therefore, there are three types of result displayed to User:

1. When User enters invalid Cap to achieve
2. When computed Cap is invalid
3. When computed Cap is valid

3.2. Module Feature

3.2.1. Module Component

The `Module` component is the building block of all other components in the system. In order to track the number of `credits` left to fulfill for each `requirement`, each `module` is stored in a `UniqueModuleList` and the `credits` tied to each `module` is then tabulated.

Besides being necessary in tracking the amount of credits left for a `requirement`, `modules` are also used to decide which `semester` the user is currently in. When a `semester` is tagged to a `module`, either when a new `module` is added or an existing `module` is edited, the latest `semester` of all `modules` in the `filteredList` of modules is taken to be the current semester.

A module must have the following non-optional values:

Value Type	Class Name	Example
String	Title	Software Engineering
String	ModuleCode	CS2103T
String	Credits	4
List	ModulePrerequisites	CS2030, CS2040
List	ModulePreclusions	CS2103, CS2103T, CS2113T, CS2113

A module may also have the following optional values:

Value Type	Class Name	Example
String	Semester	Y3S1
String	Grade	A+

Figure 20, “Module Class Diagram” illustrates the relation between the various classes:

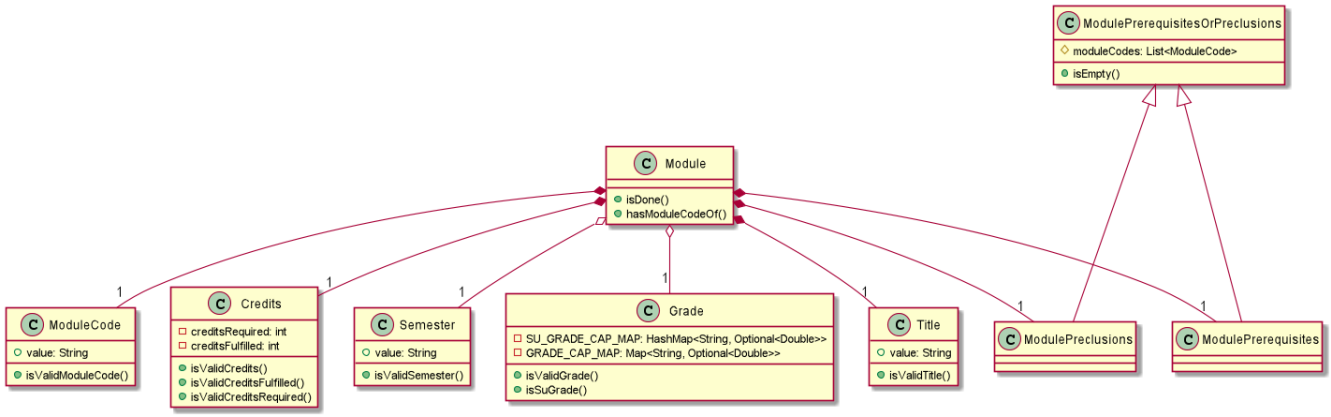


Figure 20. Module Class Diagram

3.2.2. Module Add Auto

Overview

The "automatic" addition of `modules` allows users to add up to 10 `modules` at once with all non-optional values filled in. This is done by making a HTTP GET request to the NUSMods API and fetching the module data given in a JSON format.

Implementation

The automatic filling in of `module` details on addition of a new `module` is facilitated by `NusModsRequester`. It creates a new instance of `GetRequestManager` which it relies on to make a request to the NUSMods API. Upon receiving a response, it creates an instance of JsonParsedModule.

`JsonParsedModule` parses the JSON object given in the response of the initial request and stores the following values:

Table 1. JsonParsedModule Table of Values

Value Type	Name	Example
String	title	Software Engineering
String	moduleCode	CS2103T
String	credits	4
String	prerequisite	CS2040C or (CS2030 and (CS2040 or its equivalent))
String	preclusion	CS2103, CS2103T, (CS2113T for CS2113), (CS2113 for CS2113T)

NOTE Table 1, "JsonParsedModule Table of Values" illustrates the difficulty in parsing `prerequisites` and `preclusions` as the data provided is not in a standard format

The created `JsonParsedModule` object is then converted into a `Module` object, which is subsequently added to the `courseBook` via the method `addModule` of the `ModelManager`.

Figure 21, "Module Add Auto Sequence Diagram" illustrates this:

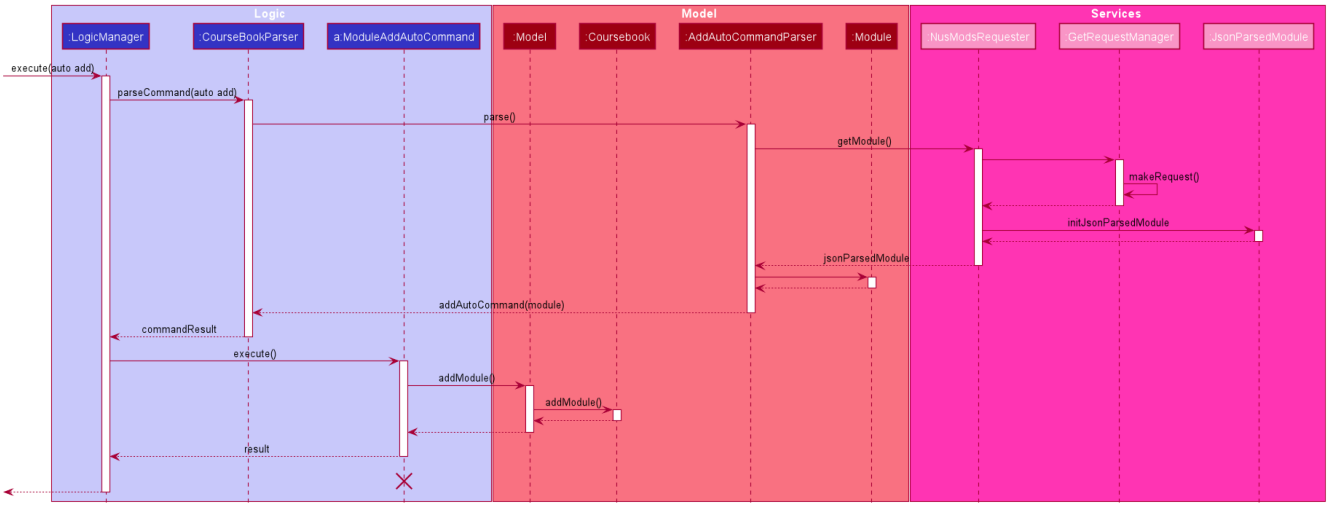


Figure 21. Module Add Auto Sequence Diagram

Design Considerations

Most of the design considerations arose as a result of having to make a network request.

