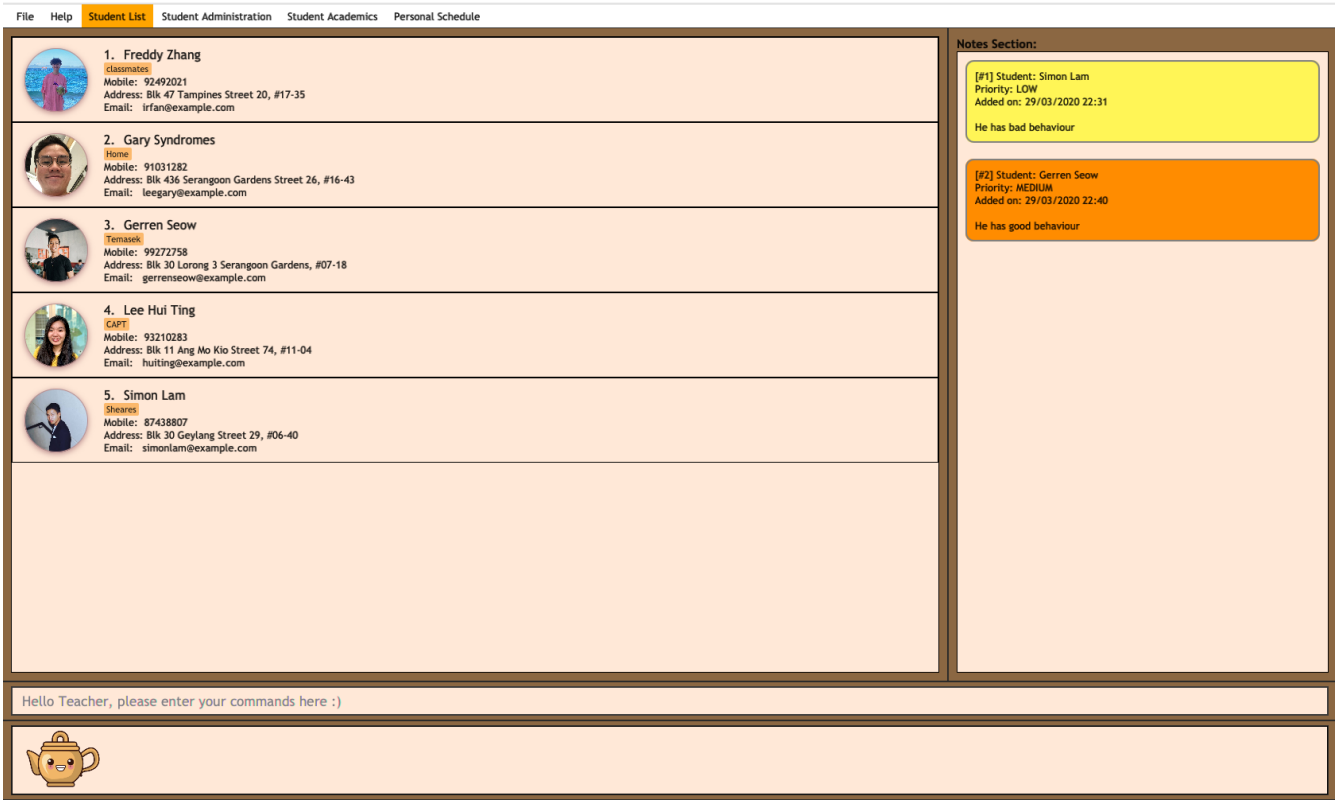
TeaPet - Developer Guide

By: CS2103T-W12-02 Since: Jan 2020 Licence: MIT

1. Introduction

Welcome to TeaPet!



TeaPet is an integrated platform fully customized for Primary School Form Teachers to help you manage your classroom effectively. It contains your personal curriculum schedule, students' particulars, and can also track your class' academic progress.

All your important information is comfortably compartmentalized on our simple and clean Graphical User Interface (GUI) and we are optimized for users who are very comfortable working on the Command Line Interface (CLI).

If you are looking for a way to easily manage your administrative classroom chores and have quick hands, then TeaPet is definitely for you!

2. About

The aim of this Developer Guide is to provide an introduction to the software architecture and implementation of TeaPet. It is made for developers who wish to understand and maintain the backbone of our application.

At TeaPet, we prioritise teamwork. We welcome you developers to contribute and improve on our

application anytime by opening an issue should you have any refreshing ideas to improve TeaPet after reading this guide.

3. How To Use

This section will teach you how should you go about understanding our Developer Guide in order to best understand our product.

Reading ahead, you will find that we've adopted the "top-down" approach into the structure of our Developer Guide — We will first look at the high-level architecture of our application before delving into the implementation details of each feature that makes up TeaPet.

