ModPlan v1.4 - Developer's Guide

1. Introduction . . . 2
2. Setting up . . . 2
3. Overall Design . . . 3
  3.1. Architecture . . . 3
  3.2. JsonWrapper . . . 4
  3.3. Storage . . . 4
  3.4. Parser . . . 5
  3.5. PlannerUi . . . 5
  3.6. Command . . . 6
  3.7. CCA . . . 6
4. Implementation . . . 7
  4.1. Parser . . . 7
  4.2. Grading . . . 10
  4.3. Cap Calculation . . . 13
  4.4. Sorting . . . 18
  4.5. Showing Module Reports . . . 21
  4.6. Update Reminders . . . 24
  4.7. UpdateModuleCommand . . . 27
  4.8. JsonWrapper . . . 28
  4.9. Logger . . . 33
  4.10. Cca . . . 34
5. Coming in v2.0 . . . 35
  5.1. Password management . . . 35
  5.2. GUI . . . 35
6. Documentation . . . 35
7. Testing . . . 36
  7.1. Overview . . . 36
  7.2. Creating Tests . . . 36
8. Dev Ops . . . 36
  8.1. Build Automation . . . 36
  8.2. Continuous Integration . . . 36
  8.3. Code Coverage Reporting . . . 36
  8.4. Making a Release . . . 37
9. Manual User Testing . . . 37
Appendix A: Product Scope . . . 38
Appendix B: User Stories . . . 38
Appendix C: Use Cases . . . 40
  C.1. Use Case C01: Adding modules to user's timetable . . . 41

C.2. Use Case C02: Viewing graduation requirements . . . 41
C.3. Use Case C03: Viewing class and CCA timetable . . . 42
C.4. Use Case C04: Viewing of modules taken in past semester . . . 42
C.5. Use Case C05: Generation of current CAP based on past modules . . . 42
C.6. Use Case C06: Check Module information via input search . . . 42
C.7. Use Case C07: Creating a personalised Module (eg. for CCAs) . . . 43
C.8. Use Case C08: Checking number of MCs taken . . . 43
Appendix D: Non-Functional Requirements . . . 44
Appendix E: Manual User Testing . . . 44
E.1. Loading Manual Test Data . . . 44
E.2. Manual Testing . . . 45
Appendix F: Glossary . . . 45

By: `Team AY1920S1-CS2113T-F10-1` Since: `Aug 2019` License: `MIT`

1. Introduction

ModPlan

ModPlan is a module planning system that expands on NUSMODS to help NUS students to plan their modules for all four years of their bachelor's degree.

ModPlan is currently only designed to work for Computer Engineering students, and the requisite modules for graduation.

ModPlan uses NUSMODS API to extract official module data from the NUS Registrar's Office.

This Developer Guide is meant for anyone who wishes to contribute to ModPlan further, and serves to guide them through the architecture, implementation methods and high-level design considerations taken during our current development.

2. Setting up

Prerequisites

- JDK version: `11` or above
- Recommended IDE: `IntelliJ IDEA`
- Fork this repo to your GitHub account and clone the fork to your computer

Importing the project into IntelliJ

1. Open IntelliJ (if you are not in the welcome screen, click `File` > `Close Project` to close the existing project dialog first).
2. Set up the correct JDK version.
   - Click `Configure` > `Structure for new Projects` (in older versions of Intellij:`Configure` >

`Project Defaults` > `Project Structure`).
   - If JDK 11 is listed in the drop down, select it. If it is not, click `New…` and select the directory where you installed JDK 11.
   - Click `OK`.
3. Click `Import Project`.
4. Locate the project directory and click `OK`.
5. Select `Create project from existing sources` and click `Next`.
6. Rename the project if you want. Click `Next`.
7. Ensure that your src folder is checked. Keep clicking `Next`.
8. Click `Finish`.

Running the Project

- If you wish to run the `Jar` file, simply run the command `java -jar [CS2113T-F10-1][ModPlan].jar` in any terminal application of your choice.

This is assumes that JDK version `11` and above is installed.

- If you are not using `gradle`, simply locate the `CliLauncher.java` file under `PROJECT_DIRECTORY/src/main/java/planner/main/CliLauncher.java`, and run it in your IDE.
- If you are using `gradle`, import the project as a gradle project using the `Import Gradle Project` function in IntelliJ, and run `gradlew run` from the command line.

3. Overall Design

3.1. Architecture

Our integration approach would be bottom-up, as some of the components we have utilised from a previous project can be adapted and reused of other purposes.

In that way, our implementation approach would also be incremental as oppose to big-bang, since we would need to gradually alter and test out which portions of Duke is reusable in ModPlan.

The **architecture diagram** of ModPlan can be seen in the below diagram, where it shows the high-level architecture design of our project:

[architecture] | *architecture.png*

As seen here, the user interacts with the **Parser** through the command line interface (CLI), which sorts through the user input and calls various commands under the **Command** section.

The classes within **Command** will then utilise data pulled from the **Storage** and **JsonWrapper**, who pull data both locally as well as from NUSMODS API through the **RequestData**.

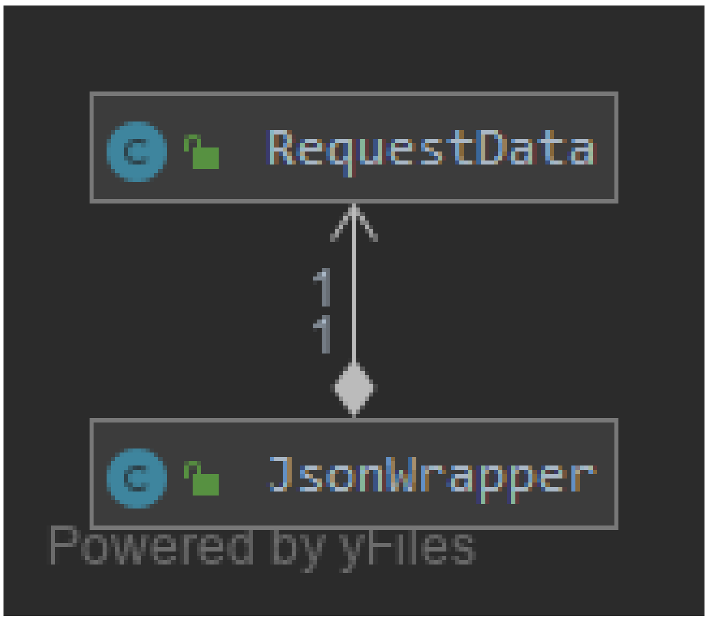
Following which, the **Command** classes perform their execution of the feature, and update the

Storage to store data that has changed. It also shows the user the completed function by printing any necessary information/feedback to the user through the **PlannerUi**.

To read more about the various packages shown in the architecture diagram, please continue reading the below sections Section 3.2, “JsonWrapper” Section 3.3, “Storage” Section 3.4, “Parser” Section 3.5, “PlannerUi” Section 3.6, “Command” Section 3.7, “CCA”.

3.2. JsonWrapper

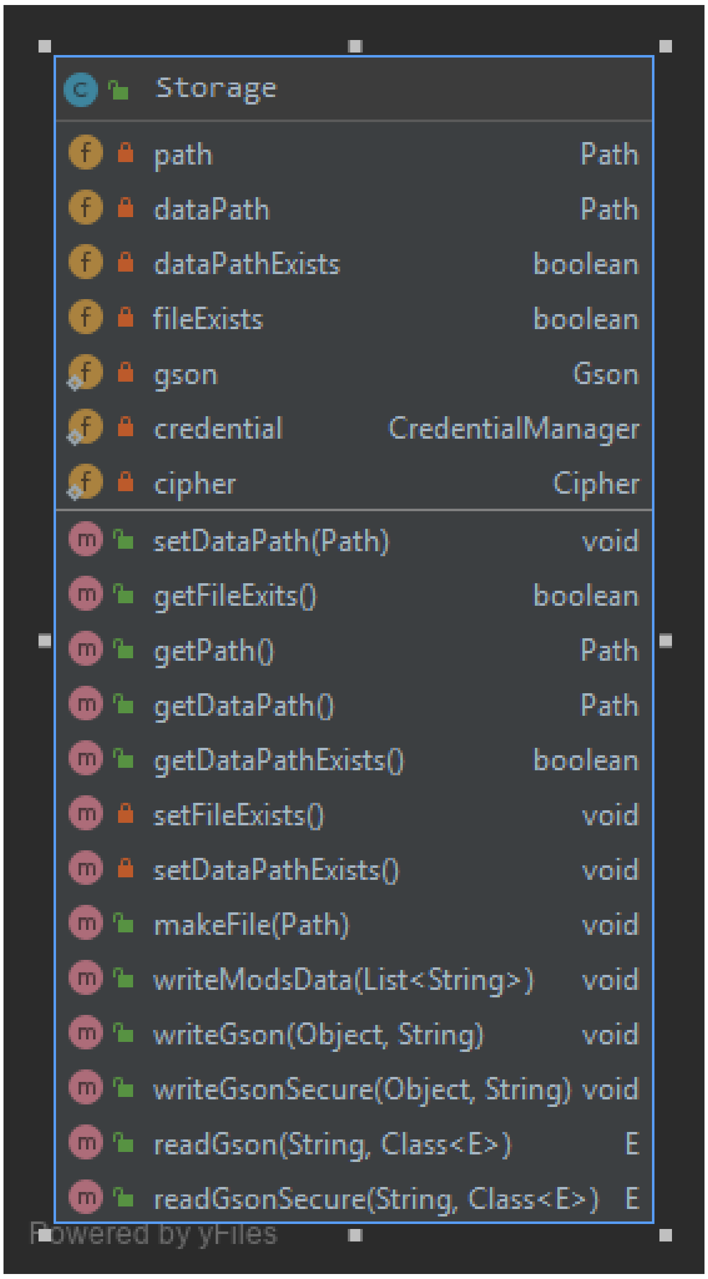
The main wrapper over requesting data from NUSMODS API, Gson library for JSON parsing. Dependant on Storage class for writing data files obtained.



This class is also dependant on a RequestData class which serves as the main API caller for ModPlanner, but it is not used anywhere else in the code base.

3.3. Storage

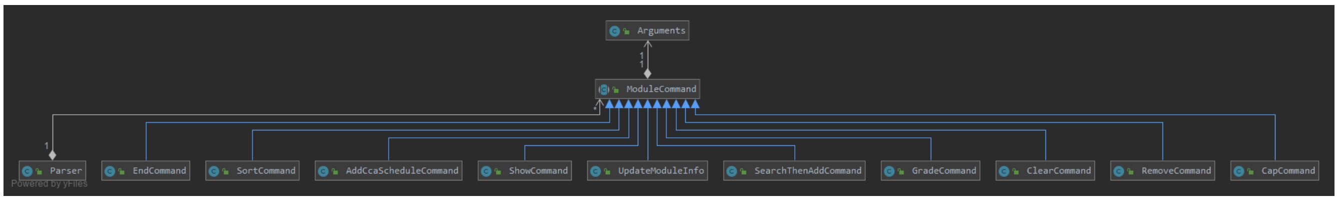
Main class which handles file writing during runtime and for maintaining user data in memory.



This class was adapted from the existing Duke storage to suit the needs of ModPlanner.

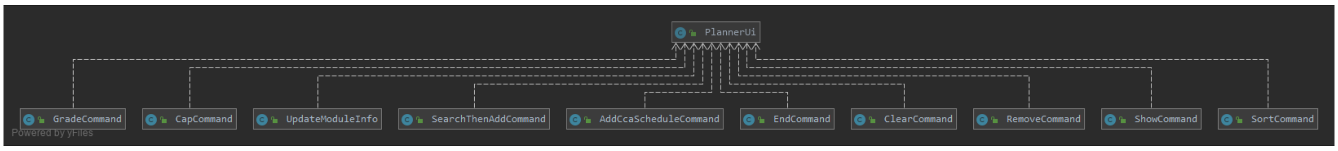
3.4. Parser

Parser is implemented as `Parser` class (formerly named `Argparse4jWrapper`) which is a wrapper for the `argparse4j` library to handle user input and manage internal command mapping.



3.5. PlannerUi

The main class which handles user display, which includes reading user input and printing module, CAP and CCA information back to the user on command-line.



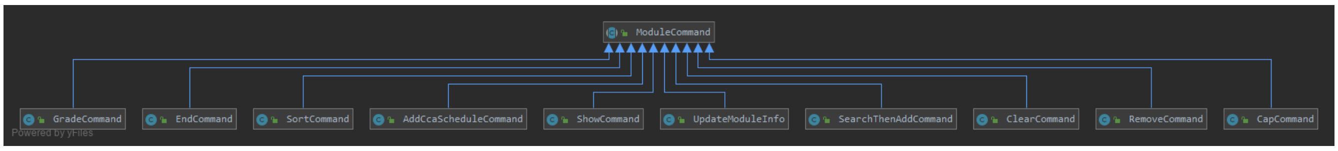
3.6. Command

Commands are the main classes with which we execute our features. All of the specific `Command` classes inherit the base `ModuleCommand` abstract class, and utilize its `execute` method.

The execute method takes in the detailed map of modules drawn from the NUSMODS API Json file, module task list, cca list, planner UI, storage, and the jsonWrapper parser. Each execute method is overriden in the specific Command classes to do what the feature requires of the class.

The `ModuleCommand` class also receives arguments based on the arguments passed by the `Parser` class, which is our main parser for the program, detailed above in Section 3.4, “Parser”.

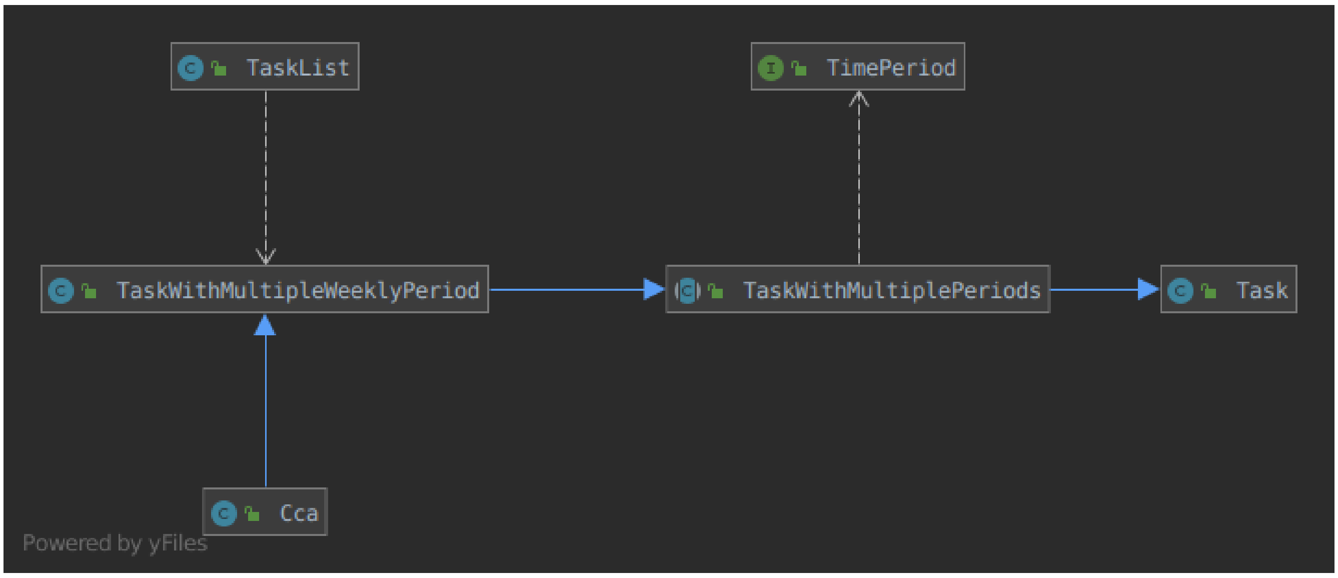
These arguments change how some of the classes function as well, using a switch case system to sort through the different argument types received. These will changes will be explained further under the Implementation section here Section 4, “Implementation”.