A *secondary* module addition feature

As with all network requests, this feature might not work as intended in certain circumstances. Possible cases are:

1. High Network Congestion
2. Poor Network Connection
3. NUSMods Offline

In such situations, it becomes difficult or impossible to carry out the addition of `modules` using this command. Therefore, this feature was built on top of the primary `module add` feature, ensuring that the user could still manage to add `modules` even when faced with the issues as listed above.

The use case for this situation is as follows:

System: iGrad

Use case: UCM1 - Add module via NUSMods

Actor: User, NUSMods

MSS:

1. User wants to add a module.
2. iGrad requires user to specify the module codes.
3. User enters the module codes corresponding to the modules he wishes to add.
4. iGrad sends a request to NUSMods.
5. NUSMods responds with the requested data.

6. iGrad adds the module to the module list.
7. User views the module in the module list.

Use case ends.

Extensions:

5a. NUSMods does not respond with the requested data.

5a1. User adds module manually

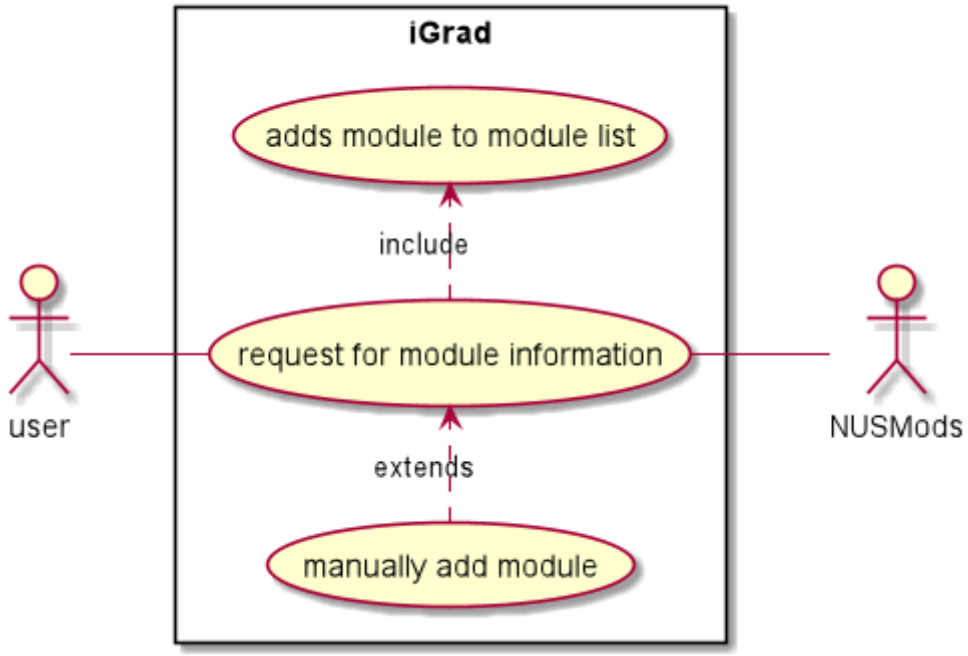


Figure 22. Module Auto Add Use Case Diagram

Messages for individual modules

As this feature allows a user to add `modules` by batches, it is possible that one or more `modules` in the batch are invalid or require warning messages. In order to facilitate this, the processing of the list of `modules` and the generation of error and warning messages were done in parallel. This was because if the list of `modules` was processed first, `modules` with issues would be filtered out without notice, leading to a confusing user experience.

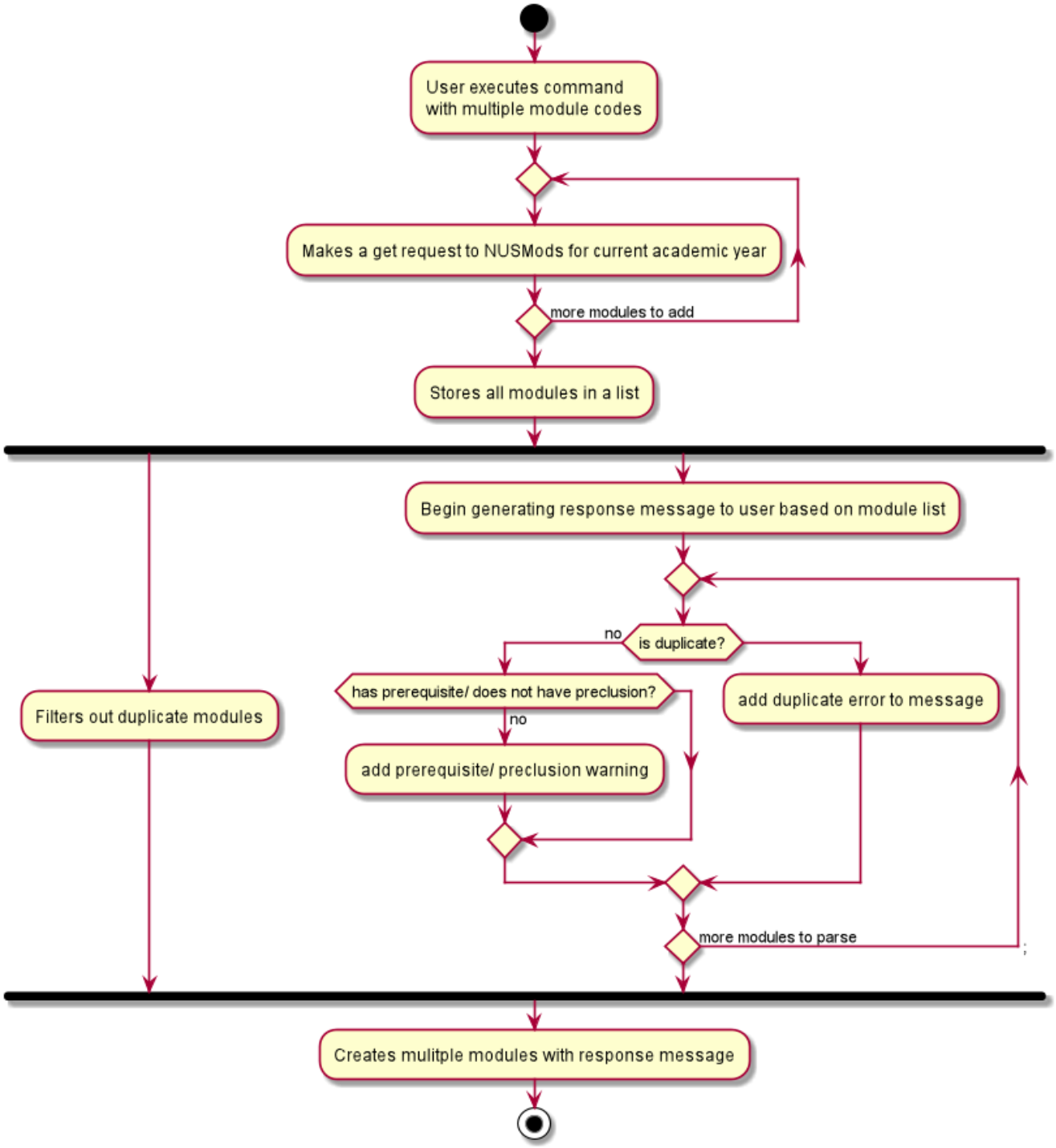


Figure 23. Module Add Auto Batch Processing

Improving the user experience [PROPOSED]

Due to network latency - the Round Trip Time taken from when a request is made to when a response is received - the user might experience a situation where it appears that the application has stopped working.