We encourage you to read every page diligently from the top to the bottom in order to get the highest level of understanding of our application. If you are only interested in how we develop one feature, you may read the high-level design then focus on the implementation of that particular feature.

Take note of the following symbols and formatting used in this document:

IMPORTANT	Use this to highlight important stuff
WARNING	Use this for warnings
CAUTION	Use this for caution
NOTE	Use this for note
TIP	This symbol indicates tips.

Table 1. A Summary of symbols used in our User Guide.

`Enter`	This symbol indicates the Enter button on your keyboard.
`command`	A grey highlight indicates that this is a command that can be typed into the command line and executed by the program.

4. Overview of Features

This section will provide you a brief overview of TeaPet's cool features and functionalities.

1. Manage your students easily
    a. Include student's particulars. e.g. address, contact number, next of kin (NOK)
    b. Include administrative details of the students. e.g. attendance, temperature
2. Plan your schedule easily
    a. Create and manage your events with a single calendar

    b. View calendar at a glance
3. Manage your class academic progress easily
    a. Include every student's grades for every examination.
    b. Easy to track progress using helpful tools. e.g. graph plots
4. Add Notes to act as lightweight, digital reminders easily
    a. Include reminders for yourself to help you remember important information.
    b. Search keywords in your notes.
    c. Save the notes as administrative or behavioural
5. Toggle different views to find information easily
    a. Different view modes show only the required information. e.g. detailed, admin, default
6. Data is saved onto your hard disk automatically
    a. Any changes made will be saved onto your computer so you dont have to worry about data being lost.

5. Setting up

This section provides you with the tools needed for you to set up TeaPet.

You can refer to the guide here.

6. Design

6.1. Architecture

This section describes the high-level software architecture of TeaPet.

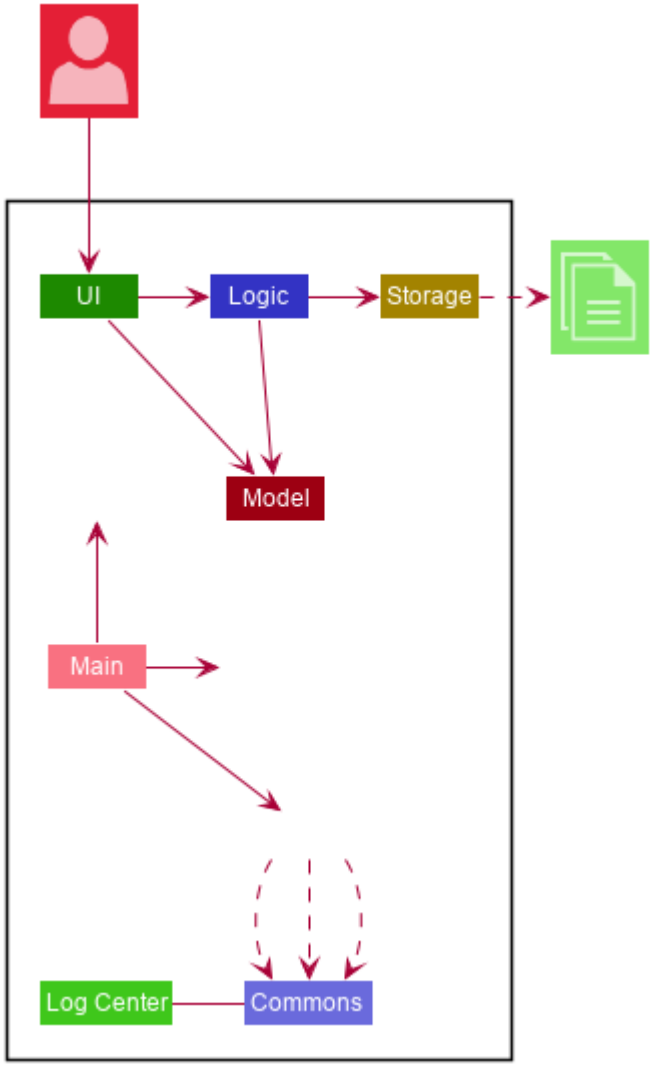


Figure 1. Architecture Diagram

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 four 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.

Each of the 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 it's API in the `Logic.java` interface and exposes its functionality using the `LogicManager.java` class.