The above diagram shows the various Command classes in ModPlan, as well as their depedencies (mostly on `ModuleCommand`).

3.7. CCA

CCA is implemented as `Cca` class which inherits from the legacy `TaskWithMultipleWeeklyPeriod`. It is the backbone for representing and managing user-defined CCAs. Typically, CCAs are loaded and stored in an `List`-like structure implemented in `TaskList<Cca>` class which inherits from `ArrayList<TaskWithMultipleWeeklyPeriod>`.



4. Implementation

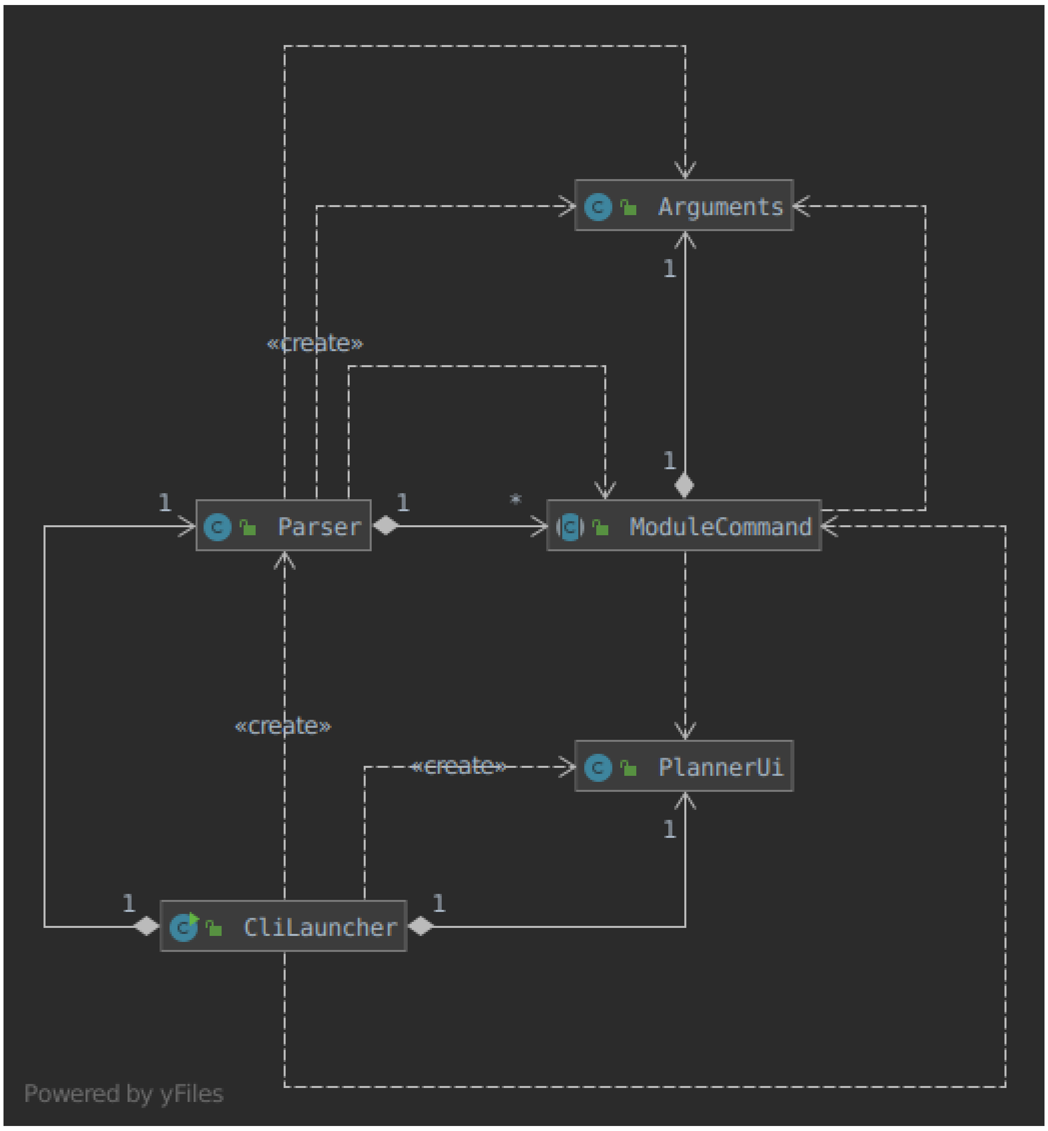
This section explains and shows diagrams of how certain features of ModPlan are implemented.

4.1. Parser

User inputs are recorded by `PlannerUi` and then handled and interpreted by `Parser` class, which is a wrapper for the `argparse4j` library for customized parsing behaviours such as command calling or custom `Action` definition.

4.1.1. Current implementation

The following diagram illustrates the relation of `Parser` to various other core classes:



Attributes

`Parser` (formerly `Argparse4jWrapper`) relies on two major `private` variables `commandMapper` which has the type `HashMap<String, Class<? extends ModuleCommand>>` and `parser` which is an instance of `ArgumentParser`.

Additionally, `Parser` also has attribute `subParserManager` which is the `Subparsers` object associated with `parser` and `subParsers` which is of type `HashMap<String, Subparser>`. Any `Subparser` object added by `subParserManager` should be added to `subParser` as new value with its name as the key for ease of retrieval.

- `parser` is the parent parser for all user inputs, any commands implemented must be parsed using a `Subparser` object added by `subParserManager` as described above.
- `commandMapper` is the `HashMap` which maps the command name (which is often the same as the `subParser` 's name and the key in `subParser`) to the corresponding `ModuleCommand` class.

`Parser` also support custom `Action` objects for custom parsing behaviours which implementation and usage details can be found in `argparse4j` 's documentation.

Methods

There are several "init" methods which would be called on every new instance of `Parser`:

- `private void initBuiltinActions()` initializes all builtin `Action` objects necessary for normal parsing of builtin commands.
- `private void mapBuiltinCommands()` maps builtin commands and corresponding classes using `commandMapper` as described in Section 4.1.1.1, “Attributes”.
- `private void initBuiltinParsers()` initializes all `Subparser` objects for builtin commands.
- `private void mapBuiltinParserArguments()` specifies all arguments and properties for `Subparser` objects generated by `initBuiltinParsers`.

Operation Overview

`Parser` provides several methods for parsing inputs:

- `parseCommand` takes in a `String` object as argument and returns an object that inherits from `ModuleCommand` (dynamic type) corresponding to the input received or `null` if the input is invalid. If any `ModuleCommand` object invoked `throw` a `ModException` object, the exception will be re-`throw`. Else, the corresponding `ModuleCommand` object is invoked by the `invokeCommand()` method This is the method used by `ModPlanner` to parse commands and invoke the corresponding `ModuleCommand` classes.
- `parse` returns a `Namespace` object containing all the parsed arguments instead. It has several overloaded implementations:
    - `public Namespace parse(String[] args)`
    - `public Namespace parse(String userInput)`
    - `public Namespace parse(String subParserName, String[] args)`
    - `public Namespace parse(String subParserName, String userInput)`

`subParserName`, if specified when calling, will tell `Parser` to look for the corresponding `Subparser` object associated with `parser` as mapped in `subParsers` for parsing instead of using `parser`.

Parsing errors are handled by a `private handleError` method and only the logs are printed to `stdout`.

Note that during execution of `parseCommand`, to initialize a `ModuleCommand` object, `Parser` will initialize an `Arguments` object from the parsed `Namespace` object to feed into the `ModuleCommand` object's constructor. However, `Arguments` is not a required class for `Parser` 's `parse` method to work.

Operation Explanation

Below is a step-by-step explanation of `Parser` 's operation principles:

Step 1. User launches the application. An instance of `Parser` will be initialized inside the `CliLauncher` class.

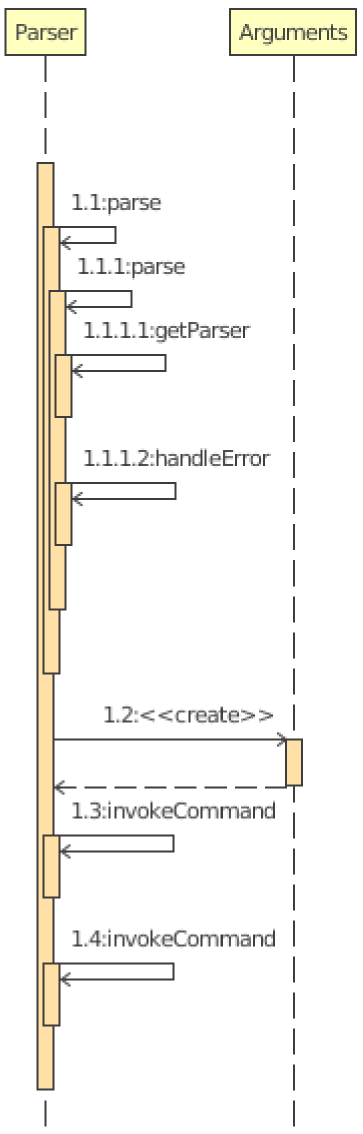
Step 2. User inputs a command to the command line, `PlannerUi` will capture the command and pass over to `Parser` 's `parse` method for interpretation.

Step 3. `Parser` receives the command, matches it against a suitable `ModuleCommand` class and parses all required arguments. If the command is invalid or missing any required arguments, `Parser` will print the errors to the command line using `PlannerUi` with corresponding helps and command suggestions. This process lies in the `handleError` method. If such errors happen, go back to Step 2.

Step 4. Upon successful command matching and arguments extracting, `Parser` initializes an instance of `Arguments` with a `Namespace` object containing parsed arguments as parameters and make a call to the Constructor of the corresponding `ModuleCommand` child class with the `Arguments` object as parameter.

Step 5. If the call succeeds, `Parser` will return the parsed `ModuleCommand` object. Any exceptions thrown during this call will be caught by `Parser` and either converted to a suitable `ModException` object or logged to the console.

A sequence diagram for `parseCommand` can be found below.



The `ModuleCommand` class is not seen in the diagram because `Parser` might not call any `ModuleCommand` -type objects. The only guarantee is that an object with class that inherits from `ModuleCommand` will be returned.

4.1.2. Design consideration

- **Pros**: Robust and reliable, includes helps and commands suggestion, also very scalable.
- **Cons**: No easy way to format help messages.

As the `argparse4j` 's library itself is very robust and powerful, we currently have no plan to modify `Parser` 's implementation. However, addition of custom `Action` will be considered as currently only `Join` is being used for parsing of multiple words `String` arguments.

4.2. Grading

4.2.1. Overview

Grading is a feature implemented to allow users to store their letter grades with the modules they have taken. The letter grades are parsed and its validity deemed according to the attributes of the modules they are storing it in. For example, if a module can be S/U-ed then the feature will allow user’s to grade that module with a 'S' or 'U' grade.

4.2.2. Current Implementation

The `grade` feature is operated by the `GradeCommand` class, which is called by the `Parser` class. Upon user input of `grade MODULECODE LETTERGRADE`, the Parser will return a new `GradeCommand`.

Since `GradeCommand` inherits the `ModuleCommand` class, it must override the `execute` method to specially execute the `grade` command. From the `Parser`, `GradeCommand` also receives two additional variable inputs from the user:

1. The module code of the module to be graded.
2. The letter grade attained for the module that the user specifies above.

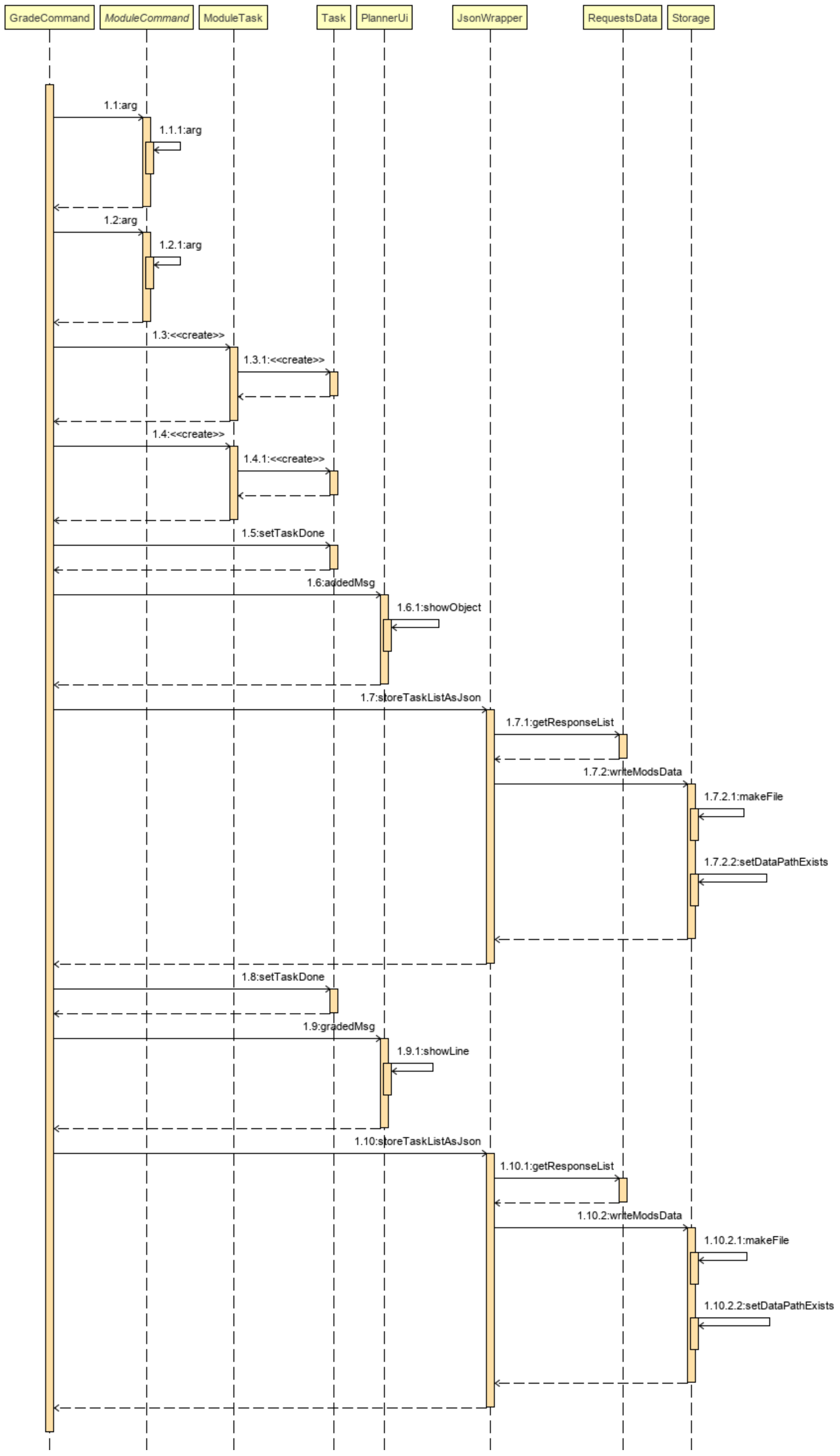
There are two ways that the execute method can execute, depending upon whether the moduleCode the user enters is in their moduleTaskList or not. In both cases the updated grade is stored along with the module details in the `Storage` package.