For a large batch of `modules`, the application might also display a *not responding* label in the toolbar. In order to improve the user experience, it is ideal that a loader be displayed when waiting for the response from the server. However, due to time constraints, this was not implemented.

Getting the latest data [DEPRECATED]

Past iterations of this feature made a maximum of two requests for one `module`. The first request would attempt to get the `module` for the current academic year, whilst the second request attempted to get the `module` for the previous academic year, in the event the module for the current academic year was not available.

This process is illustrated in Figure 24, "Previous implementation of Module Add Auto". However, it was decided that the benefits of making a maximum of one request outweighed that of getting the latest module information and thus, currently only one request is made.

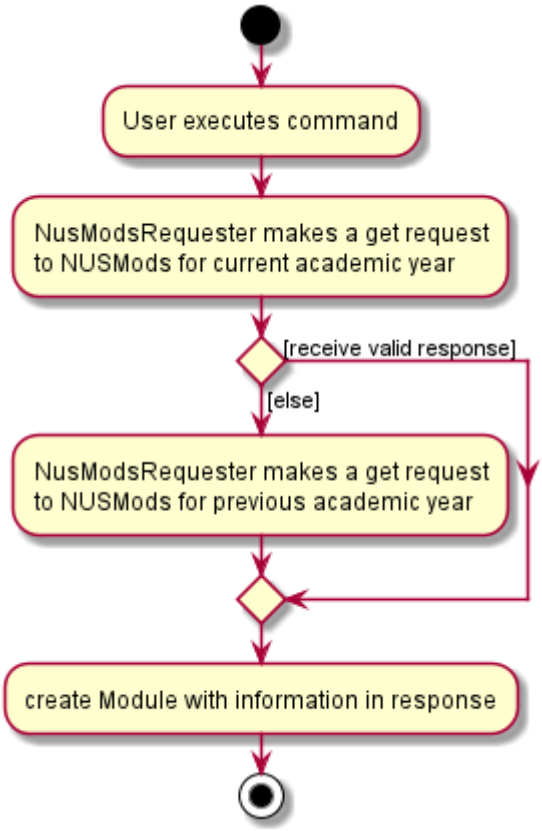


Figure 24. Previous implementation of Module Add Auto

3.2.3. Module Filter

Overview

The average number of `modules` a student has for a 4 year program in NUS is 40. The application window, however, can display a maximum of 9 `modules` for a 17" screen and considerably less for smaller displays. As a result, it is imperative that users have a way to filter `modules` so that only what is required is displayed.

Implementation

The filtering of modules is done by calling the `execute` function of `ModuleFilterCommand` .`ModuleFilterCommand` takes in optional parameters `Semester`, `Credits`, `Grade` and an operator, which could be `AND` or `OR`.

The matching is done by the functions `checkSemesterMatch(Module m)`, `checkCreditsMatch(Module m)` and `checkGradeMatch(Module m)`. Unlike the other two functions, `checkCreditsMatch(Module m)` does not check if the actual `credits` for a module is present since the `credits` field in a module is

compulsory.

The `AND` operator specifies that the provided parameters be chained with the logical *and* operator.

The `OR` operator specifies that provided parameters be chained with the logical *or* operator.

When the filter command is issued, the `Model` updates the module list based on the predicate given. Figure 25, “Module Filter Sequence Diagram” illustrates this sequence of events:

