Le Hong Long - Project Portfolio

PROJECT: ModPlan v1.3

Overview

ModPlan is a command line interface (CLI) based module planning system that expands on NUSMODS to help NUS Computer Engineering students to plan their modules and CCAs for all four years of their bachelor's degree. It is written in Java and utilizes NUSMODS API to extract official module data from NUS Registrar's Office. The project is maintained under this GitHub repository.

This document details my contributions to the project.

Summary of contributions

This section describes my contributions to ModPlan's features and code base.

- **Feature**: added **support for CCA management**
  - What it does: CCA management allows the user to add, schedule CCAs, and check for CCA/module schedule clashing
  - Justification: NUSMODS does not provide such capabilities as it is limited to module scheduling only. With this additional CCA scheduling ability, it would be much more convenient for students to organize their timetables.
  - Highlights: Implementing CCA requires an additional multiple weekly time periods task implementation as Java does not directly provide such a feature. Also, the CCAs must not clash with each others and each CCA can have multiple time slots.
- **Feature**: added **user profile with encryption and decryption**
  - What it does: Enables user profile management, encrypts and decrypts user data directly from disk and performs integrity check, detecting any tampering to the user data.
  - Justification: Since ModPlan is a personal scheduler and may have access to user's "sensitive" information (if provided) such as grades and detailed daily task scheduling, it is important that the data is kept as safe as possible without being easily modified.
  - Highlights: Supports multi-layer encryption with mixed algorithms and decryption can be configured to require a password set by user on program startup. Any modification to user's stored data will be detected.
- **Feature**: added **Parser for user input interpretation with helps and suggestions**
  - What it does: Parser interprets and checks for invalid inputs. If user is unfamiliar with ModPlan, parser will display helps and suggest potential commands. It also makes call to the corresponding command to fulfill user's request.

    - Justification: This feature is the backbone for input-command mapping, essentially plays a key role in the "brain" of ModPlan. Helps and command suggestions are also very useful for new user to familiarize with ModPlan.
    - Highlights: Parser affects all command calling in ModPlan. It is **not** if-else based, thus highly scalable, and very robust due to the backend support of the powerful `argparse4j` library.
- **Feature** (**code merged** but **temporarily disabled**): added **password management**
    - What it does: Allows user to setup/change/clear password for credential managing purposes.
    - Justification: This feature is complementary with the user data encryption, further boosting the security of ModPlan.
    - Highlights: One-way password hashing so that original password cannot be retrieved, encrypted with user data for additional protection.
    - Special note: Temporarily disabled for ease of peer testing in PE as forgetting password is troublesome due to ModPlan's strict security, will be reactivated in ModPlan v2.0.
- **Feature** (coming in v2.0): **GUI**
    - What it does: Allows user to interact with ModPlan using a graphical interface instead of a command line interface.
    - Justification: This feature significantly enhances ModPlan's visual attractiveness and clarity.
    - Highlights: Highly responsive due to intelligent thread management.
    - Special note: Available in ModPlan v2.0 onward.
- **Code contributed**:
    - Project Dashboard
    - Functional code
        - Package Cca, Task and Class TaskList for CCA management.
        - Package credential and Class ClearCommand for user profile management with direct encryption and decryption.
        - Package Parser for command parser.
        - Class SetPasswordCommand and ClearCommand for password management.
        - Package gui and Class GuiLauncher for GUI.
    - Test code
        - Package credential
        - Class ClearTest (ClearCommand test)
        - Package Parser
- **Other contributions**:
    - Project management:
        - Managed project branch protection rules
    - Enhancements to existing features:
        - Unified and integrated commands with argparse4j (Pull request #100)

- Documentation:
  - Did cosmetic tweaks to existing contents of User Guide
  - Wrote Parser and CCA implementation sections for Developer Guide (Pull requests #114, #120)
- Community:
  - PRs reviewed (with non-trivial review comments): #128, #117, #80, #53
  - Contributed to forum discussions (examples: #213, #107, #105, #80, #73)
  - Reported bugs and suggestions for other teams in the class (examples: #218, #111, #74)
- Tools:
  - Integrated a third party library (`argparse4j`) to the project (Pull request #89)

Contributions to the User Guide

This section details my contributions to the User Guide. They showcase my ability to write documentation targeting end-users. I wrote the Parser Errors section and made cosmetic changes across the whole document. Introduced below is the Parser Errors section

Parser Errors

If you encountered an error message starting with `ModPlanner: error:`, then this section is for you!