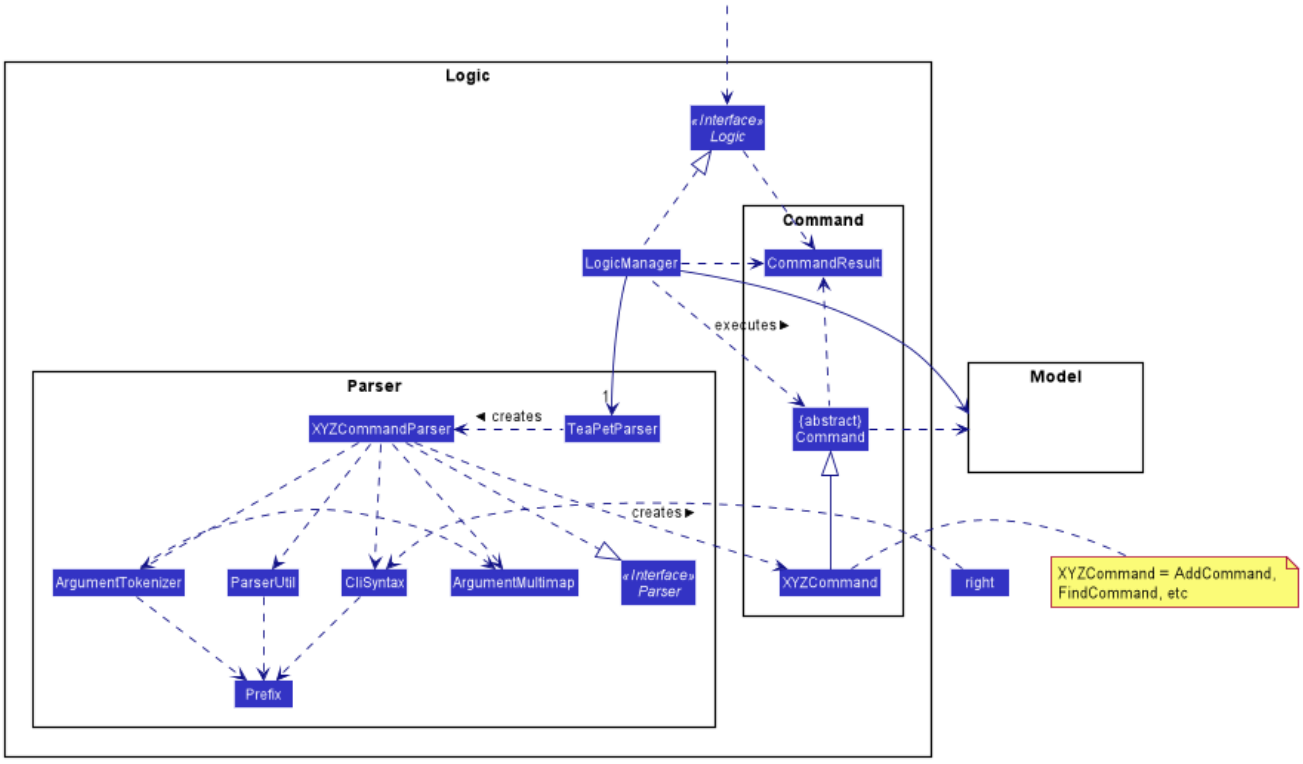


Figure 2. Class Diagram of the Logic Component

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 `{Entity Name} student delete 1`.