- Case 1: Module is not in the task list
  If the module is not in the task list, `GradeCommand` executes in a similar fashion to `SearchThenAddCommand`, creating a temporary `ModuleInfoDetailed` class to check if the moduleCode entered by the user exists or not.
  Following which, the letterGrade of the module is set using the method `setGrade` under the `ModuleInfoDetailed` class.
  - **Note:** The `setGrade` method will check if the letterGrade input by the user is valid (a valid letter grade, as well as S/U capabilities).

    If the letter grade is invalid, either `ModModBadSUException` or `ModBadGradeException` will be thrown.

    Finally, the temporary module will be added to the `ModuleTaskList`, with the `letterGrade` included in its details.
- Case 2: Module is in the task list If the module already exists in the task list, (i.e `ModuleTaskList` contains `moduleCode`) `GradeCommand` will simply check if the module can be S/U-ed, and update the `letterGrade` according to what the user inputs using the `setGrade` method.

Below is a Sequence Diagram showing how `GradeCommand` works.



4.2.3. Design Considerations

How GradeCommand executes

Checks had to be implemented to check if the module can be S/U-ed, as well as if the `letterGrade` the user inputs is a valid grade according to NUS specifications. These checks were implemented into the `ModuleInfoDetailed` class itself, which `ModuleTask` inherits as the baseline of the module task list. This way, other classes are able to use the methods and checks to set the letter grade for the respective module.

- Pros: This checking system can be used by other classes/methods just by calling the method under `ModuleInfoDetailed`.
- Cons: Calling the method requires a much deeper reach to extract the method/check needed, and may impact the code simplicity.

Other alternatives would be to have the checks directly in the `GradeCommand` class, however this alternative was disregarded as other `Command` that required use of these checks were created, such as the `CapCommand` class.

4.3. Cap Calculation

4.3.1. Overview

This feature allows users to calculate their Cumulative Average Point (NUS's version of GPA) from their inputted data. It uses various external data, as well as prerequisite and preclusion module checks to determine the user's CAP in three different ways.

4.3.2. Current Implementation

The `cap` feature is operated by the `CapCommand` class, which is called by the `Parser` class. Upon user input of `cap TYPE`, the Parser will return a new `CapCommand`.

Since `CapCommand` inherits the `ModuleCommand` class, it must override the `execute` method to specially execute the `cap` command.

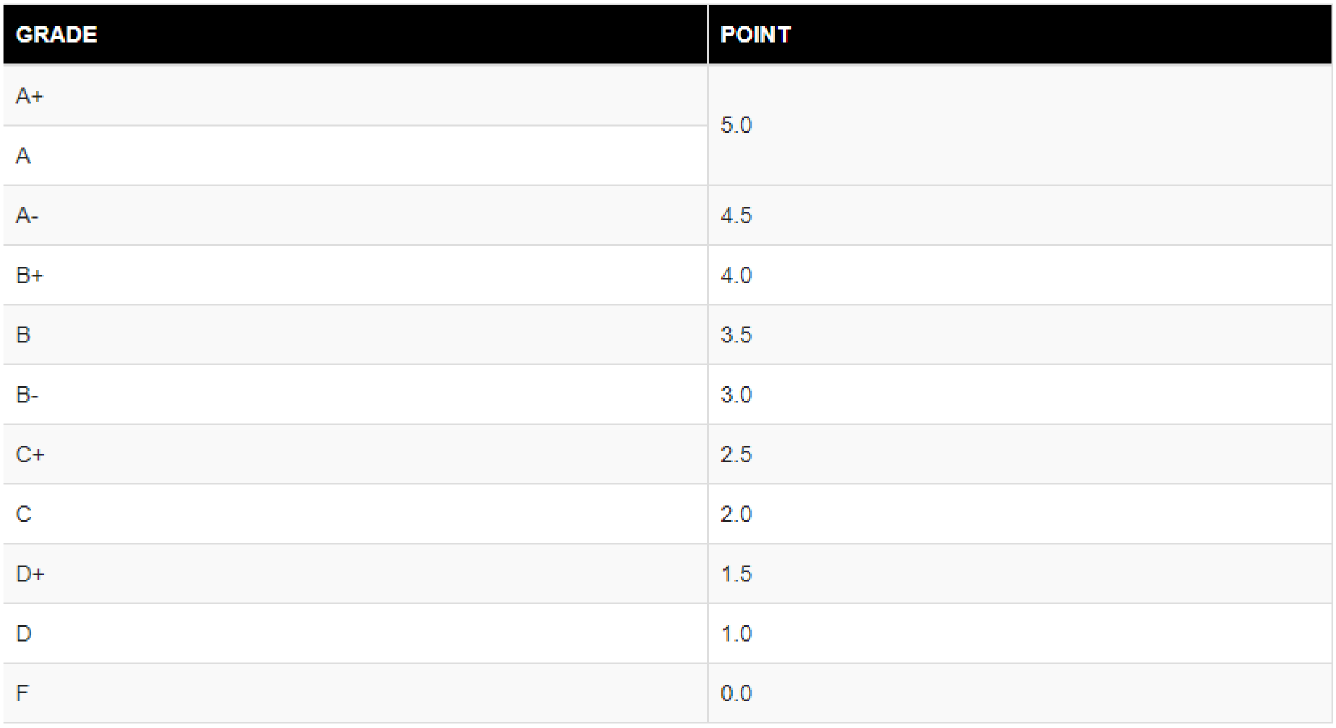
The parameter `TYPE` can take three forms according to the user input:

- `cap overall` Where the user inputs modules of their choosing, as well as the letter grade, and the CAP is calculated accordingly.
- `cap list` Where the user's CAP is calculated from the modules with letter grades in the module task list.
- `cap module` Where the CAP of a module of the user's choosing can be calculated using the grades of prerequisite modules that the user has completed.

These `TYPE` parameters will be parsed by the `Parser` class and pass the corresponding argument of `toCap` into the `CapCommand` class. A switch case statement will handle the `toCap` argument, and choose to execute from three methods accordingly:
`calculateOverallCap`, `calculateListCap` and `calculateModuleCap`

Upon construction of the `CapCommand` class, a few variables involved in calculating the CAP of the user are initialized, notably the users `mcCount`, `currentCap`, `projectedModuleCap` and `projectedCap`. These variables will be used in the three different ways CapCommand can currently execute in.

The user’s CAP is calculated according to NUS guidelines, following the below specifications:



GRADE	POINT
A+	5.0
A	
A-	4.5
B+	4.0
B	3.5
B-	3.0
C+	2.5
C	2.0
D+	1.5
D	1.0
F	0.0

$$\text{CAP} = \frac{\text{sum (module grade point x MCs assigned to module)}}{\text{sum (MCs assigned to all modules used in calculating the numerator)}}$$

As stated above, there are three methods that can be executed depending upon the `TYPE` the user inputs.

- Case 1: `cap overall`
  If the argument read for `toCap` is "overall", the `calculateOverallCap` method will be executed under the `execute` method.
  Firstly, a new `Scanner` will be created to continue reading in the modules and grades that the user wishes to calculate their CAP for.
  The user will be prompted to input a module and its respective letter grade.
  The user inputs are read in until the user inputs `done`, proceeding which the scanner will close and the calculation is done.
  Finally the user’s CAP is calculated and printed.
- Case 2: `cap list`
  If the argument read for `toCap` is "list", the `calculateListCap` method will be executed under the `execute` method.
  This method calculates the CAP of modules from the user’s `ModuleTaskList`.
  - Note it will only take into account modules that have a letter grade attached to its details, and calculate the CAP accordingly.
- Case 3: `cap module`
  If the argument read for `toCap` is "module", the `calculateModuleCap` method will be executed under the `execute` method.
  This method calculates the CAP of modules from the user’s completed prerequisites in their `ModuleTaskList`.
  Firstly, a new `Scanner` will be created to continue reading in the module that the user wishes to calculate a predicted CAP for.
  After taking in the user input, ModPlan will check if the module is a legitimate module in the `detailedMap` pulled from NUSMODS API.
  If it is invalid, a new `ModNotFoundException` will be thrown. Otherwise the prerequisite tree (if any) will be scanned for that particular module using the `parsePrerequisiteTree` method.

For the `cap module` command, we came up with two ways to go about parsing the prerequisite tree of the module whose CAP is to be predicted.

Alternative 1: Array of Module Codes (currently implemented) Currently in v1.4 the `cap module` method only parses the first level of prerequisites that a module has, as well as their preclusions, and checks against the user’s module task list.

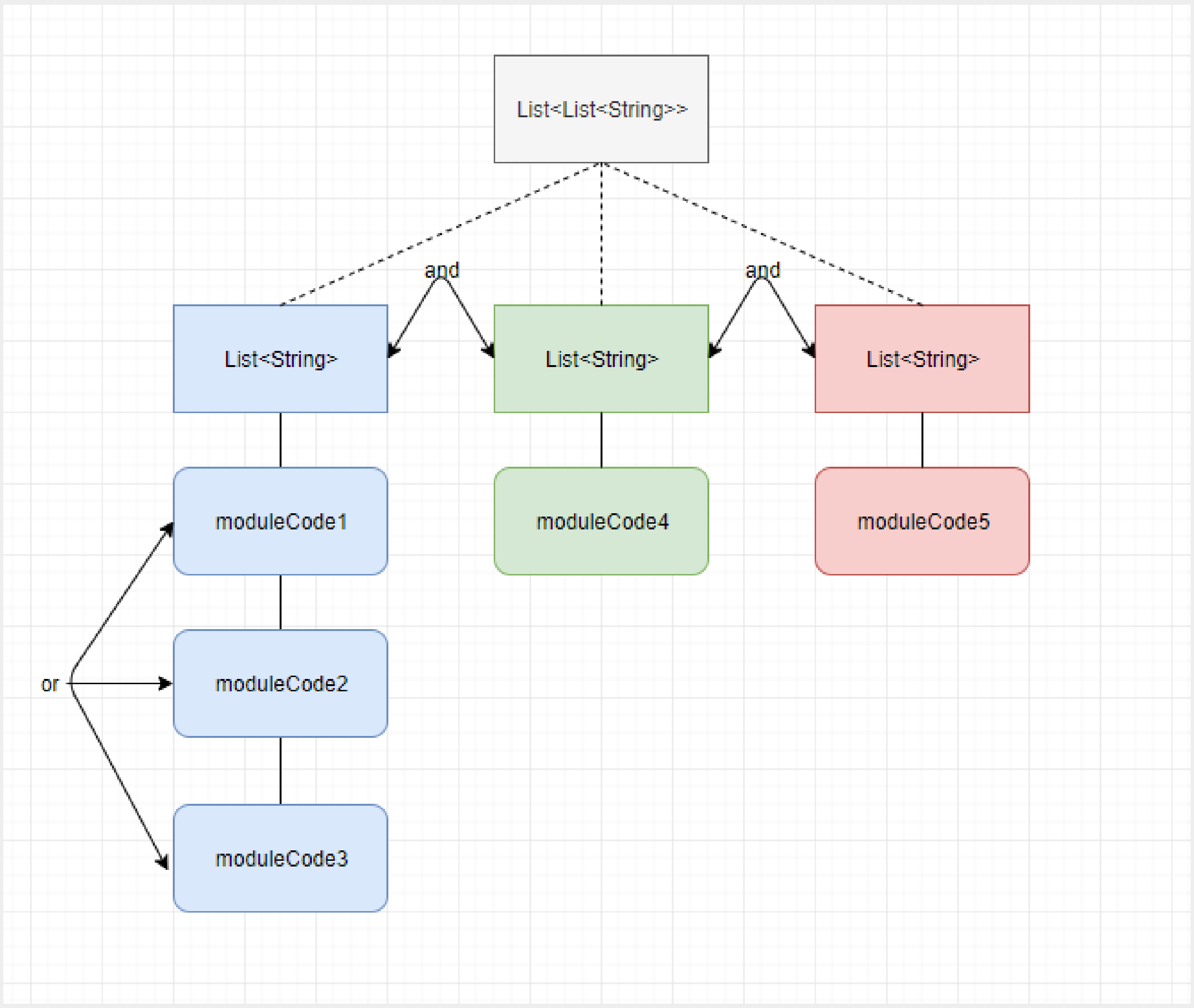
If there are any prerequisite modules taken for the predicted module, their grades will be taken to predict the CAP for the inputted module. The array of module codes is rechecked again for any remaining preclusions that may have been omitted, before finally calculating and printing the CAP for the user.

Alternative 2: List of List of Strings (coming in v2.0) This method uses the string split method to parse the string of prerequisites into individual module codes, and sorts them into a List of Lists of Strings (LLS).

Each List of Strings (LS) contains prerequesite modules as part of an 'or' tree, while the modules across the LLS are part of an 'and' tree. Once the methods finds one of the prerequisite modules in a LS that corresponds to a graded module taken in the user’s module task list, it removes that LS entirely from the LLS, and moves on to check the next LS for any prerequisite modules taken.

If the entire LLS is empty at the end of the execution, it means that the user has fulfilled enough of the prerequisite modules required for that module, and the user's CAP is calculated according to the graded prerequisite modules identified in the user's module task list.