There are 4 common types of Parser Errors:

`ModPlanner: error: invalid choice …`

This error appears when you input an invalid command or argument to ModPlanner. However, the error message will display the valid options for you. In some cases, ModPlanner may even suggest a possible command that it thinks you intended to write!

Example of input that can cause this error: `clean`
Example error message:

```
clean
usage: ModPlanner [-h]
                  {add,cap,reminder,scheduleCca,grade,show,clear,update,sort,remove,bye}
                  ...
ModPlanner: error: invalid choice: 'clean' (choose from 'add', 'cap',
'reminder', 'scheduleCca', 'grade', 'show', 'clear', 'update', 'sort',
'remove', 'bye')

Did you mean:
    clear
```

> ***Solving the error:***
> Select one from the provided legal options. ModPlanner even noticed that you probably meant `clear` which is a valid command, and suggested it.

`ModPlanner: error: too few arguments`

This error appears when you do not supply enough arguments for a specific command.

Example of input that can cause this error: `add module`
Example error message:

```
add module
usage: ModPlanner add module [-h] moduleCode
ModPlanner: error: too few arguments
```

> ***Solving the error:***
> Look for the missing arguments as provided in the error message. In this case, it is `moduleCode`. If you are unsure what to input for `moduleCode`, try `add module -h`.

```
add module -h
usage: ModPlanner add module [-h] moduleCode

positional arguments:
  moduleCode            Codename of module to add

named arguments:
  -h, --help            show this help message and exit
```

`ModPlanner: error: unrecognized arguments: …`

This error appears when the name of a named argument is specified incorrectly.

Example of input that can cause this error: `add cca test cca --beginTime 15:00 --end 5pm --dayOfWeek MONDAY`
Example error message:

```
add cca test cca --beginTime 15:00 --endTime 5pm --dayOfWeek MONDAY
usage: ModPlanner add cca [-h] --begin BEGIN [BEGIN ...]
                  --end END [END ...] --dayOfWeek DAYOFWEEK name [name ...]
ModPlanner: error: unrecognized arguments: '--beginTime'
```

> ***Solving the error:***
> Look for the correct argument name as provided in the error message! In this case, `--beginTime` should be changed to `--begin`.

`ModPlanner: error: argument index: could not convert …`

Certain arguments should be parsed in the correct format in order for the value to be evaluated correctly. If you encounter this error, chances are you tried to parse a non-integer value to an integer-type argument.

Example of input that can cause this error: `remove cca notANumber`
Example error message:

```
remove cca notANumber
usage: ModPlanner remove [-h] {module,cca} index
ModPlanner: error: argument index: could not convert 'notANumber' to
integer (32 bits)
```

> ***Solving the error:***
> Look for the correct type of the argument from the error message and change your argument to match the type. In this case, `index` should be an `int` but the ModPlanner could not convert the input value `notANumber` to an `int`. An example of a correct command is `remove cca 1` (provided your CCA list is not empty!).