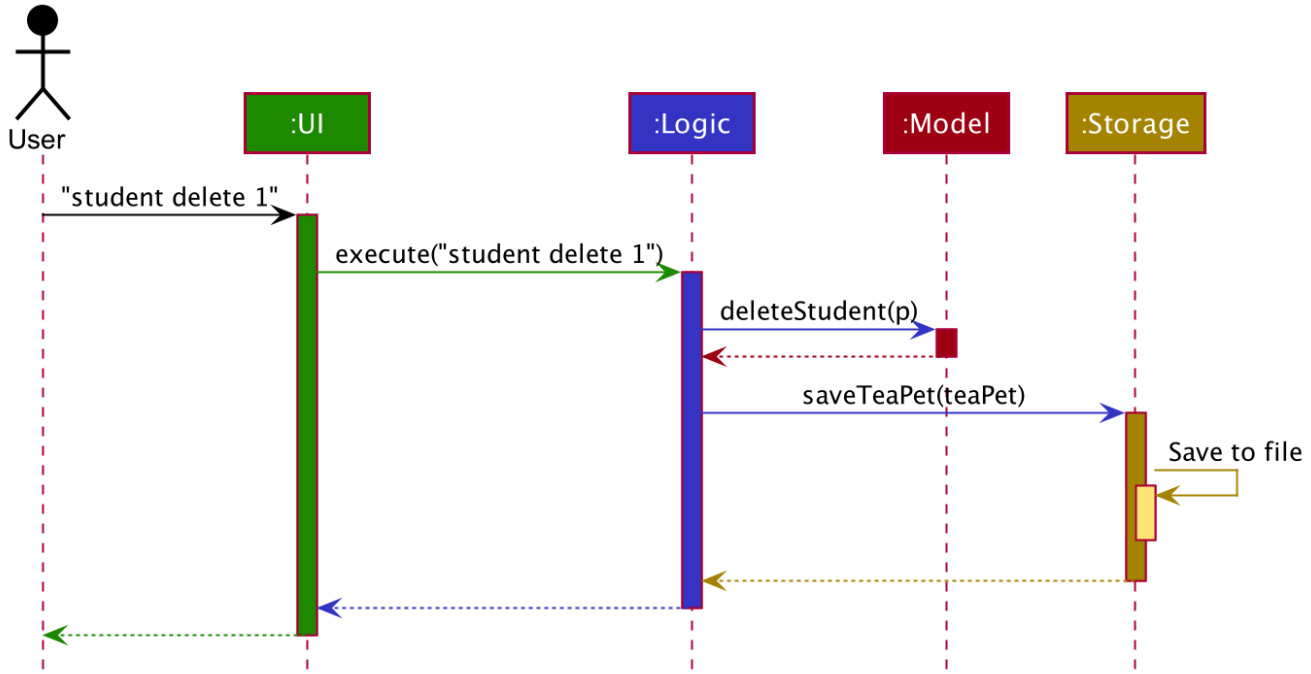


Figure 3. Component interactions for `student delete 1` command

The sections below give more details of each component.

6.2. UI component

This section describes the high-level software structure of TeaPet's UI Component.

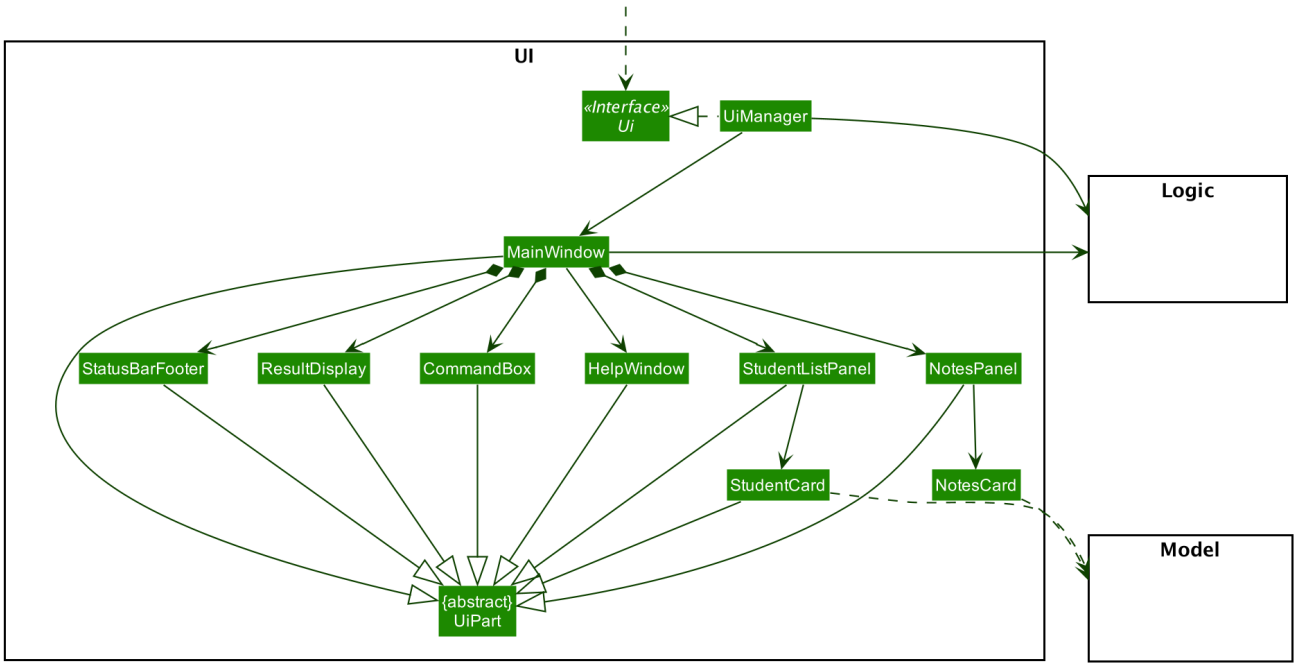


Figure 4. Structure of the UI Component

API : `Ui.java`

The UI consists of a `MainWindow` that is made up of parts e.g.`CommandBox`, `ResultDisplay`, `StudentListPanel`, `NotesPanel`, `StatusBarFooter` and `HelpWindow`. All these, including the `MainWindow`,

inherit from the abstract `UiPart` class.

The `UI` component uses JavaFx UI framework. The layout of these UI parts are defined in matching `.fxml` files that are in the `src/main/resources/view` folder. For example, the layout of the `MainWindow` is specified in `MainWindow.fxml`

The `UI` component,

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

6.3. Logic component

This section describes the high-level software structure of TeaPet's Logic Component.