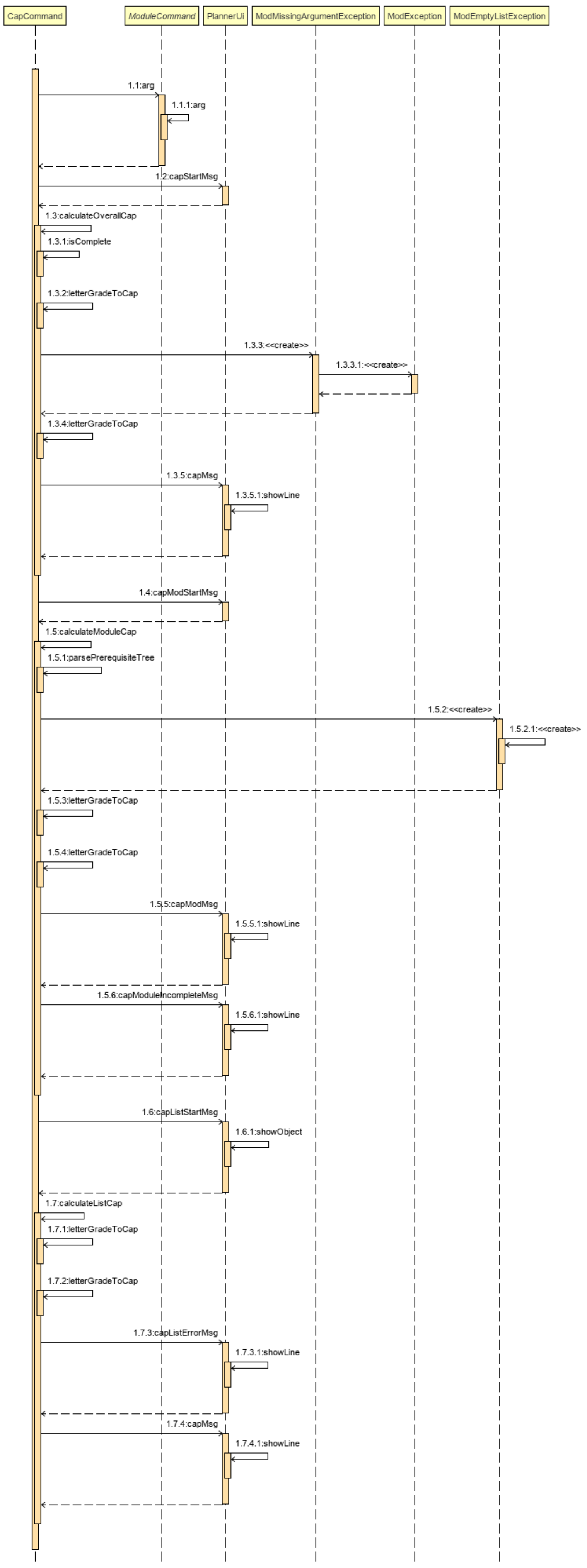
The diagram below shows the example more clearly, where only one of the prerequisites within a LS need to be taken, while all of prerequisites across the LLS need to be taken.



In this case, the modules the user needs to take are:

- One of moduleCode1, moduleCode2, moduleCode3
- moduleCode4
- moduleCode5

Below is a sequence diagram showing how `CapCommand` works.


CapCommand
ModuleCommand
PlannerUi
ModMissingArgumentException
ModException
ModEmptyListException
1.1:arg
1.1.1:arg
1.2:capStartMsg
1.3:calculateOverallCap
1.3.1:isComplete
1.3.2:letterGradeToCap
1.3.3:<<create>>
1.3.3.1:<<create>>
1.3.4:letterGradeToCap
1.3.5:capMsg
1.3.5.1:showLine
1.4:capModStartMsg
1.5:calculateModuleCap
1.5.1:parsePrerequisiteTree
1.5.2:<<create>>
1.5.2.1:<<create>>
1.5.3:letterGradeToCap
1.5.4:letterGradeToCap
1.5.5:capModMsg
1.5.5.1:showLine
1.5.6:capModuleIncompleteMsg
1.5.6.1:showLine
1.6:capListStartMsg
1.6.1:showObject
1.7:calculateListCap
1.7.1:letterGradeToCap
1.7.2:letterGradeToCap
1.7.3:capListErrorMsg
1.7.3.1:showLine
1.7.4:capMsg
1.7.4.1:showLine

As seen `CapCommand` uses three different CAP calculation methods contained within itself to generate the CAP reports, in addition to all the sub-methods such as `letterGradeToCap` and `parsePrerequisiteTree`, and then pushes the information to `PlannerUi` for printing. It also instantiates different `Exception` classes to feedback to the user if anything is wrong.

4.4. Sorting

4.4.1. Current implementation

The `sort` feature is operated by the `SortCommand` class, which is called by the `Parser` class. Upon user input of `sort TOSORT TYPE`, the Parser will return a new `SortCommand`.

Since `SortCommand` inherits the `ModuleCommand` class, it must override the `execute` method to specially execute the `SortCommand`. From the `Parser`, `SortCommand` also receives two additional variable inputs from the user:

- The object to sort, e.g. module, cca or time.
- The type of sorting the user specifies. e.g. for modules they can sort by code, grade, level and mcs; and for time they must specify a day of the week.

The parameter `TYPE` can take three main forms according to the user input:

- `sort cca` Where the user receives a sorted list of their ccas in alphabetical order.
- `sort time DAY_OF_WEEK` Where the user receives a list of ccas and/or modules in a timely order for the day of week they choose.
- `sort module BY` Where the user receives a sorted list of their modules in the order they specifies.

These `TYPE` parameters will be parsed by the `Parser` class and pass the corresponding argument of `toSort` into the `SortCommand` class. A switch case statement will handle the `toSort` argument, and choose to execute the corresponding sorting.

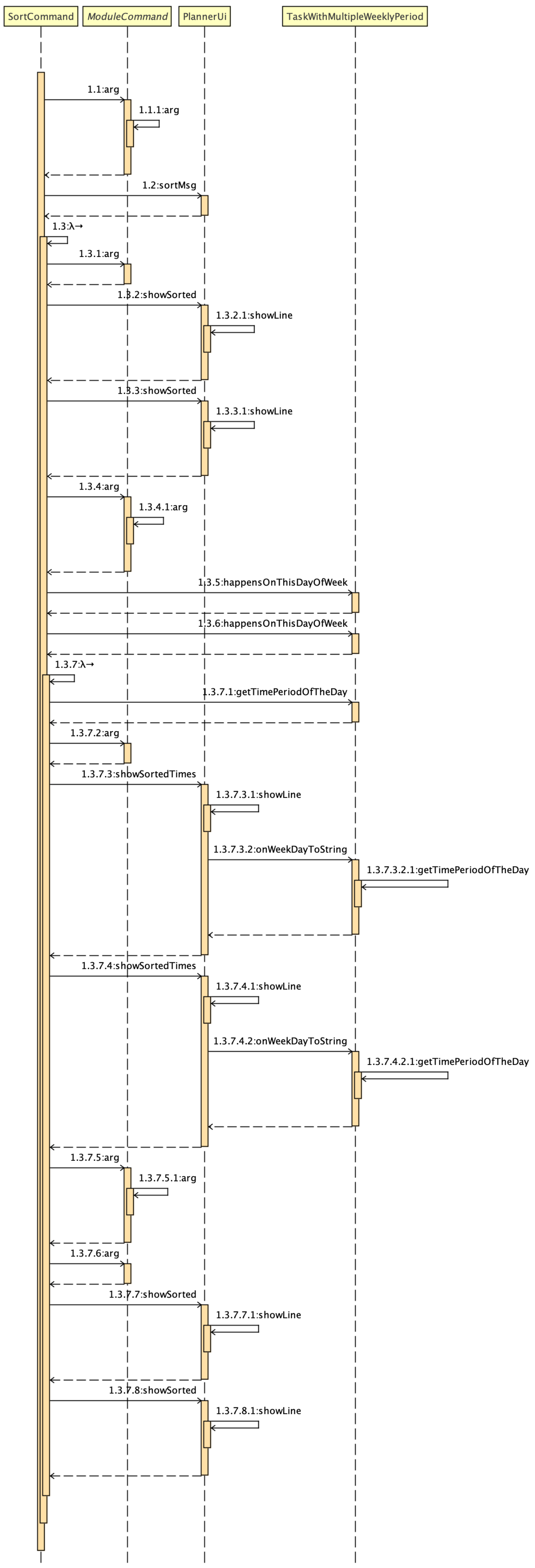
As stated above, there are three forms that can be executed depending upon the `TYPE` the user inputs.

- Case 1: `sort cca`
  If the argument read for `toSort` is “cca”, the cca list will be sorted in alphabetical order, and printed to the users.
- Case 2: `sort time DAY_OF_WEEK`
  If the argument read for `toSort` is “time”, a new list of TaskWithMultipleWeeklyPeriod will be initialized, and all the ccas and modules that happens on the specified DAY_OF_WEEK will be added to the new list. This list is then sorted in ascending order of the starting time of the ccas/modules, and this list is printed to the users.
  - Note it will only take into account modules that was initially given a time period of the week attached to its details.
- Case 3: `sort module BY`
  If the argument read for `toSort` is “module”, another switch case statement will handle the `BY` argument, and choose to execute the corresponding sorting of the modules. The options for `BY` are `code`, `grade`, `level` and `mc`.

For all the above forms, the users may choose to enter an optional flag of `--r` to have all their modules/ccas sorted in the reverse order. This is done by the parser setting the argument as true (default is set as false). Then if the argument is true, the previously sorted list will be reversed then printed to the users.

Below is a Sequence Diagram showing how `SortCommand` works. SortCommand inherits the attributes and methods from the ModuleCommand, which is shown by the 1.1 arg arrow connecting SortCommand and ModuleCommand. The ModuleCommand calls itself as it uses its own attributes method as shown by 1.1.1 args. The SortCommand also calls certain methods of PlannerUi as shown by the arrows from 1.2 and 1.3 args. This is because the SortCommand uses the methods in PlannerUi such as the `sortMsg`, `showSorted` and `showSortedTimes`.

As both the modules and ccas inherit from the class `TaskWithMultipleWeeklyPeriod`, when the users indicate they want to `sort time DAY_OF_WEEK`, the SortCommand calls the `happensOnThisDayOfWeek` method to check whether the module or cca in the list happens on the `DAY_OF_WEEK` specified by the user, then adds to a temporary list. The method `getTimePeriodOfTheDay` from `TaskWithMultipleWeeklyPeriod` is used to sort the modules and ccas in a timely order.


SortCommand
ModuleCommand
PlannerUi
TaskWithMultipleWeeklyPeriod
1.1:arg
1.1.1:arg
1.2:sortMsg
1.3:λ→
1.3.1:arg
1.3.2:showSorted
1.3.2.1:showLine
1.3.3:showSorted
1.3.3.1:showLine
1.3.4:arg
1.3.4.1:arg
1.3.5:happensOnThisDayOfWeek
1.3.6:happensOnThisDayOfWeek
1.3.7:λ→
1.3.7.1:getTimePeriodOfTheDay
1.3.7.2:arg
1.3.7.3:showSortedTimes
1.3.7.3.1:showLine
1.3.7.3.2:onWeekDayToString
1.3.7.3.2.1:getTimePeriodOfTheDay
1.3.7.4:showSortedTimes
1.3.7.4.1:showLine
1.3.7.4.2:onWeekDayToString
1.3.7.4.2.1:getTimePeriodOfTheDay
1.3.7.5:arg
1.3.7.5.1:arg
1.3.7.6:arg
1.3.7.7:showSorted
1.3.7.7.1:showLine
1.3.7.8:showSorted
1.3.7.8.1:showLine

4.5. Showing Module Reports

4.5.1. Current implementation

The `show` feature is opearted by the `ShowCommand` class, which is called by the `Parser` class. Upon user input of `show TYPE`, the Parser will return a new `ShowCommand`.

Since `ShowCommand` inherits the `ModuleCommand` class, it must override the `execute` method to specially execute the `show` command.

The parameter `TYPE` can take five forms according to the user input:

- `show cca`
- `show core`
- `show ge`
- `show ue`
- `show module`

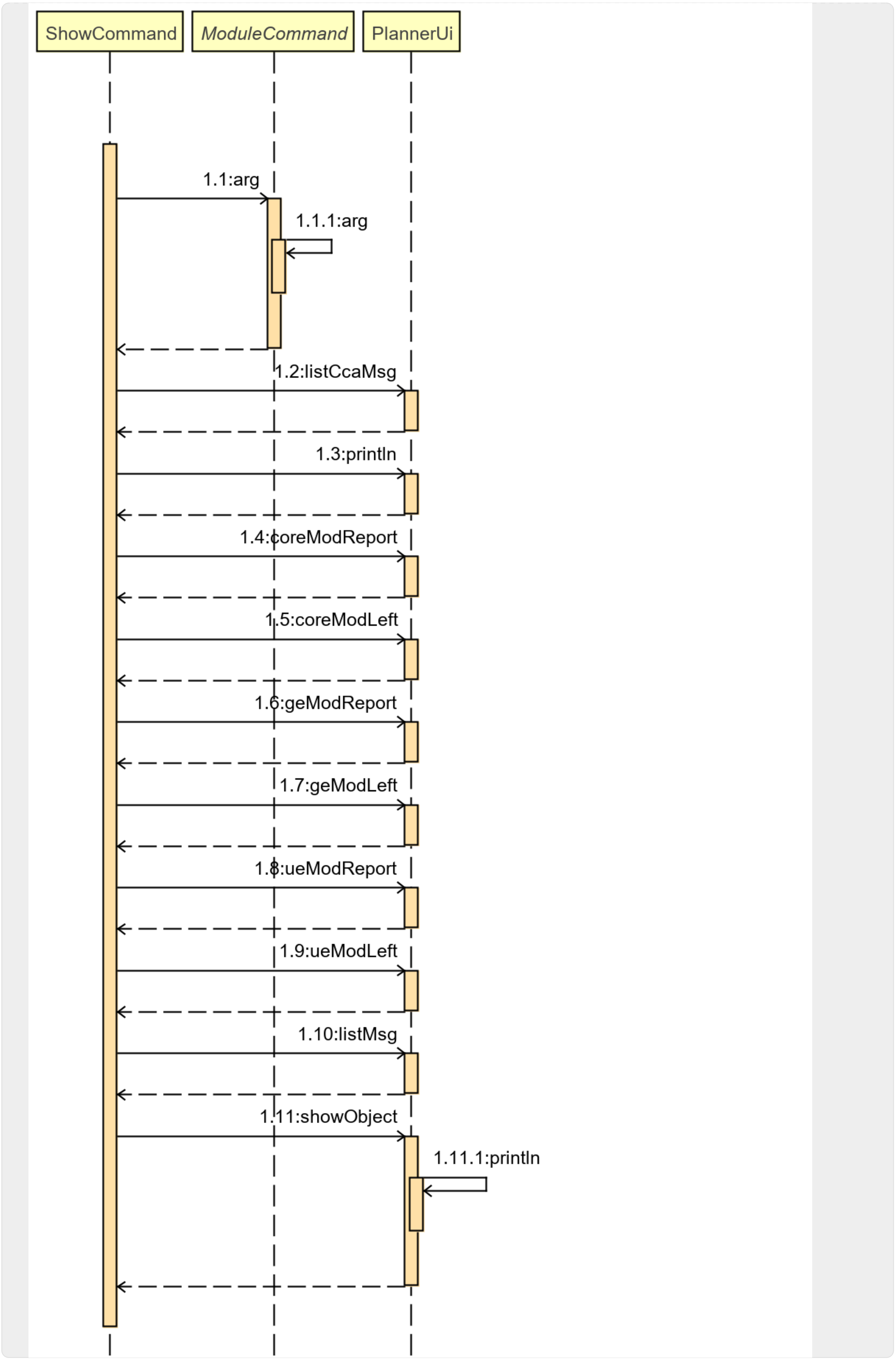
These `TYPE` parameters will be parsed by the `Parser` class and pass the corresponding argument of `toShow` into the `ShowCommand` class. A switch case statement will handle the `toShow` argument.