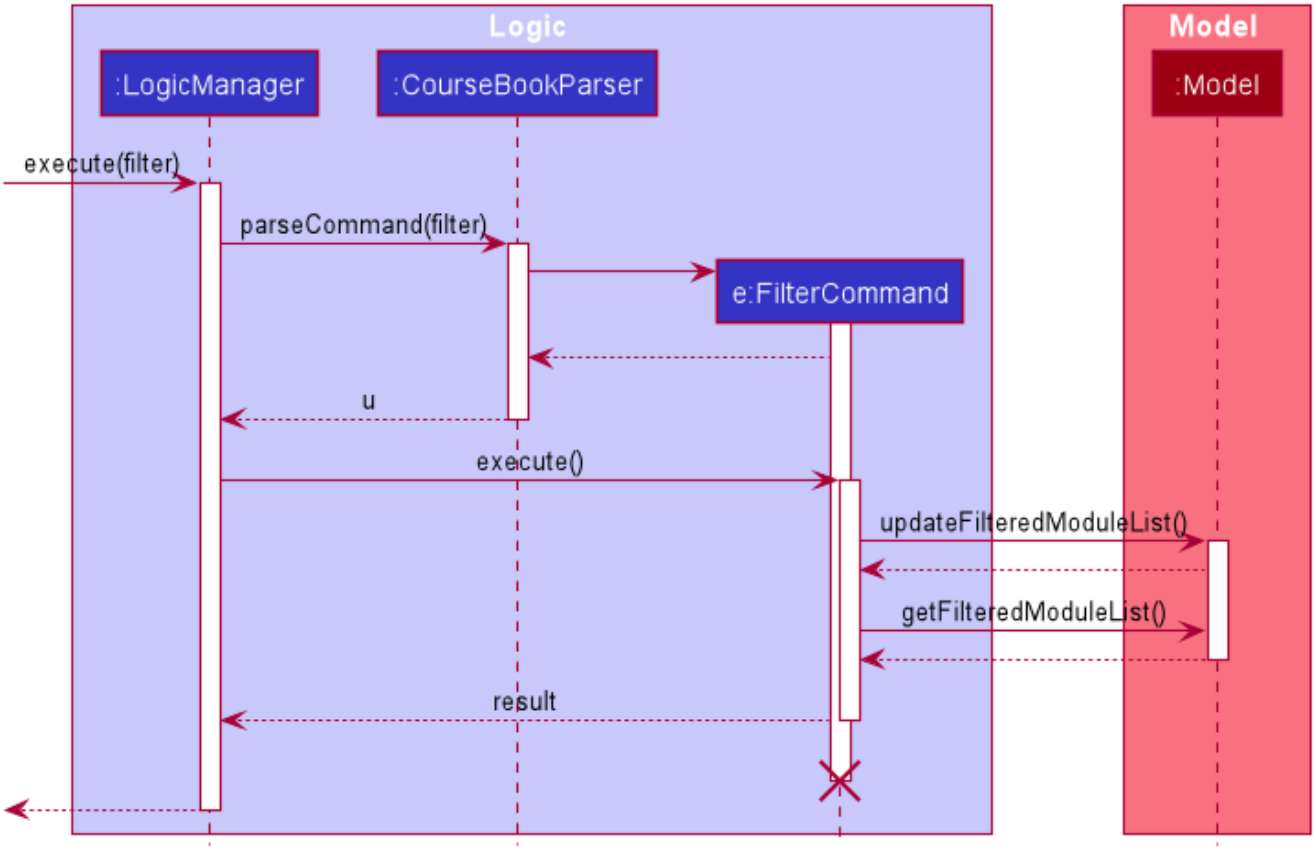


Figure 25. Module Filter Sequence Diagram

Design Considerations

Resetting the state

When this command is issued, the modules that do not match the predicate given will disappear from the module list. It is thus necessary to allow the user to issue a new command in order to view all the modules again.

Whilst creating a new command such as `module reset` was proposed, it was decided that a new command would only serve to make the user experience more complicated than it should be.

Therefore, an allowance was made for `module filter` to reset the state when receiving no parameters, a divergence from the way other functions handled the situation of empty parameters.

Figure 26, “Module Filter Activity Diagram” illustrates this clearly:

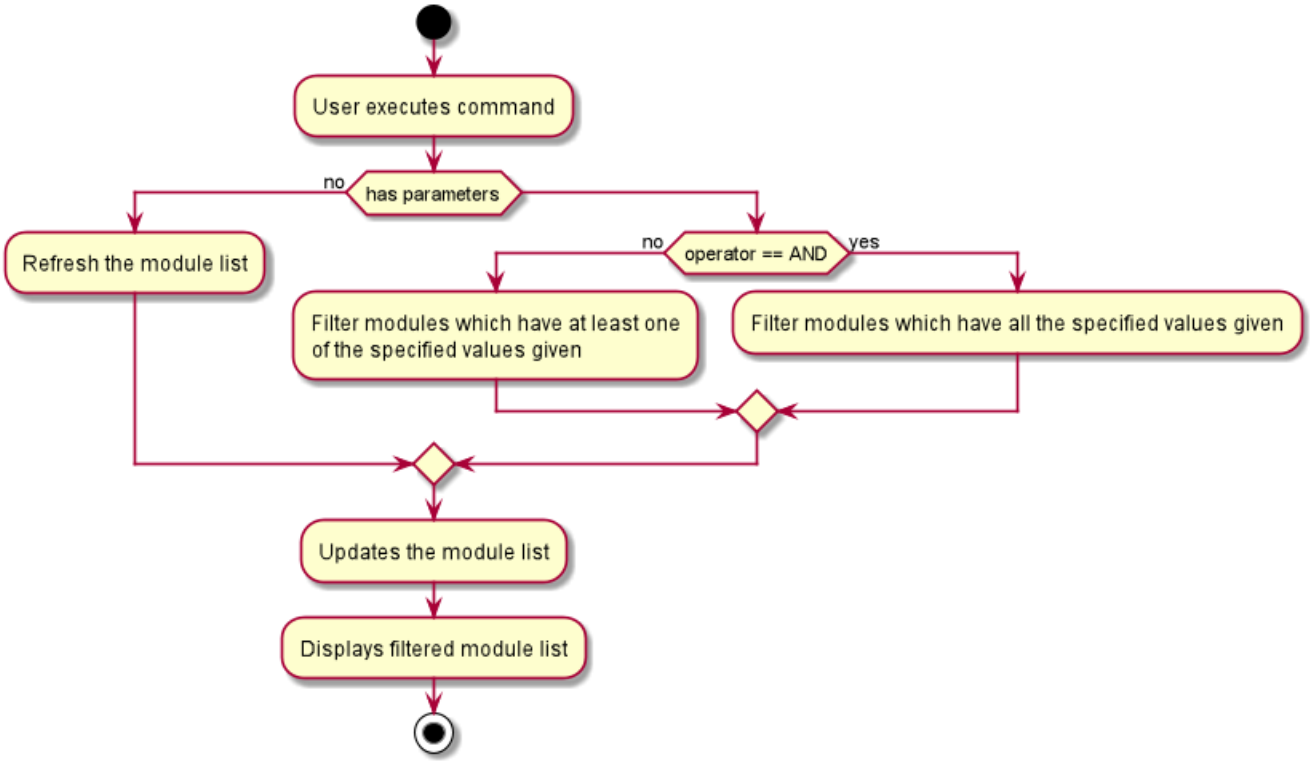


Figure 26. Module Filter Activity Diagram

The `filteredList` of `modules` therefore takes three general states (see Figure 27, "Module Filter State Diagram"):

1. Initial State

   The `module` list is unfiltered. All `modules` are displayed.

2. Filtered by `AND` State

   The `module` list is filtered by a predicate composed of the provided parameters chained together with the logical *and* operator

3. Filtered by `OR` State

   The `module` list is filtered by a predicate composed of the provided parameters chained together with the logical *or* operator

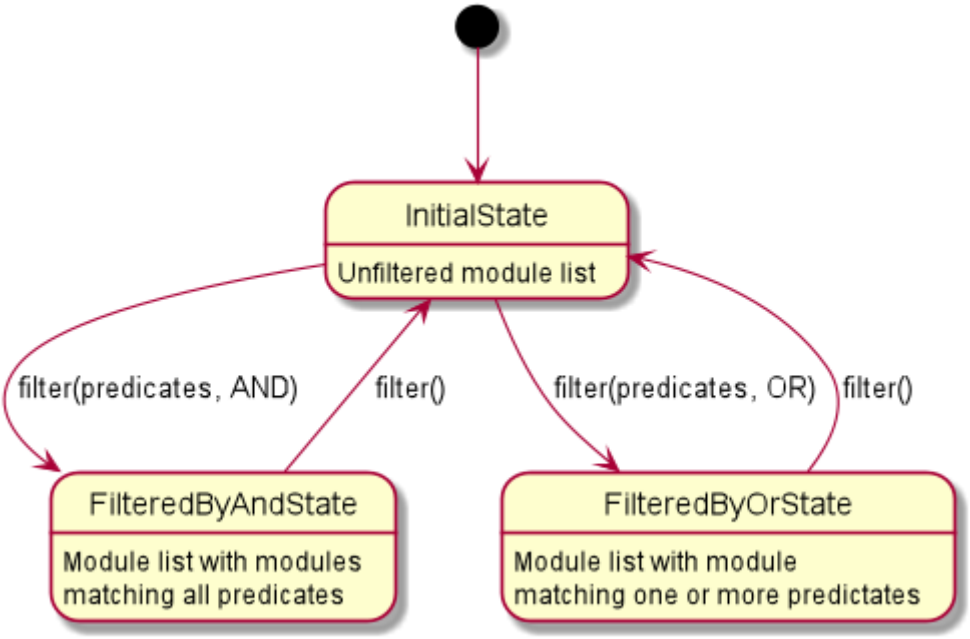


Figure 27. Module Filter State Diagram

Displaying Filter State [PROPOSED]