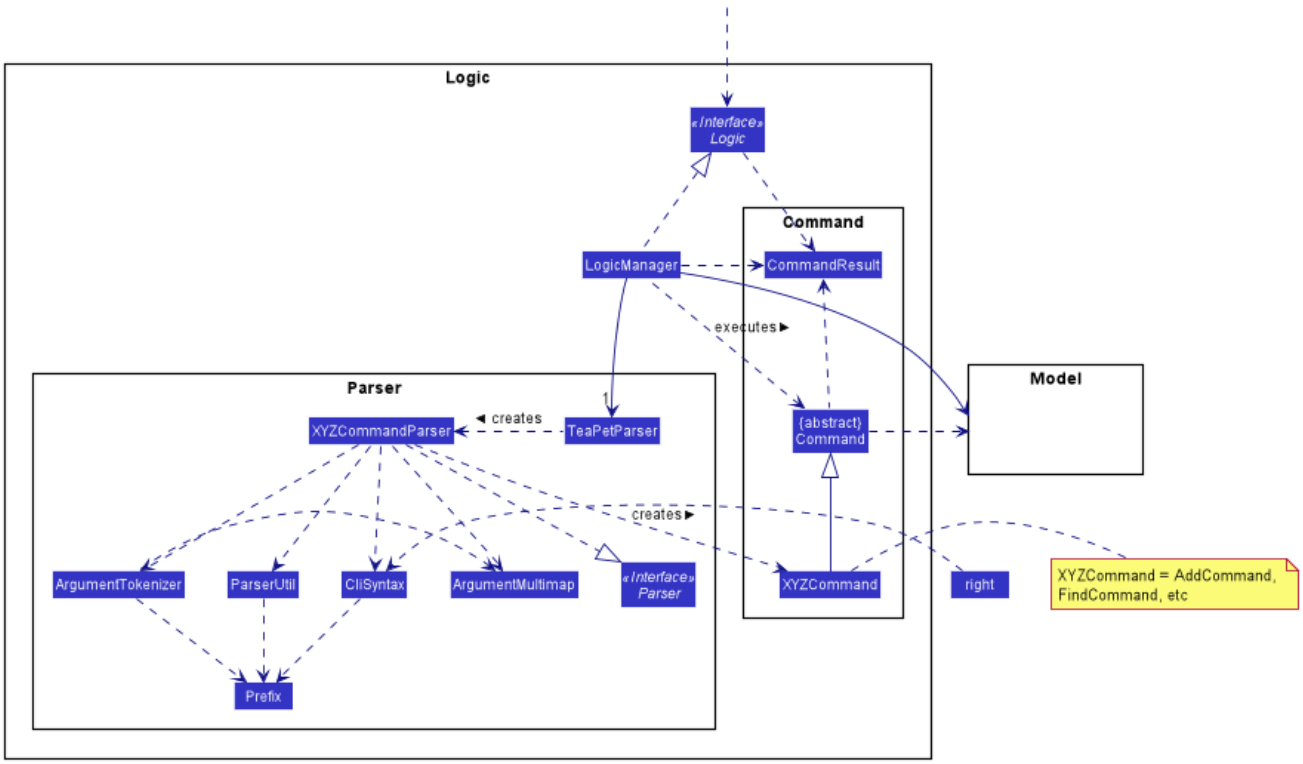


Figure 5. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `TeaPetParser` 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 student).
4. The result of the command execution is encapsulated as a `CommandResult` object which is passed back to the `UI`,and then displayed to the user.
5. In addition, the `CommandResult` object can also instruct the `UI` to perform certain actions, such as displaying help (for commands) and toggling of view between `student default`, `student detailed`.

Given below is the Sequence Diagram for interactions within the `Logic` component for the `execute("student delete 1")` API call.