Upon construction of the `ShowCommand` class, one variable involved in the generation of the report is initiazlised, which refers to the `coreModList`. This variable is being used to store the strings of core modules taken by CEG students throughout the four years. The list of core modules are being taken from the NUS module requirements website.

As stated above, there are five methods that can be executed depending on the `TYPE` that the user inputs.

- Case 1: `show cca`
  If the argument read for `toShow` is "cca", an iterator iterates through the array list of `ccas`, which includes the CCAs added, and prints it out.
- Case 2: `show core`
  If the argument read for `toShow` is "core", an iterator loops through the array list of `ModuleTask` and checks it against the set of `coreModList`. If the `moduleCode` in `ModuleTask` matches the `moduleCode` from the `coreModList`, that `moduleCode` is being printed out.
  - The second part subtracts the number of core modules taken from the total number of core modules required to be taken, which the information is taken from the NUS module requirements website. Hence, it shows the users the number of core modules left to take for the rest of the years.
- Case 3: `show ge`
  If the argument read for `toShow` is "ge", an iterator loops through the array list of `ModuleTask` and checks the starting two letters of the `moduleCode`. If the `moduleCode` starts with a "GE", it is classified as a General Elective(GE) module and will be printed out.
  - The second part subtracts the number of GE modules taken from the total number of GE modules required to be taken, which the information is taken from the NUS module requirements website. Hence, it shows the users the number of GE modules left to take for the rest of the years.
- Case 4: `show ue`
  If the argument read for `toShow` is "ue", an iterator loops through the array list of `ModuleTask`. It checks the starting two letters of the `moduleCode` and also checks the `moduleCode` against the set of `coreModList`. If the `moduleCode` does not match the `moduleCode` from the `coreModList` and it does not start with a `GE`, it is classified as a Unrestricted Elective(UE) module and will be printed out.
  - The second part subtracts the number of UE modules taken from the total number of UE modules required to be taken, which the information is taken from the NUS module requirements website. Hence, it shows the users the number of UE modules left to take for the rest of the years.
- Case 5: `show module`
  If the argument read for `toShow` is "module", an iterator loops through the array list of `ModuleTask`. It then prints out the entire list of modules being added.

Below is a Sequence Diagram showing how `ShowCommand` works.
ShowCommand inherits the attributes and methods from the ModuleCommand, which is shown by the 1.1 arg arrow connecting ShowCommand and ModuleCommand. The ModuleCommand calls itself as it uses its own attributes method as shown by 1.1.1 arg. The ShowCommand also calls certain methods of PlannerUi as shown by the arrows from 1.2 to 1.11 args. This is because the ShowCommand uses the methods in PlannerUi such as the `listCcaMsg` and `coreModReport`.


ShowCommand
ModuleCommand
PlannerUi
1.1:arg
1.1.1:arg
1.2:listCcaMsg
1.3:println
1.4:coreModReport
1.5:coreModLeft
1.6:geModReport
1.7:geModLeft
1.8:ueModReport
1.9:ueModLeft
1.10:listMsg
1.11:showObject
1.11.1:println

4.6. Update Reminders

4.6.1. Current implementation

The `reminder` feature is operated by the `ReminderCommand` class, which is called by the `Parser` class. Upon user input of `reminder TYPE`, the Parser will return a new `ReminderCommand`.

Since `ReminderCommand` inherits the `ModuleCommand` class, it must override the `execute` method to specially execute the `reminder` command.

The parameter `NUMBER` can take six forms according to the user input:

- `reminder list`
- `reminder one`
- `reminder two`
- `reminder three`
- `reminder four`
- `reminder stop`

These `TYPE` parameters will be parsed by the `Parser` class and pass the corresponding argument of `toReminder` into the `ReminderCommand` class. A switch case statement will handle the `toReminder` argument.

Upon construction of the `ReminderCommand` class, variables using Timer class and ScheduledTask class are inititalised.

As stated above, there are six methods that can be executed depending on the `TYPE` that the user inputs.

- Case 1: `reminder list`
  If the argument read for `toReminder` is "list", the reminderList() method in plannerUI class is called. A list of the different reminder interval options are being printed. The four different options allow user to determine how often the user want to set the update reminder message. User can set the reminder message to pop up every 10 seconds, 30 seconds, 1 minute or 2 minutes.

- Case 2: `reminder one`
  If the argument read for `toReminder` is "one", the printEveryTenSec() method is called. The `time` variable initialised by the Timer class is used to schedule the reminder message in ScheduledTask class every 10000ms, which equates to 10 seconds.

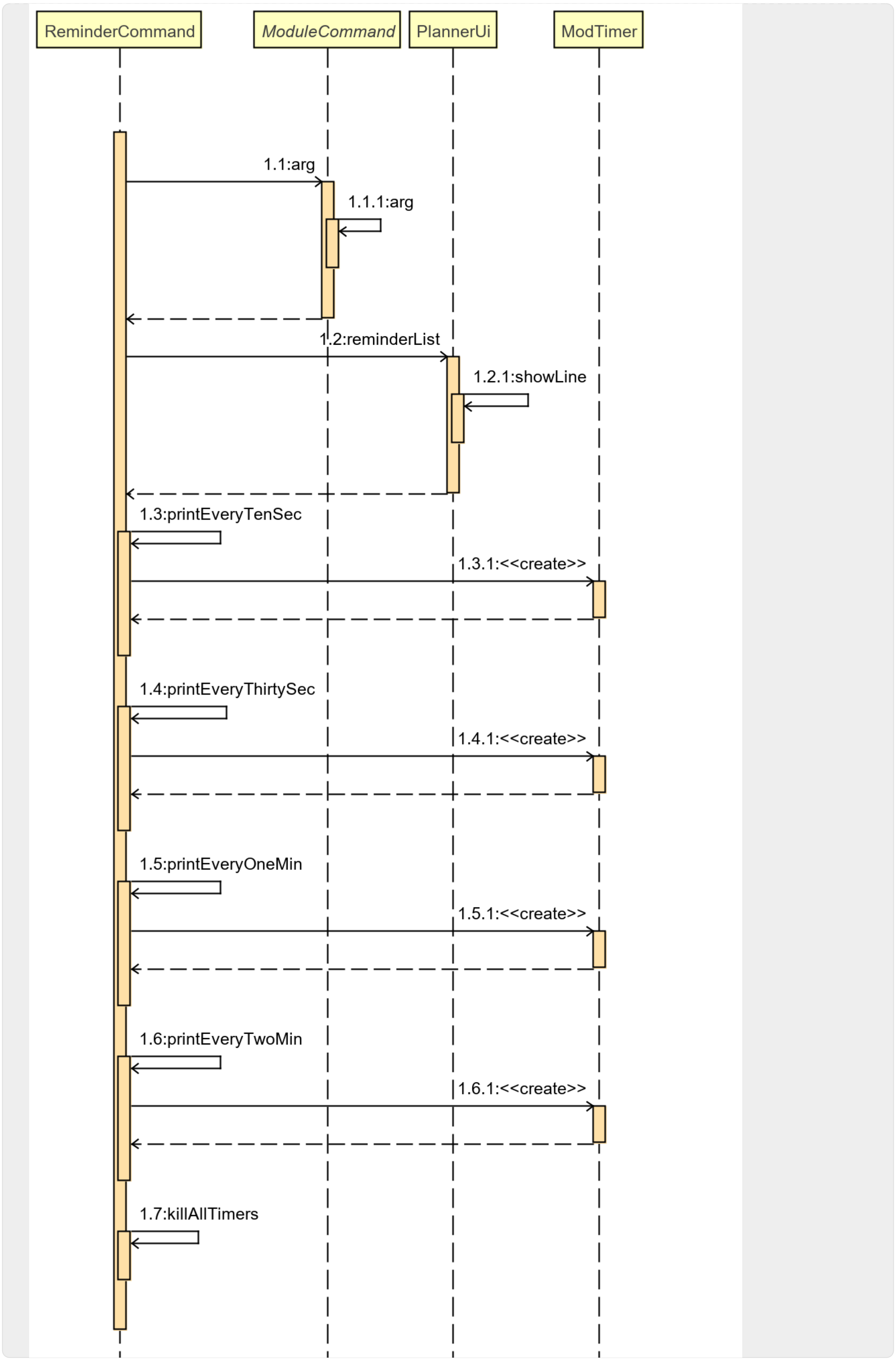
- Case 3: `reminder two`
  If the argument read for `toReminder` is "two", the printEveryThirtySec() method is called. The `time` variable initialised by the Timer class is used to schedule the reminder message in ScheduledTask class every 30000ms, which equates to 30 seconds.

- Case 4: `reminder three`
  If the argument read for `toReminder` is "three", the printEveryOneMin() method is called. The `time` variable initialised by the Timer class is used to schedule the reminder message in ScheduledTask class every 60000ms, which equates to 1 minute.

- Case 5: `reminder four`
  If the argument read for `toReminder` is "four", the printEveryTwoMin() method is called. The `time` variable initialised by the Timer class is used to schedule the reminder message in ScheduledTask class every 120000ms, which equates to 2 minutes.

- Case 6: `reminder stop`
  If the argument read for `toReminder` is "stop", the killAllTimers() method is called. A Timer iterator loops through the list of all the Timer threads and cancels all the Timer tasks. A message is then printed to notify the user that the reminder message for the update is being stopped.

Below is a Sequence Diagram showing how `ReminderCommand` works.
ReminderCommand inherits the attributes and methods from the ModuleCommand, which is shown by the 1.1 arg arrow connecting ReminderCommand and ModuleCommand. The ModuleCommand calls itself as it uses its own attributes method as shown by 1.1.1 arg. The ReminderCommand also calls the reminderList method in PlannerUi as shown by the 1.2 arrow. The reminderList method, in turn calls itself by using the showLine method as shown by 1.2.1 arrow. The ReminderCommand calls its own method, such as printEveryTenSec as shown by the 1.3 arrow. It then uses an object in ModTimer class, which is removed from the memory, as shown by the 1.3.1 arrow. This repeats three more times as shown by the 1.4 to 1.6 arrows. Lastly, the ReminderCommand calls itself by using the killAllTimers method as shown by the 1.7 arrow.


ReminderCommand
ModuleCommand
PlannerUi
ModTimer
1.1:arg
1.1.1:arg
1.2:reminderList
1.2.1:showLine
1.3:printEveryTenSec
1.3.1:<<create>>
1.4:printEveryThirtySec
1.4.1:<<create>>
1.5:printEveryOneMin
1.5.1:<<create>>
1.6:printEveryTwoMin
1.6.1:<<create>>
1.7:killAllTimers

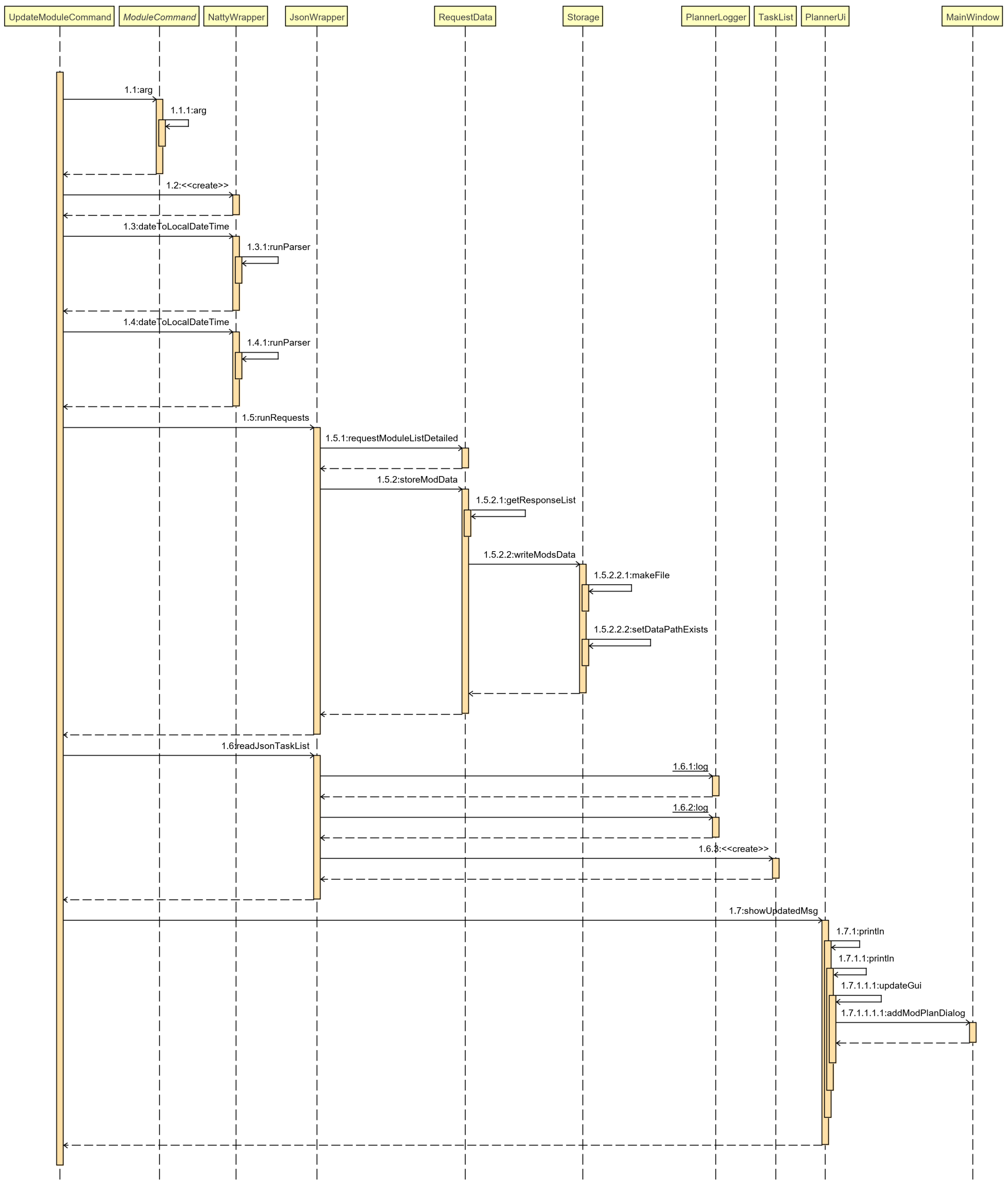
4.7. UpdateModuleCommand

4.7.1. Current implementation

The `update` feature is executed by the `UpdateModuleCommand` class, which is called by the `Parser` class. Upon user input of `update module`, the Parser will return a new `UpdateModuleCommand`.