An issue with the filtering of modules is that when the current state is not obvious, the user might lose track of what the module list is filtering on. To solve this problem, it would be an improvement to display the current state prominently to the user. '"

3.3. Requirements Feature

Within a course, there are multiple requirements to be tracked.

3.3.1. Implementation

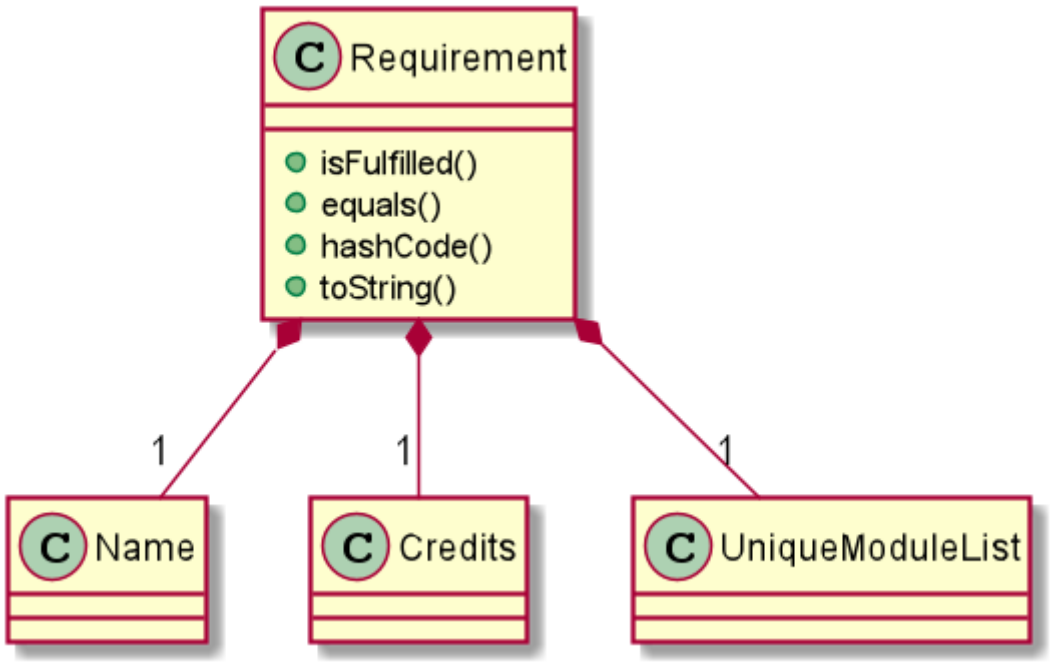


Figure 28. Structure of the Requirement class.

A requirement consists of three components: title, credits and unique module list. The unique module list implies that each requirement stores modules assigned to that requirement. Multiple requirements can exist in the course book at any one time.

The requirement-related commands that can be called are:

- `requirement add` - adds a new requirement to the course book
- `requirement edit` - edits an existing requirement in the course book
- `requirement delete` - deletes an existing requirement from the course book
- `assign` - assigns a module to the requirement

Here is how the requirement class updates when a requirement is added:

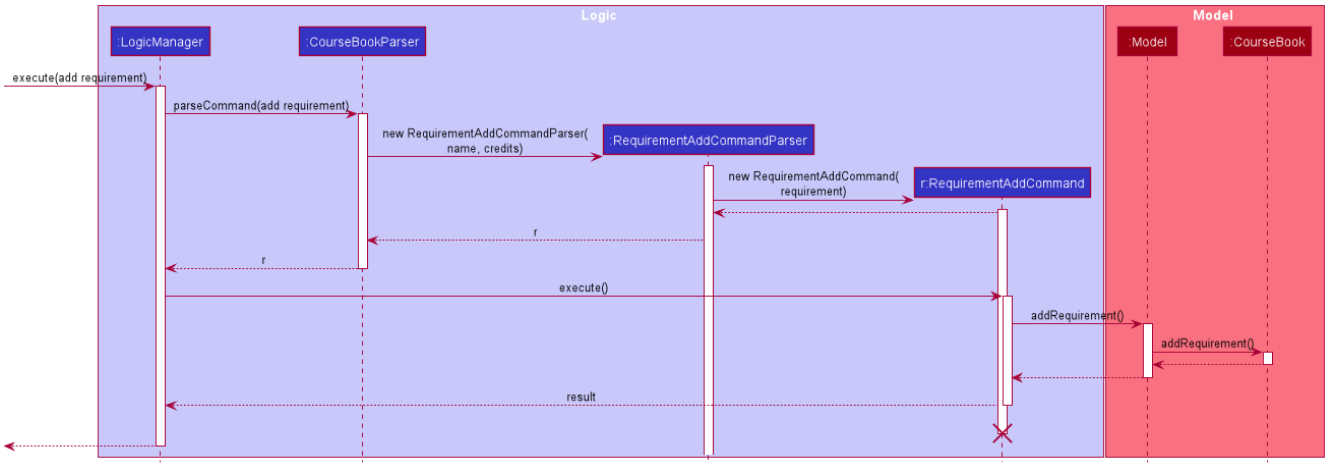


Figure 29. Sequence Diagram when adding a requirement.

When the user adds a requirement, the user has to specify two prefixes: `n/` for title and `u/` for credits value (number of credits needed to fulfill for the requirement). Then, the application proceeds to do the following steps

Step 1: The RequirementAddCommandParser is called to parse the RequirementAddCommand with the `n/` and the `u/` prefixes into a new requirement.

Step 2: The RequirementAddCommand is executed to add the new requirement to the model. In this step, the following check is performed:

- Check if a requirement with the same title already exists in the course book.

Step 3: The new requirement is added to the course book.

3.4. Export Feature

3.4.1. Overview

The export feature allows the user to export their data into a .csv file.