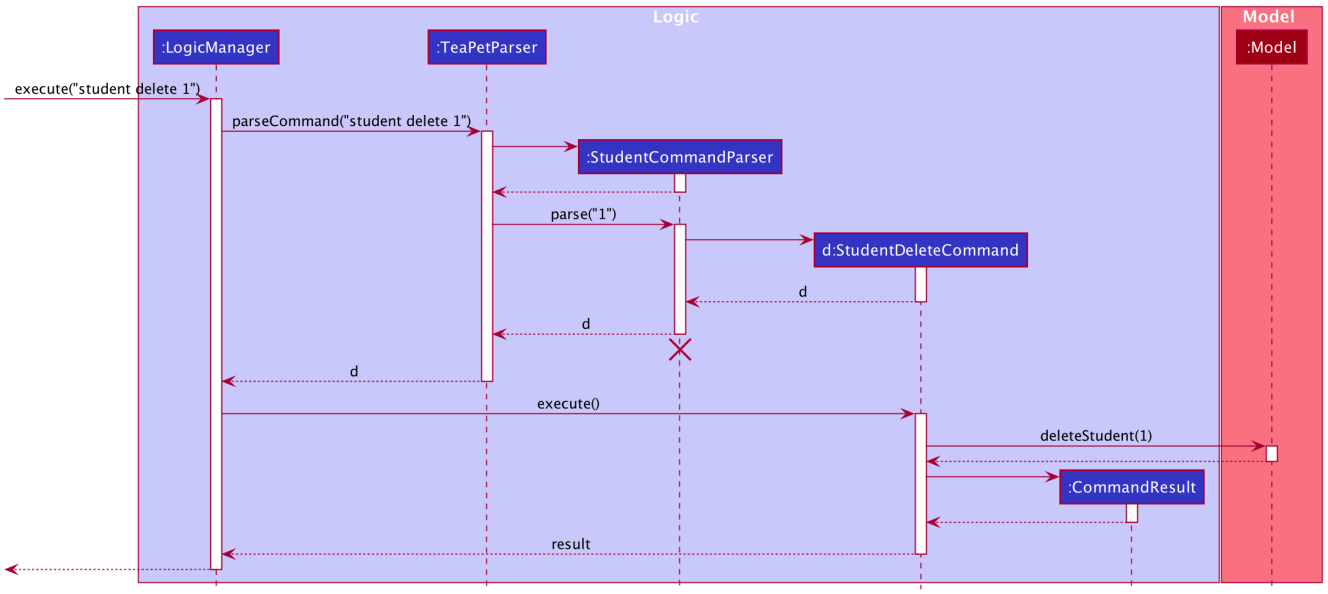


Figure 6. Interactions Inside the Logic Component for the `student delete 1` Command

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

6.4. Model component

This section describes the high-level software structure of TeaPet's Model Component.