Since `UpdateModuleCommand` inherits the `ModuleCommand` class, it must override the `execute` method to specially execute the `update` command.

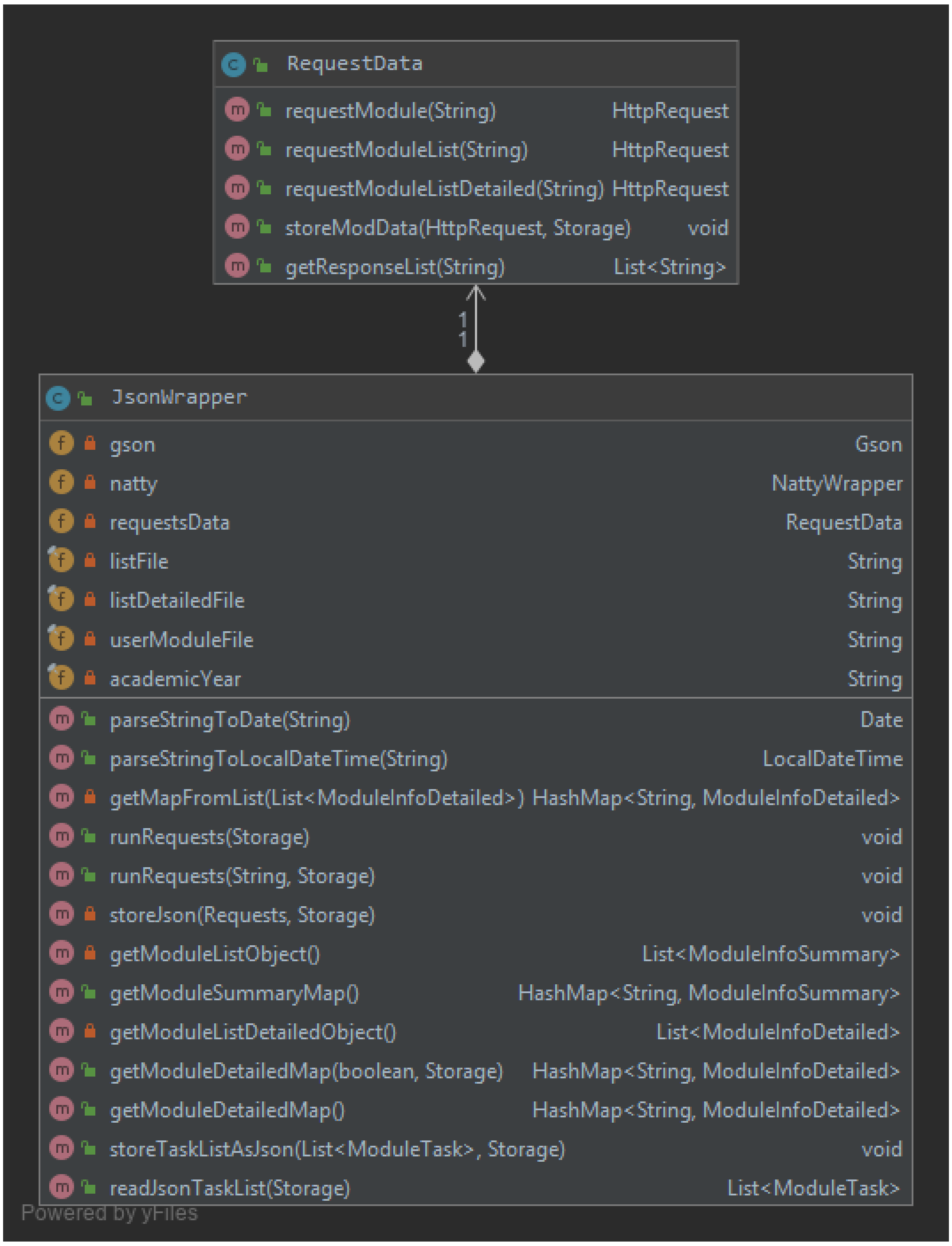
Below is a Sequence Diagram showing how `UpdateModuleInfo` works.
UpdateModuleCommand inherits the attributes and methods from the ModuleCommand, which is shown by the 1.1 arg arrow connecting the UpdateModuleCommand and ModuleCommand. The ModuleCommand calls itself as it uses its own attributes method as shown by 1.1.1 arg. It then uses an object in NattyWrapper class, which is removed from the memory, as shown by the 1.3.1 arrow. The UpdateModuleCommand also calls the dateToLocalDateTime method from the NattyWrapper as shown by the 1.3 and 1.4 arrows. The UpdateModuleCommand fetches the updated module information from the RequestData , Storage and PlannerLogger, as shown by the 1.5 arrows. To end off, the UpdateModuleCommand calls the showUpdateMsg method from the PlannerUi to print the updated message.



4.8. JsonWrapper

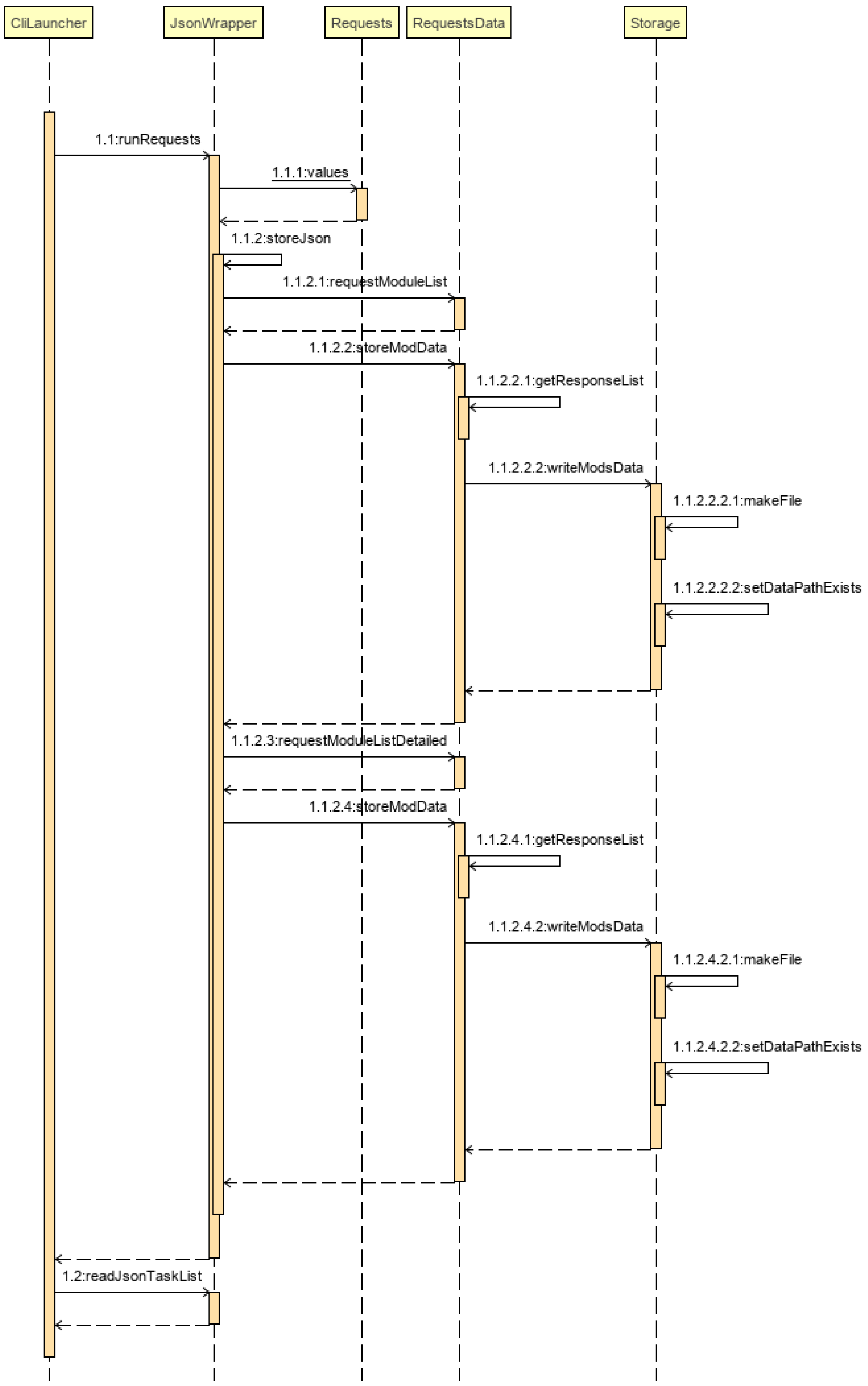
4.8.1. Current implementation

`JsonWrapper` contains our usage of the `Gson` library for JSON file processing, as well as to call `RequestData` to obtained the module data consolidated by `NUSMODS API`.


RequestData
requestModule(String) HttpRequest
requestModuleList(String) HttpRequest
requestModuleListDetailed(String) HttpRequest
storeModData(HttpRequest, Storage) void
getResponseList(String) List<String>
1
1
JsonWrapper
gson Gson
natty NattyWrapper
requestsData RequestData
listFile String
listDetailedFile String
userModuleFile String
academicYear String
parseStringToDate(String) Date
parseStringToLocalDateTime(String) LocalDateTime
getMapFromList(List<ModuleInfoDetailed>) HashMap<String, ModuleInfoDetailed>
runRequests(Storage) void
runRequests(String, Storage) void
storeJson(Requests, Storage) void
getModuleListObject() List<ModuleInfoSummary>
getModuleSummaryMap() HashMap<String, ModuleInfoSummary>
getModuleListDetailedObject() List<ModuleInfoDetailed>
getModuleDetailedMap(boolean, Storage) HashMap<String, ModuleInfoDetailed>
getModuleDetailedMap() HashMap<String, ModuleInfoDetailed>
storeTaskListAsJson(List<ModuleTask>, Storage) void
readJsonTaskList(Storage) List<ModuleTask>
Powered by yFiles

To prevent multiple requests to NUSMODS, our implementation would check if the user has previously downloaded the module data before. If they have not, only then would `JsonWrapper` call `RequestData` to initialize the module data file.

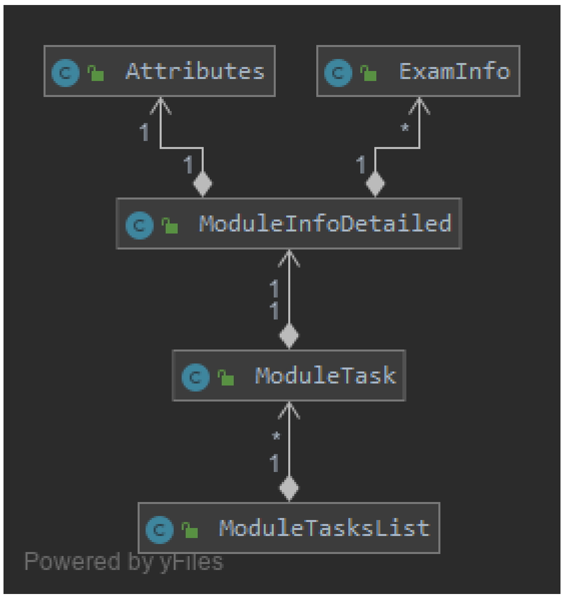
The sequence diagram of this implementation is shown below:



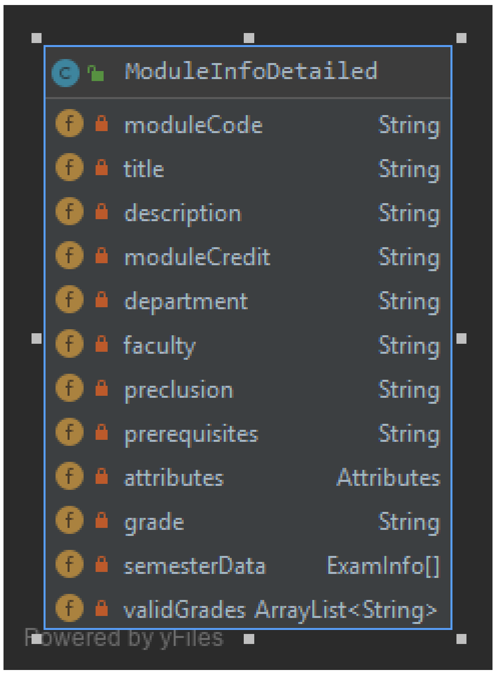
The above function is done in the `CliLauncher` class, during the setup function call.

Since `Gson` is able to internally read a JSON file and when given the same template within a Java object class, it would be able to map the value fields to each of the same keys in the Java Object.

This allows the application to have a direct access to all the modules which are currently offered in NUS, by reading the returned JSON string and parsing it directly into a list of Module information. Since the data had to be modelled, the following ModuleInfoDetailed and ModuleInfoSummary was created to capture the data in the JSON file to be used during runtime.



Since some modules may not contain data for every field, each of the module classes above are required to have default values on initialisation so as to prevent `NullPointerExceptions` during runtime when such module data is accessed.



ModuleInfoDetailed

moduleCode	String
title	String
description	String
moduleCredit	String
department	String
faculty	String
preclusion	String
prerequisites	String
attributes	Attributes
grade	String
semesterData	ExamInfo[]
validGrades	ArrayList<String>

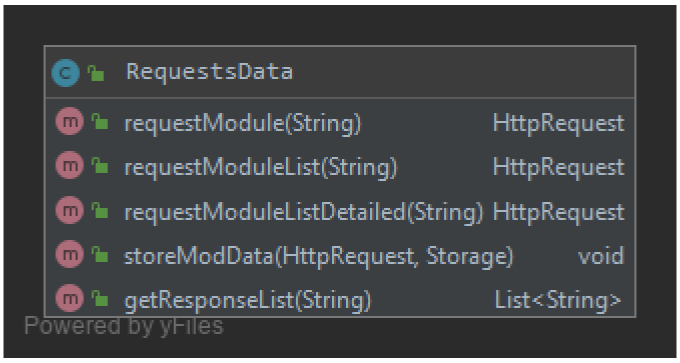
This also extends to choosing the right data type for modelling our module information, since certain fields maybe malformed and thus our implementation of the fields data type in module information classes would mostly contain `strings`, unless it is certain that the data type found in the returned JSON string is strictly `boolean` or `double`.

To allow for quick access to the module information classes, after parsing module information into a list of `ModuleInfoDetailed`, it would then be converted into a `HashMap`, where the key-value pair is the string containing the module code, and the value is the `ModuleInfoDetailed`.

This is `HashMap` is exposed to all the command classes during runtime, and this is done automatically on startup so that the module data is accessible directly to the user. This functionality is handled by `JsonWrapper` since it involves parsing JSON files into direct Java Objects.