Contributions to the Developer Guide

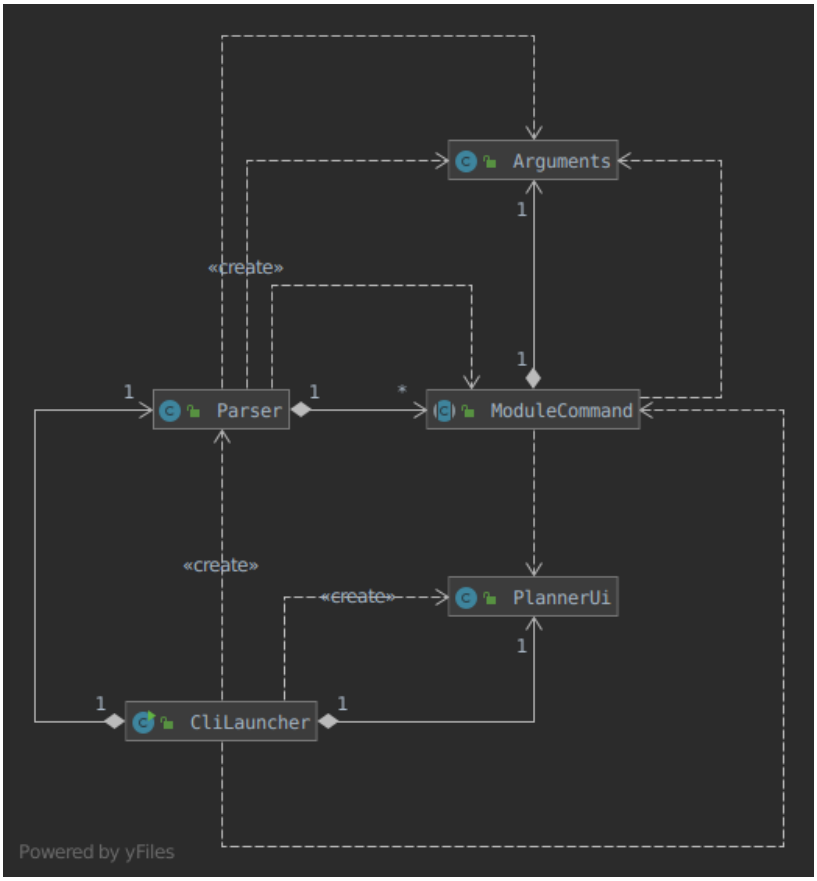
This section details my contributions to the Developer Guide. They showcase my ability to write technical documentation and the technical depth of my contributions to the project. I have written the Cca and Parser implementations and made multiple minor changes across the document. Below is the Parser section

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.

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 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.

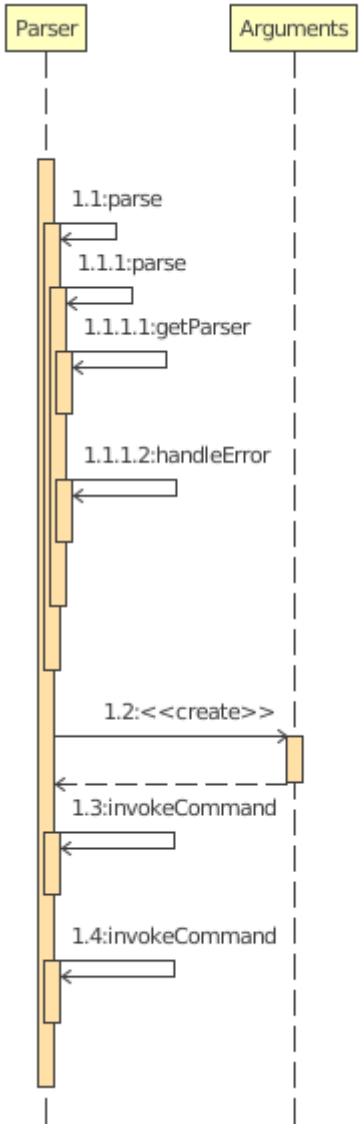
Step 3. `Parser` receives the command, matches it against a suitable `ModuleCommand` class and parses all required arguments. If the command is invalid or missing any required arguments, `Parser` will print the errors to the command line using `PlannerUi` with corresponding helps and command suggestions. 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.

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.