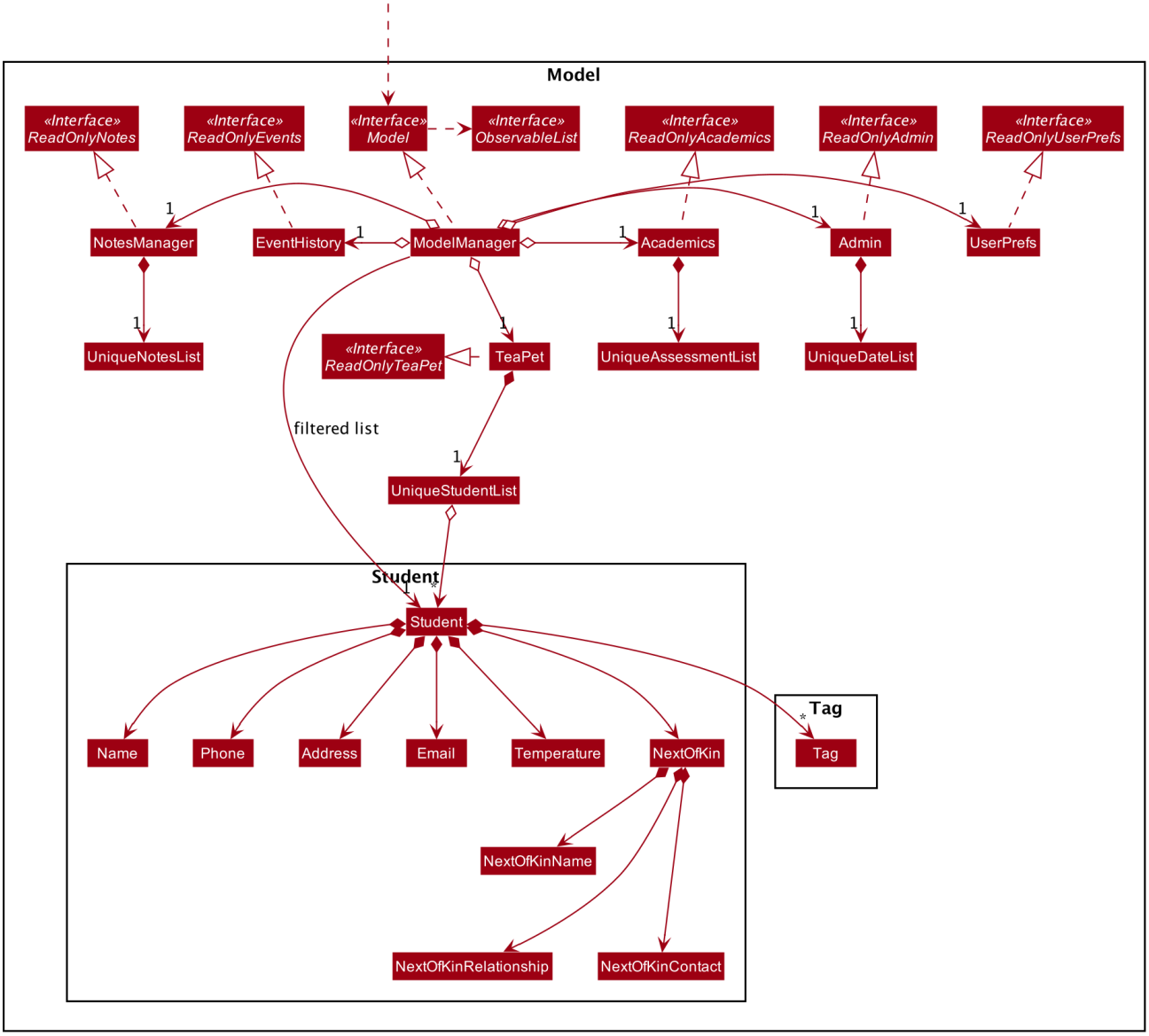


Figure 7. Structure of the Model Component with `Student` class as a detailed example.

API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user's preferences.
- stores the data of different Entities.
- stores in-memory data of Students, Admin, Academics, Notes and Events.
- exposes multiple unmodifiable `ObservableLists` that can be 'observed' e.g. the UI can be bound to these lists so that the UI automatically updates when the data in the lists change.
- does not depend on any of the other three components.

6.5. Storage component

This section describes the high-level software structure of TeaPet's Storage Component.

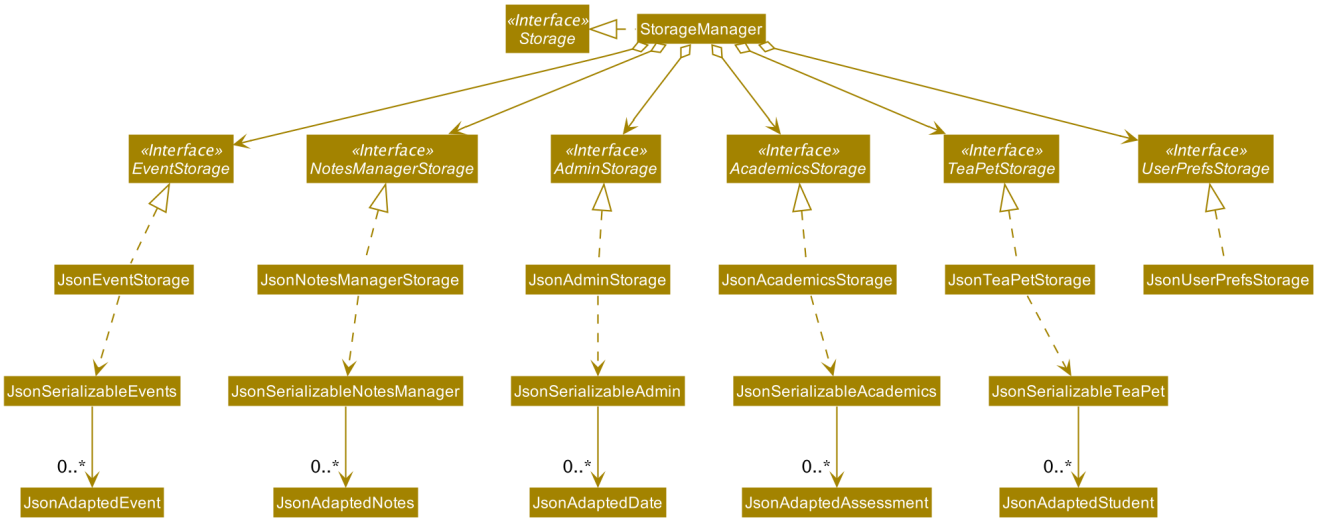


Figure 8. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

- converts Model objects into savable data in JSON-format and vice versa.
- can save `UserPref` objects in json format and read it back.
- can store `Students`, `Admin`, `Academics`, `Notes` and `Events` in a several json files, which can be read.

6.6. Common classes

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

7. Implementation

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

7.1. Student particulars feature

The student particulars feature keeps track of essential student details. The feature comprises of commands namely,

- `AddCommand` - Adds the student particulars into the class list
- `EditCommand` - Edits the particulars of a student
- `DeleteCommand` - Deletes the student information
- `FindCommand` - Finds information of the required student
- `ClearCommand` - Deletes all student details from the student list

The student commands all share similar paths of execution and is illustrated in the following sequence diagram below, which shows the sequence diagram for the StudentAddCommand.

The commands when executed, will interface with the methods exposed by the `Model` interface to perform the related operations (See logic component for the general overview).