4.8.2. RequestData

Internally, this class is responsible for requesting data from the `NUSMODS API` and thus uses Java's Native `HTTPRequest` Library.



RequestsData
requestModule(String) HttpRequest
requestModuleList(String) HttpRequest
requestModuleListDetailed(String) HttpRequest
storeModData(HttpRequest, Storage) void
getResponseList(String) List<String>
Powered by yFiles

The current implementation is fixed to only request data for the current academic year, but this is subject to change in the coming versions to allow for users to choose to update the data once it gets outdated.

A better implementation might be considered so that the class itself can be more customized for `ModPlan`.

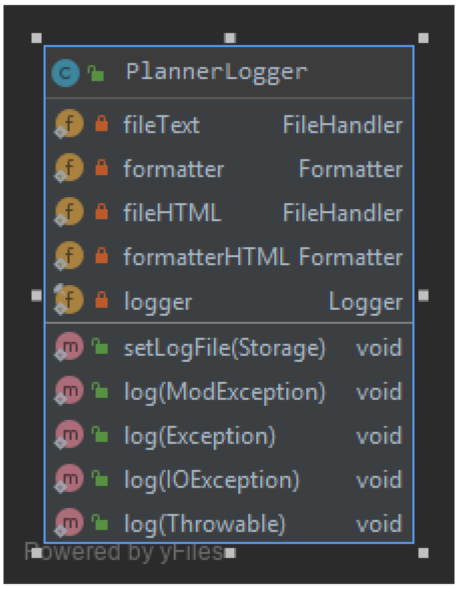
The remove further dependency on the Internet for making API calls to NUSMODS, in future this implementation would be modified to use provided the data in the JAR resources.

For the proposed implementation of updating user data, there are a few alternatives which could be considered:

- Alternative 1: There would be a prompt for the user on startup to check if they wish to update their module data. If the user decides to do so, they enter `yes` and it would be updated.
  - Since this may be distracting for users on every startup, once they user entered `no`, it would no longer prompt for the data update.
  - This implementation requires our application to remember user settings preferences which can be added as an additional feature.
- Alternative 2: Automatic prompting, where the initial startup date is recorded, and would be mapped to a particular semester.
  - Once the semester has been completed, it would prompt the user to update the data. Since this requires and internet connection, this prompt is necessary.
- Alternative 3 (Selected): The base data is packaged into the resources package, thus the `JAR` would be able to generate the HashMap of ModuleInfoDetailed directly without needed to query from NUSMODS API.
  - The existing implementation to connect to NUSMODS to obtain the module data would be converted to an Update command, giving the user flexibility to choose when to update their module data.
  - After the user ran their first update command, it ModPlan will no longer read module data from the internal resource file, but will read the data downloaded locally by the `update module` command.

4.9. Logger

The follow dependency diagram shows the relationship Logger has with the rest of the other classes.



`LoggerFormatter` formats the logging standard which all logging entries uses. `PlannerLogger` is called on every exception handling errors which occurs in `Parser`, `JsonWrapper` and the main `CliLauncher` class.

It is present to capture all `ModException` errors as `WARNING`, while any other unhandled exceptions would be deemed as `SEVERE`.

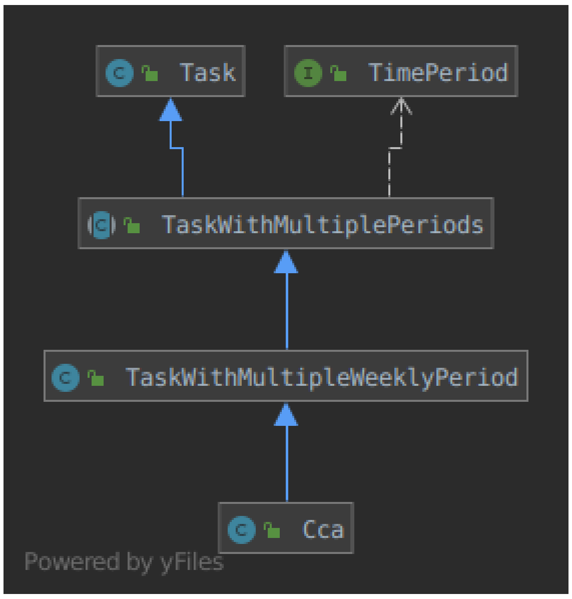
The logging data is written to the `logging.log` file found in the data folder when the application is started.

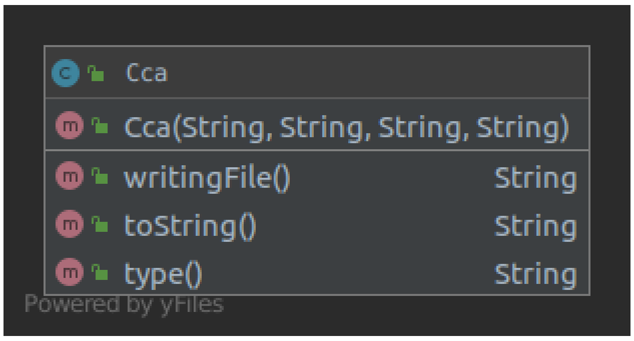
4.10. Cca

`Cca` is a class designed for managing user-defined CCAs. All instances of `Cca` are stored in a `List`-like structure `TaskList<Cca>` which inherits from `ArrayList<TaskWithMultipleWeeklyPeriod>`.

4.10.1. Current implementation

These diagrams illustrate the `Cca` class:





`Cca` is a child class of the legacy class `TaskWithMultipleWeeklyPeriod` which belongs to our implementation of `Duke`. However, since `TaskWithMultipleWeeklyPeriod` is a legacy class, we will update on its implementation after setting up other features and documentations (preferably after ModPlan v2.0 release).

5. Coming in v2.0

5.1. Password management

5.1.1. Add or update an existing password: `passwd`

Add or update an existing password.
If a password has been previously set (updating password mode), a prompt will show up asking for old password before a new one can setup. In that case, kindly enter your current password as prompted to setup a new one.
Format:
`passwd myPassword`

5.1.2. Implementation

Password management will be implemented in a new `SetPasswordCommand` class for password adding/updating in complementary with `ClearCommand` for password removing.

5.2. GUI

ModPlan v2.0 will support GUI (Graphical User Interface). We have not decided whether the CLI (Command Line Interface) will be fully replaced by GUI or will be an additional feature. However, the new GUI will hopefully improve ModPlan’s user-friendliness and user’s productivity.

6. Documentation

We utilise AsciiDoc as the default tool to write our documentation, as the layout is more flexible and aesthetically pleasing to view compared to other alternatives such as MarkDown. Additionally our preferred IDE IntelliJ has AsciiDoc support, and is thus easier for us to code and preview changes in.

We also use PlantUML and draw.io to help use make our diagrams used in this Developer’s Guide.

These software enable us to create different types of diagrams easily, and for free.

7. Testing

7.1. Overview

We used JUnit to code our test cases, and Jacoco to monitor the test coverage of our project.

There are two ways to run tests.

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 'specificTest'`

Method 2: Using Gradle

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

7.2. Creating Tests

Create new test classes under the `/main/src/test/java/planner` package to increase the coverage of the tests for the project.

- We have a sample testing framework put in place such as `CommandTest` to aid in testing implementation.
- Remember to utilise the JUnit assertions in your testing, as well as ensure testing works before pushing the repo.

8. Dev Ops

8.1. Build Automation

We use Gradle to automate our build process, as well as to create our `JAR` release.

8.2. Continuous Integration

We use Travis CI to perform continuous integration on our project through GitHub.

8.3. Code Coverage Reporting

We use Jacoco to check the code coverage of our project.

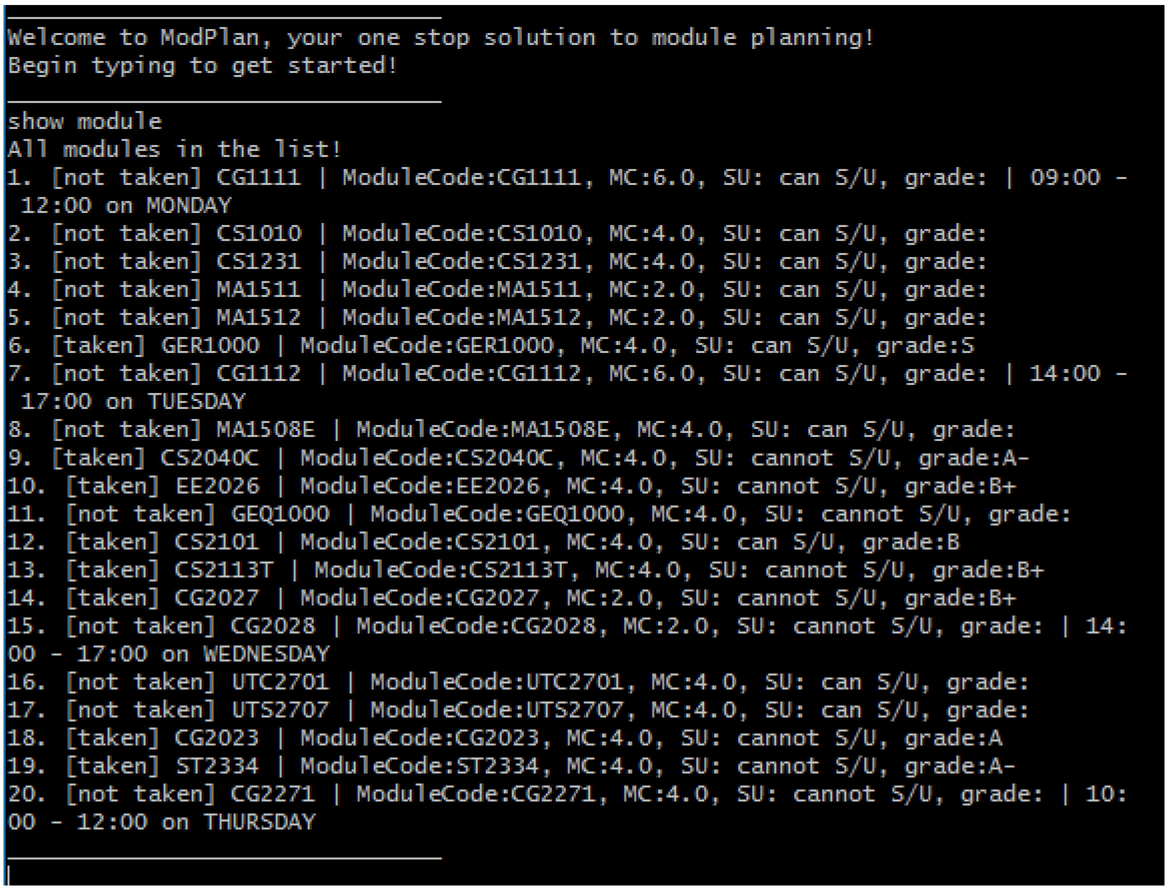
8.4. Making a Release

Here are the steps to create a new release.

- Update the version number in `ModPlan.java`.
- Generate a JAR file using gradle.
- Tag the repo with the version number. e.g. `v0.1`
- Create a new release using GitHub and upload the JAR file you created.

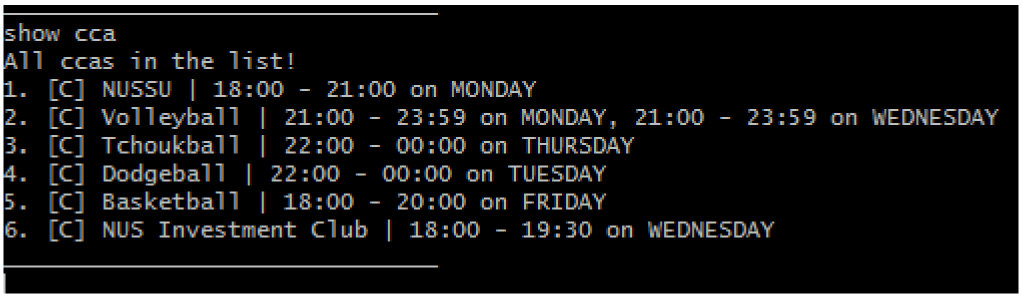
9. Manual User Testing

The following should be obtained after running `show module`:



```
________________________________________________________________
Welcome to ModPlan, your one stop solution to module planning!
Begin typing to get started!
________________________________________________________________
show module
All modules in the list!
1. [not taken] CG1111 | ModuleCode:CG1111, MC:6.0, SU: can S/U, grade: | 09:00 -
 12:00 on MONDAY
2. [not taken] CS1010 | ModuleCode:CS1010, MC:4.0, SU: can S/U, grade:
3. [not taken] CS1231 | ModuleCode:CS1231, MC:4.0, SU: can S/U, grade:
4. [not taken] MA1511 | ModuleCode:MA1511, MC:2.0, SU: can S/U, grade:
5. [not taken] MA1512 | ModuleCode:MA1512, MC:2.0, SU: can S/U, grade:
6. [taken] GER1000 | ModuleCode:GER1000, MC:4.0, SU: can S/U, grade:S
7. [not taken] CG1112 | ModuleCode:CG1112, MC:6.0, SU: can S/U, grade: | 14:00 -
 17:00 on TUESDAY
8. [not taken] MA1508E | ModuleCode:MA1508E, MC:4.0, SU: can S/U, grade:
9. [taken] CS2040C | ModuleCode:CS2040C, MC:4.0, SU: cannot S/U, grade:A-
10. [taken] EE2026 | ModuleCode:EE2026, MC:4.0, SU: cannot S/U, grade:B+
11. [not taken] GEQ1000 | ModuleCode:GEQ1000, MC:4.0, SU: cannot S/U, grade:
12. [taken] CS2101 | ModuleCode:CS2101, MC:4.0, SU: can S/U, grade:B
13. [taken] CS2113T | ModuleCode:CS2113T, MC:4.0, SU: cannot S/U, grade:B+
14. [taken] CG2027 | ModuleCode:CG2027, MC:2.0, SU: cannot S/U, grade:B+
15. [not taken] CG2028 | ModuleCode:CG2028, MC:2.0, SU: cannot S/U, grade: | 14:
00 - 17:00 on WEDNESDAY
16. [not taken] UTC2701 | ModuleCode:UTC2701, MC:4.0, SU: can S/U, grade:
17. [not taken] UTS2707 | ModuleCode:UTS2707, MC:4.0, SU: can S/U, grade:
18. [taken] CG2023 | ModuleCode:CG2023, MC:4.0, SU: cannot S/U, grade:A
19. [taken] ST2334 | ModuleCode:ST2334, MC:4.0, SU: cannot S/U, grade:A-
20. [not taken] CG2271 | ModuleCode:CG2271, MC:4.0, SU: cannot S/U, grade: | 10:
00 - 12:00 on THURSDAY
________________________________________________________________
```

The following should be obtained after running `show cca`:



```
________________________________________________________________
show cca
All ccas in the list!
1. [C] NUSSU | 18:00 - 21:00 on MONDAY
2. [C] Volleyball | 21:00 - 23:59 on MONDAY, 21:00 - 23:59 on WEDNESDAY
3. [C] Tchoukball | 22:00 - 00:00 on THURSDAY
4. [C] Dodgeball | 22:00 - 00:00 on TUESDAY
5. [C] Basketball | 18:00 - 20:00 on FRIDAY
6. [C] NUS Investment Club | 18:00 - 19:30 on WEDNESDAY
________________________________________________________________
```

Testing the `update module` command would work only if there is stable internet connection and that the NUSMODS Server API is operational.