3.4.2. Implementation

The export feature is facilitated by the `CsvWriter`. The `ExportCommand` calls `exportModuleList()` on `ModelManager`, which then performs a filter on the `filteredList` of modules in order to filter out all modules where the `Optional<Semester>` object `isEmpty()`.

The `ModelManager` allows for `write()` to be called on `CsvWriter` only if the `filteredList` has at least one module.

Figure 30, "Export Sequence Diagram" shows the process of exporting modules with semesters:

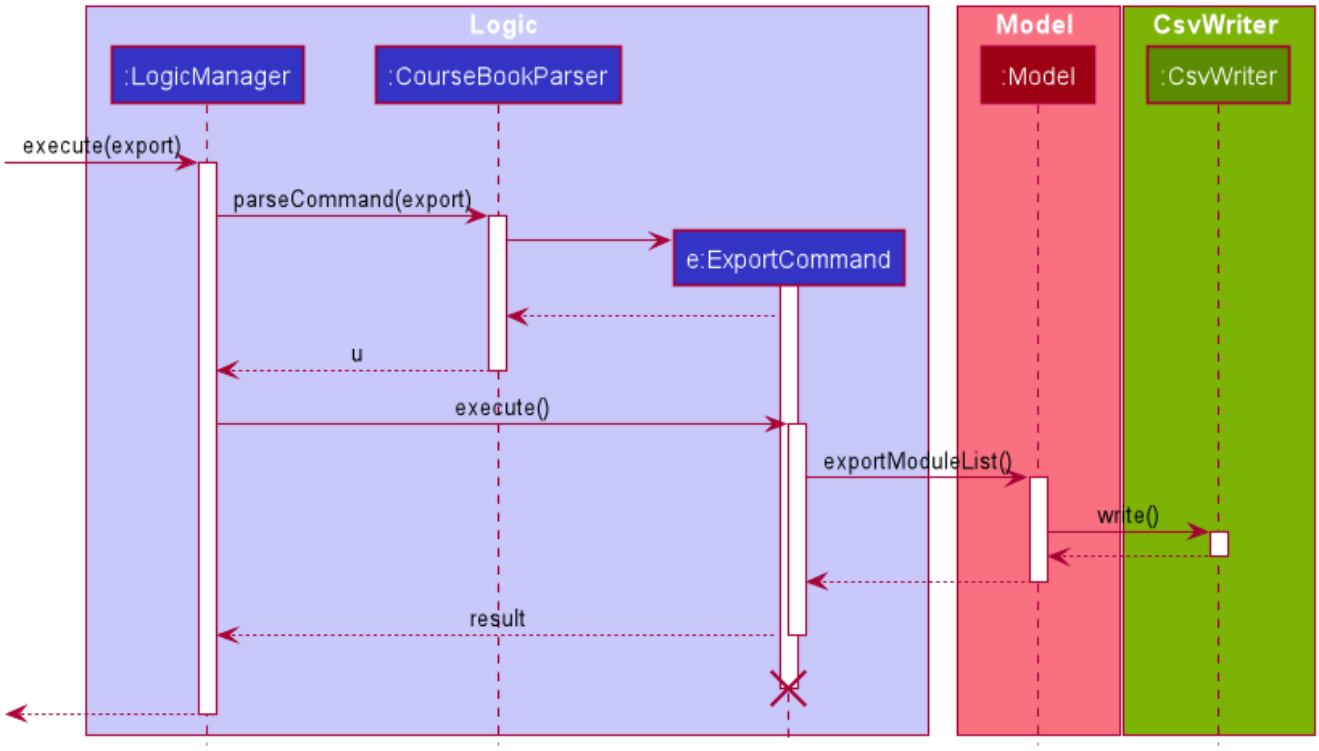


Figure 30. Export Sequence Diagram

Once the `CsvWriter` returns from `write()`, a file titled *study_plan.csv* will be created in the top-level directory.

The top-level directory has two states concerning the generated file (see Figure 31, “Export State Diagram”):

1. Empty State
    - does not contain the file study_plan.csv
2. Non-empty State
    - contains the file study_plan.csv

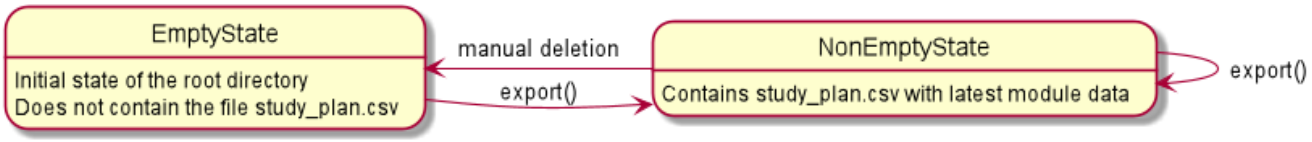


Figure 31. Export State Diagram

As seen in the figure, it is only possible to revert the state i.e. from Non-empty State to Empty State, by externally deleting the file.

Additionally, if `export` is issued when there is an existing *study_plan.csv*, the current file will be overwritten.

3.4.3. Design Considerations

Writing and Reading Issues

It is possible that the user has the *study_plan.csv* file open while attempting to export data. The only known solution is for the user to close the file and issuing the command again.

3.5. Undo Feature

3.5.1. Overview

The undo feature allows for the prior state of a `courseBook` to be saved and loaded from when needed.

3.5.2. Implementation

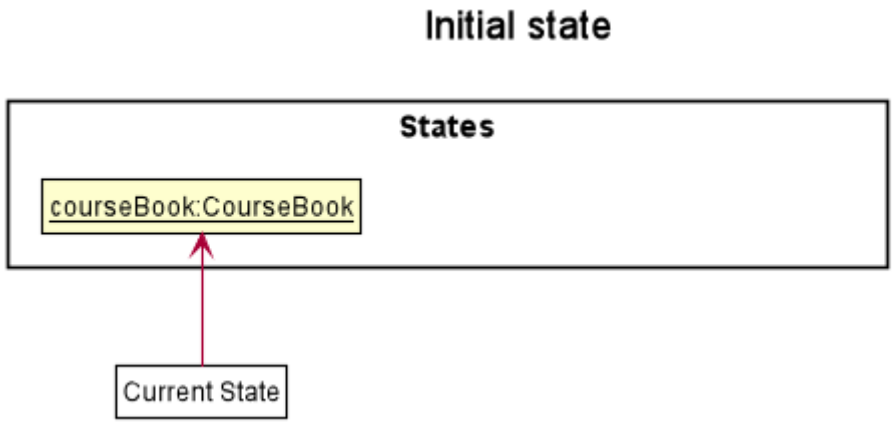
The undo mechanism is facilitated by the `ModelManager` and the `LogicManager`.