Appendix A: Product Scope

A better module planner

We aim to fulfill a need that is currently lacking in module planning, which in this case the is ability to plan ahead for more semesters up until graduation. Additional features would likely include the ability to generate a projection report for CAP computation and CCA planning.

Target Users: NUS CEG Students
* have a need for additional module planning aside from utilising NUSMODS.
* comfortable with using CLI
* can type fast
* prefers desktop applications
* has a computer with Internet access available

Value proposition:
* Provides many more features that NUSMODS currently does not have, which we have deemed to still be important to target users.

Appendix B: User Stories

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

Priority	As a …	I can …	So that …
* * *	NUS CEG Student	Search for a module’s workload	Balance my workload for the current semester
* * *	NUS CEG Student	Monitor my total workload from my modules	Track my total workload for the current semester
* * *	NUS Student	See my daily timetable	Keep a schedule of what classes and extra-curricular activities I have
* * *	NUS CEG Student	Check if I have completed the required prerequisite modules	Plan ahead for what modules to take

Priority	As a …	I can …	So that …
* * *	Forgetful NUS CEG Student	Add up my total number of MCs taken	Track my progress towards graduation
* * *	NUS CEG Student	View the core modules required for graduation	Know what are the modules I still need to take to graduate
* *	NUS Student	Add CCAs to my class timetable	Take CCAs that do not clash with my lessons
* *	NUS Student	Create a custom module for my CCAs	Personalise the timing and location of my CCA in my timetable
* *	NUS Undergraduate Student	Know requirements for a Master’s/PHD at NUS	Plan my course of action if I wish to apply for post-graduate studies
* *	NUS CEG Student	Easy access to my recommended study schedule	Know what modules I should prioritise bidding for
* *	NUS CEG Student	Plan to take modules ahead of the current semester	Alter my holiday/graduation plans as required
* *	NUS CEG Student	Know what GE modules I have not completed	Plan to take GE modules over a few semesters
* *	NUS CEG Student	Know what UE modules I have completed	Plan to take UE modules over a few semesters

Priority	As a …	I can …	So that …
* *	NUS CEG Student	View the total number of Level-1000 modules taken	Check if I have exceeded the 60MC limit for Level-1000 modules
* *	NUS CEG Student	Know if the module has S/U options	Plan ahead for my S/U usage
* *	NUS CEG Student	Project my future CAP based on my expected and past grades	See how hard I must work to hit my target CAP
*	NUS CEG Student	Download my timetable as a photo	View it on other mediums such as my mobile phone
*	NUS Student	Know the directions to my classes	Plan my route accordingly
*	NUS Student	Know my priority score when bidding for a module	Plan my module bidding appropriately
*	NUS Student	See a list of my course’s modules available in SEP/NOC	Plan what modules to take should I go for SEP/NOC
*	Exchange Student	Know if a module can be mapped to my home university	Plan what modules to take in NUS

(more to be added as necessary)

Appendix C: Use Cases

C.1. Use Case C01: Adding modules to user’s timetable

Actor: NUS CEG Student

MSS

1. User inputs the module code
2. ModPlan shows the module information to the user, such as description, number of MCs, prerequisite modules etc. and requests confirmation from the user to add this module
3. User confirms they want to add the module
4. ModPlan shows the non-clashing available timings of the module to the user
5. User confirms which class timing they wish to add to their timetable
6. ModPlan adds that specific class to the user’s timetable, and prints the user’s updated timetable
   Use case ends.

Extensions

2a1. If the module is a Level-1000 module, ModPlan checks for the user’s current number of Level-1000 modules taken
2a2. If the limit is not exceeded, proceed to step 3
2a3. If the limit will be exceeded, warn the user, and prevent addition of the module
2a4. Additionally, if the prerequisites of the module have not been fulfilled, prevent addition of the module, and inform user of the modules needed to be taken
Return to step 3.

C.2. Use Case C02: Viewing graduation requirements

Actor: NUS CEG Undergraduate Student

MSS

1. User inputs their course name
2. ModPlan shows the courses that match the user’s input
3. User selects the correct course they wish to check graduation requirements for
4. ModPlan displays all the modules required for graduation, and lists the number of MCs required for graduation
   Use case ends.

Extensions

3a. User can input the modules they have taken already that count towards graduating that course
3b. ModPlan will exclude these modules from the list and MC count
Return to Step 4.

C.3. Use Case C03: Viewing class and CCA timetable

Actor: NUS Student

MSS

1. User inputs the command to view timetable
2. ModPlan shows the user their current timetable, including class and CCA timings

C.4. Use Case C04: Viewing of modules taken in past semester

Actor: NUS Student

MSS

1. User inputs the command to view past modules
2. ModPlan shows the user a list of all modules taken, and those they are currently taking.
   Use case ends.

C.5. Use Case C05: Generation of current CAP based on past modules

Actor: NUS Student

MSS

1. User inputs the command to generate CAP report
2. ModPlan shows the user modules they had taken, and requests user to input their grades obtained
3. User inputs the modules they have taken, as well as the respective grades obtained
4. After inputting the grades, ModPlan calculates and shows the user their current MCs accumulated and CAP.
   Use case ends.

Extensions

4a. User can then input a future module they plan to take and project their CAP 4b. ModPlan will show the projected CAP using grades the user obtained from the module’s prerequisite classes

C.6. Use Case C06: Check Module information via input search

Actor: NUS Student

MSS

1. User inputs the command to search module information
2. ModPlan shows the user key information regarding the module, if it is SU-able or if it has any prequisites.
   Use case ends.

C.7. Use Case C07: Creating a personalised Module (eg. for CCAs)

Actor: NUS Student

MSS

1. User inputs the command to create custom module
2. ModPlan prompts the user for additional details of the custom module, such as description and times
3. User inputs the description and date/times
4. ModPlan prompts user to confirm addition of custom module to timetable
5. User confirms addition
6. ModPlan adds custom module to timetable, and shows user updated timetable.
   Use case ends.

Extensions

5a. User can cancel addition
5b. ModPlan will cancel addition of custom module, and delete information inputted

C.8. Use Case C08: Checking number of MCs taken

Actor: NUS Student

MSS

1. User inputs the command to check MC
2. ModPlan will show the total MCs taken up to this point.
   Use case ends.

Extensions

1a. User can specify additional parameters to check MCs completed for specific periods
eg. `check MC 1-1` will check for MCs taken in Year 1 Semester 1

(Use case will be tackled in v2.0 with multiple semester support)

Appendix D: Non-Functional Requirements

1. ModPlan should run on any machine with JDK `11` and above installed.
2. ModPlan should be fast to view and input commands.
3. ModPlan should require as few steps as possible for the user to do what they want to do.
4. ModPlan should store data between sessions so the user does not have to input all their information again.
5. ModPlan should scrape data from NUSMODS API at least once a semester to keep up to date with any changes in modules.

Appendix E: Manual User Testing

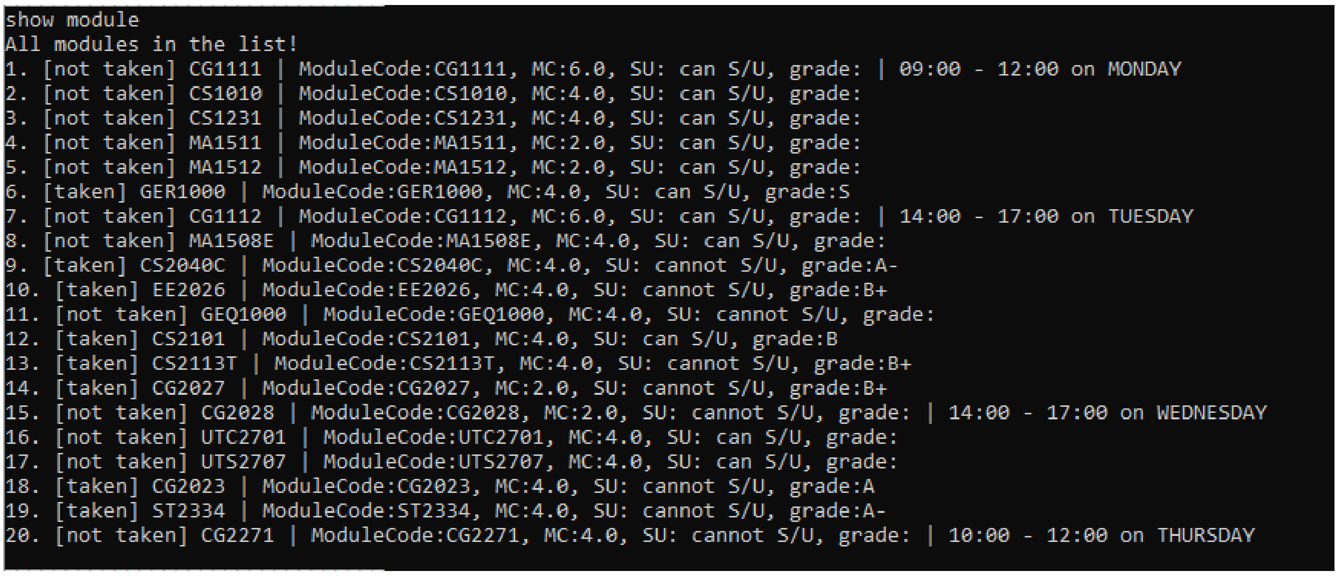
We have included a base data file populated with ~20 modules and ~5 CCAs for manual testing. The program will load from these data files upon the first startup if a save file is not found.

E.1. Loading Manual Test Data

E.1.1. Test Data

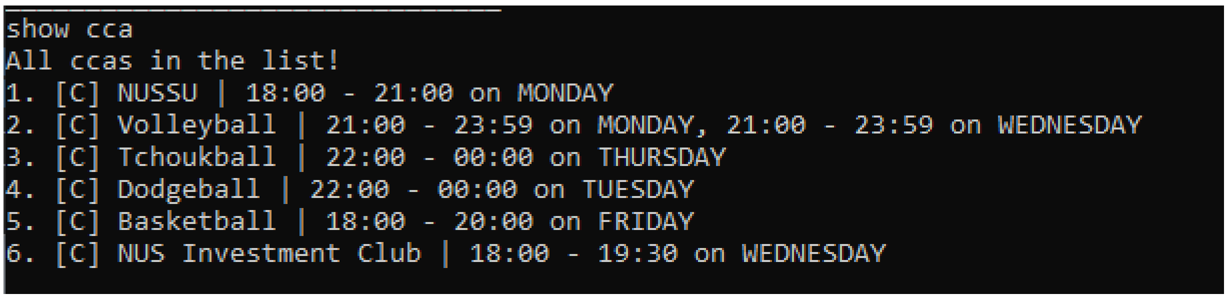
To demonstrate the functionality of the product, the `JAR` release contains sample data so that the user can easily test and observe that the application runs as intended.

Type `show module` and observe that the following output is obtained:



```
show module
All modules in the list!
1. [not taken] CG1111 | ModuleCode:CG1111, MC:6.0, SU: can S/U, grade: | 09:00 - 12:00 on MONDAY
2. [not taken] CS1010 | ModuleCode:CS1010, MC:4.0, SU: can S/U, grade:
3. [not taken] CS1231 | ModuleCode:CS1231, MC:4.0, SU: can S/U, grade:
4. [not taken] MA1511 | ModuleCode:MA1511, MC:2.0, SU: can S/U, grade:
5. [not taken] MA1512 | ModuleCode:MA1512, MC:2.0, SU: can S/U, grade:
6. [taken] GER1000 | ModuleCode:GER1000, MC:4.0, SU: can S/U, grade:S
7. [not taken] CG1112 | ModuleCode:CG1112, MC:6.0, SU: can S/U, grade: | 14:00 - 17:00 on TUESDAY
8. [not taken] MA1508E | ModuleCode:MA1508E, MC:4.0, SU: can S/U, grade:
9. [taken] CS2040C | ModuleCode:CS2040C, MC:4.0, SU: cannot S/U, grade:A-
10. [taken] EE2026 | ModuleCode:EE2026, MC:4.0, SU: cannot S/U, grade:B+
11. [not taken] GEQ1000 | ModuleCode:GEQ1000, MC:4.0, SU: cannot S/U, grade:
12. [taken] CS2101 | ModuleCode:CS2101, MC:4.0, SU: can S/U, grade:B
13. [taken] CS2113T | ModuleCode:CS2113T, MC:4.0, SU: cannot S/U, grade:B+
14. [taken] CG2027 | ModuleCode:CG2027, MC:2.0, SU: cannot S/U, grade:B+
15. [not taken] CG2028 | ModuleCode:CG2028, MC:2.0, SU: cannot S/U, grade: | 14:00 - 17:00 on WEDNESDAY
16. [not taken] UTC2701 | ModuleCode:UTC2701, MC:4.0, SU: can S/U, grade:
17. [not taken] UTS2707 | ModuleCode:UTS2707, MC:4.0, SU: can S/U, grade:
18. [taken] CG2023 | ModuleCode:CG2023, MC:4.0, SU: cannot S/U, grade:A
19. [taken] ST2334 | ModuleCode:ST2334, MC:4.0, SU: cannot S/U, grade:A-
20. [not taken] CG2271 | ModuleCode:CG2271, MC:4.0, SU: cannot S/U, grade: | 10:00 - 12:00 on THURSDAY
```

The following should be obtained after running `show cca`:



```
show cca
All ccas in the list!
1. [C] NUSSU | 18:00 - 21:00 on MONDAY
2. [C] Volleyball | 21:00 - 23:59 on MONDAY, 21:00 - 23:59 on WEDNESDAY
3. [C] Tchoukball | 22:00 - 00:00 on THURSDAY
4. [C] Dodgeball | 22:00 - 00:00 on TUESDAY
5. [C] Basketball | 18:00 - 20:00 on FRIDAY
6. [C] NUS Investment Club | 18:00 - 19:30 on WEDNESDAY
```

Testing the `update module` command would work only if there is stable internet connection and that the NUSMODS Server API is operational.

E.2. Manual Testing

To test our product manually, kindly open the `[CS2113T-F10-1][ModPlan].jar` file from the `v1.4` release using the command prompt of your preferred OS, and test using commands featured in our User Guide.

Appendix F: Glossary

- **API** : Application Programming Interface
- **CEG** : Computer Engineering
- **NUSMODS** : NUSMODS is an external library where consolidated module data from NUS is collected
- **JSON** : JavaScript Object Notation