The function `saveCourseBook()`, which composes the `storage` object, is used to save the previous state of the `coursebook` in the file *backup_coursebook.json* when a new command is executed.

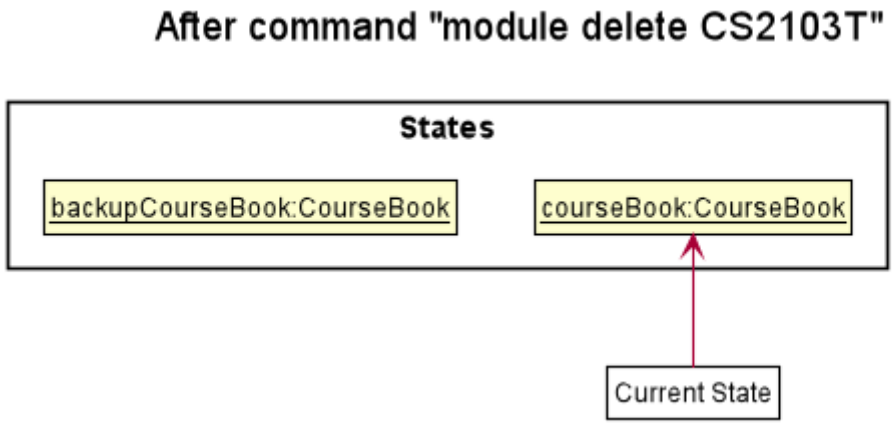
The function `undoCourseBook()`, implemented by the `ModelManager`, reads from the file *backup_coursebook.json* and restores the data by calling `setCourseBook()`

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

Step 1. The user starts up the application.

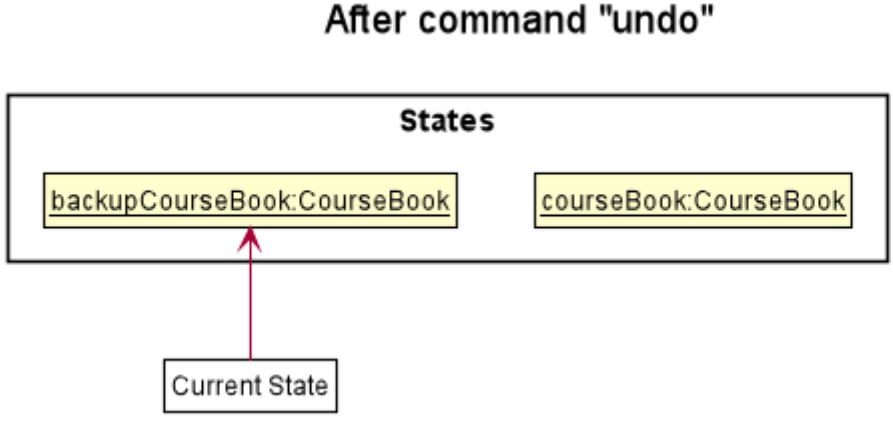


Step 2. The user issues the command `module delete CS2103T`. The previous state of the course book is stored into the file at `backup_coursebook.json` by `saveCourseBook()`. This state still contains the module CS2103T.

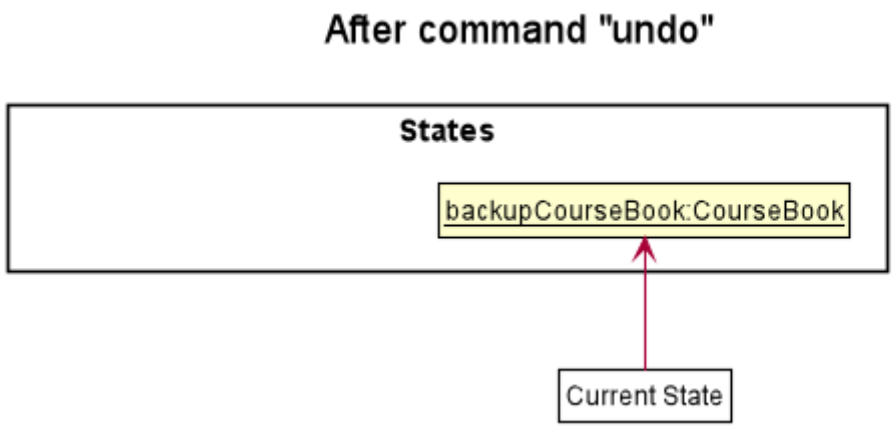


NOTE When the `undo` command is executed, the state of the `courseBook` is not saved.

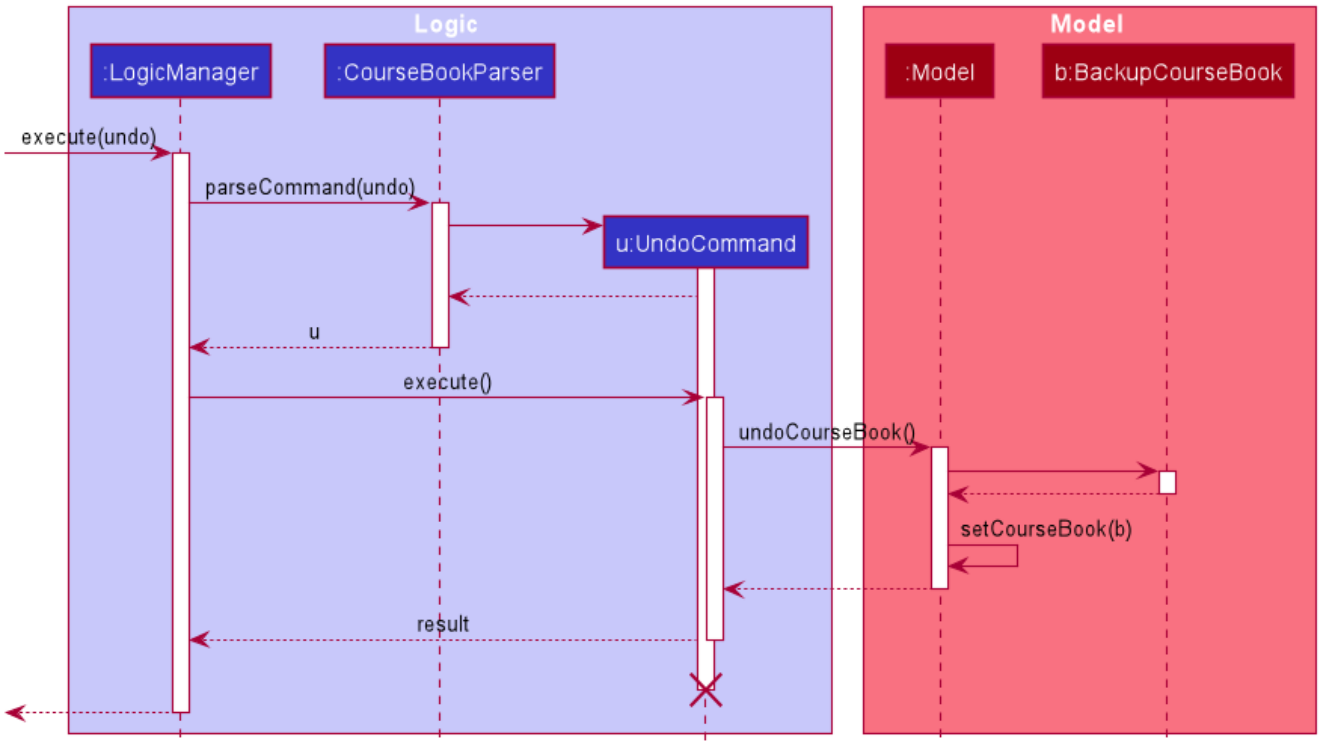
Step 3. The user issues the command `undo`. The file at `backup_coursebook.json` is read from and loaded as the main `courseBook` to be read from.



Step 4. The `backupCourseBook` becomes the main `courseBook` and another instance of `courseBook` will be created as its backup if another command is issued.



The following sequence diagram shows how the undo operation works:



3.5.3. Design Considerations

How undo executes

- Option 1 (current choice): Saves the entire course book.
  - Pros: Easy to implement.
  - Cons: May have performance issues in terms of memory usage.
- Option 2: Individual command knows how to undo/]] by itself.
  - Pros: Will use less memory (e.g. for `delete`, just save the module being deleted).
  - Cons: We must ensure that the implementation of each individual command are correct.

NOTE	While it is recognised that Option 1 may have performance issues in terms of memory, the actual implementation of the feature is unlikely to cause any memory issues. This is due to the fact that modules, requirements and other data users would require are limited to a degree which will not require large memory allocation.

3.6. [Proposed] Data Encryption

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

3.7. 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 3.8, "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

3.8. 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

Refer to the guide here.

5. Testing

Refer to the guide here.

6. Dev Ops

Refer to the guide here.

Appendix A: Product Scope

Target user profile:

- is a NUS undergraduate
- prefers desktop apps over other types
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition: convenient course requirements tracker for NUS undergraduates

Appendix B: User Stories

Priorities:

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

Priority	As a …	I want to …	So that I can …
* * *	first-time user	create a course	
* * *	student	create a graduation requirement	
* * *	student	input modules under a graduation requirement	keep track of when a graduation requirement is fulfilled
* * *	careless user	change the graduation requirements which I assigned to a course	amend any mistakes made when entering data

Priority	As a …	I want to …	So that I can …
* * *	fickle user	change the modules which I assigned to a graduation requirement	change my study plan
* * *	fickle user	have the option to defer adding modules to a graduation requirement	delay making up my mind on which modules I wish to take
* * *	basic user	see information regarding the course I created, including graduation requirements, modules and gaps (e.g. modules that are unassigned) that need to be filled	
* * *	user	see the latest updated information about any module	make informed decisions
* * *	basic user	mark when a module is completed	
* * *	basic user	input the grades of a module	
* *	basic user	retrieve my CAP of any semester at a command	stay updated about my results

Priority	As a …	I want to …	So that I can …
* *	user	input my desired CAP and have the program calculate what grades I need to achieve	find out how well I need to do in following semesters
* *	user	group modules by graduation requirement	view by requirement
* *	user	group modules by semester	view by semester
*	user who wants to take notes	record notes for each module	record why I took it
*	picky user	customize display settings	customize to my needs
*	advanced command line user	use familiar linux commands	navigate more easily

{All user stories can be viewed from our wiki page and from our issues tracker.}

Appendix C: Use Cases

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

Use case: U01 - Create Course

MSS:

1. iGrad starts up.
2. User requests to create a course.
3. iGrad creates the course.

   Use case ends.

Extensions:

2a. The course name is not provided.

- 2a1. iGrad prompts user for course name.
- 2a2. User enters a course name.

  Steps 2a1-2a2 are repeated until the a non-empty course name is provided.

  Use case resumes at step 3.

Use case: U02 - Create Requirement

MSS:

1. User requests to create a course.
2. iGrad creates course (UC01).
3. User requests to create a requirement.
4. iGrad creates the requirement.

   Use case ends.

Extensions:

3a. The requirement title is not provided.

- 3a1. iGrad prompts user for requirement title.
- 3a2. User enters a requirement title.

  Steps 3a1-3a2 are repeated until the a non-empty requirement title is provided.

  Use case resumes at step 4.

Use case: U03 - Create Module

MSS:

1. User requests to create a module by providing a module code.
2. iGrad creates the module with its data pulled from NUSMods.

   Use case ends.

Extensions:

1a. Module data fails to get pulled due to network error.

- 1a1. iGrad takes from its local module data copy.

  Use case ends.

1b. Module data does not exist on NUSMods.

- 1b1. iGrad creates a empty module with only the module code.

  Use case ends.

Use case: U04 - Assign Module to Requirement

MSS:

1. User requests to assign a module to a requirement by specifying its module code.
2. iGrad assigns module to requirement.

   Use case ends.

Extensions:

1a. Module does not exist in system.

- 1a1. iGrad creates the module (UC03).

  Use case resumes at step 2.

1b. Module has already been assigned to the requirement.

- 1b1. iGrad generates a warning and stops the assignment.

  Use case ends.

{More to be added}

Appendix D: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java 11 or above installed.
2. Should be able to hold up to 100 modules without a noticeable sluggishness in performance (i.e. should take less than 1 second to load)
3. A user with above 70 wpm 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.
4. The interface should be intuitive enough such that a user who has never seen the user guide is able to use the basic features.

{More to be added}

Appendix E: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

Private contact detail

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

Appendix F: Product Survey

Product Name

Author: …

Pros:

- …
- …

Cons:

- …
- …

Appendix G: Instructions for Manual Testing

Given below are instructions to test the app manually.

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

G.1. Launch and Shutdown

1. Initial launch
   a. Download the jar file and copy into an empty folder
   b. Double-click the jar file
      Expected: Shows the GUI with 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 … }*

G.2. Deleting a module

1. Deleting a module while all modules are listed

    a. Prerequisites: List all modules using the `list` command. Multiple modules in the list.

    b. Test case: `delete 1`
    Expected: First module is deleted from the list. Details of the deleted module shown in the status message. Timestamp in the status bar is updated.

    c. Test case: `delete 0`
    Expected: No module 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 … }

G.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 … }*