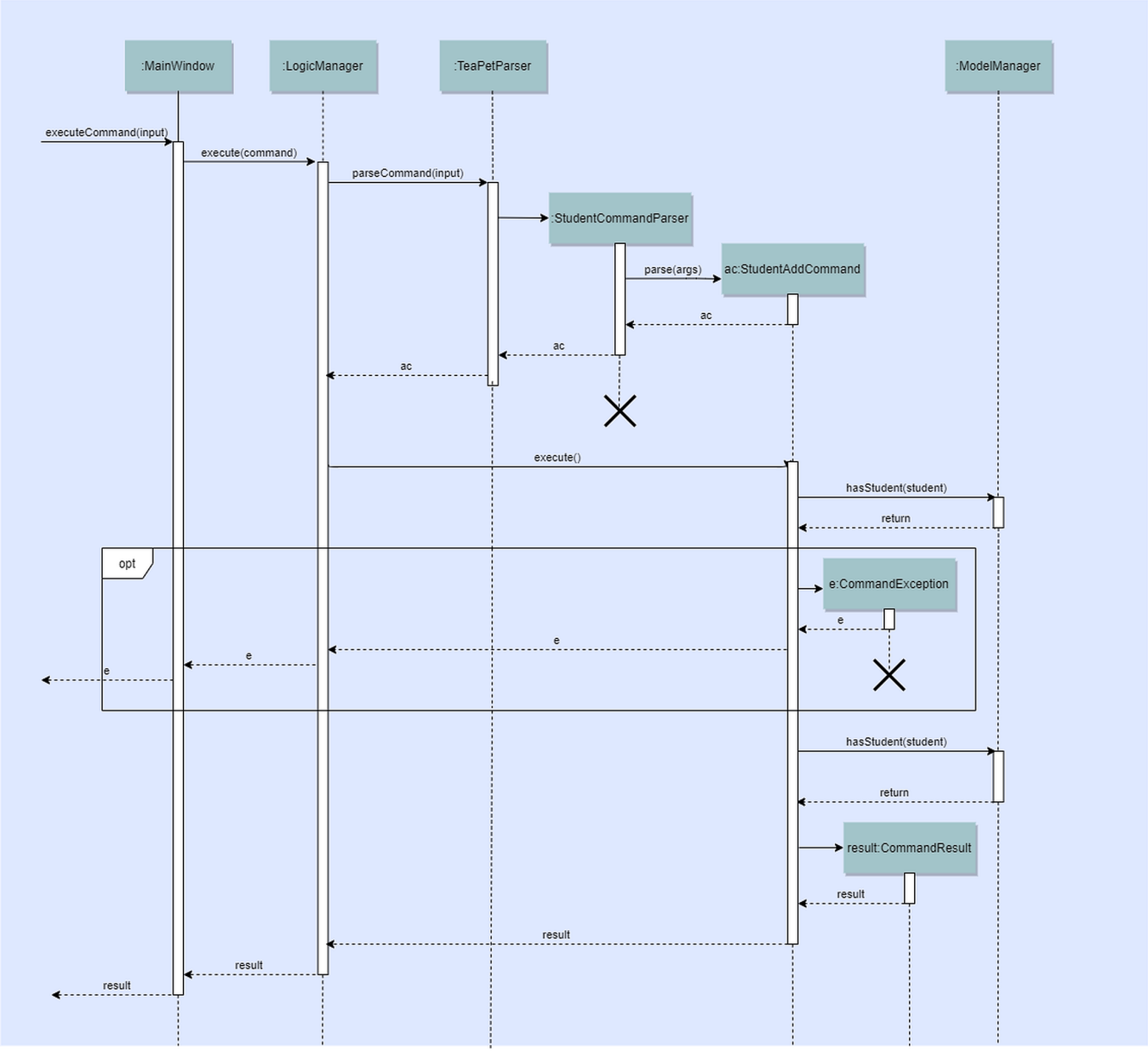


Figure 9. Sequence Diagram for StudentAddCommand

These are the common steps among the Student Commands:

1. The `TeaPetParser` will assign the `StudentCommandParser` to parse the user input
2. The `StudentCommandParser#parse` will take in a string of user input consisting of the arguments
3. The arguments are tokenized and the respective models of each argument are created.

7.1.1. Student Add command

Implementation

The following is a detailed explanation of the operations which `StudentAddCommand` performs.

1. After the successful parsing of user input, the `StudentAddCommand#execute(Model model)` method is called which validates the student defined.
2. As student names are unique, if a duplicate student is defined, a `CommandException` is thrown which will not add the defined student.
3. The method `Model#addStudent(Student student)` will then be called to add the student. The command box will be reflected with the `StudentAddCommand#MESSAGE_SUCCESS` constant and a new `CommandResult` will be returned with the success message.

NOTE	If the format or wording of adding a student contains error(s), the behaviour of TeaPet will be that either a unknown command or wrong format error message will be displayed.

4. The newly created student is added to the `UniqueStudentList`.

The following activity diagram summarizes what happens when a user executes the `student add` command:

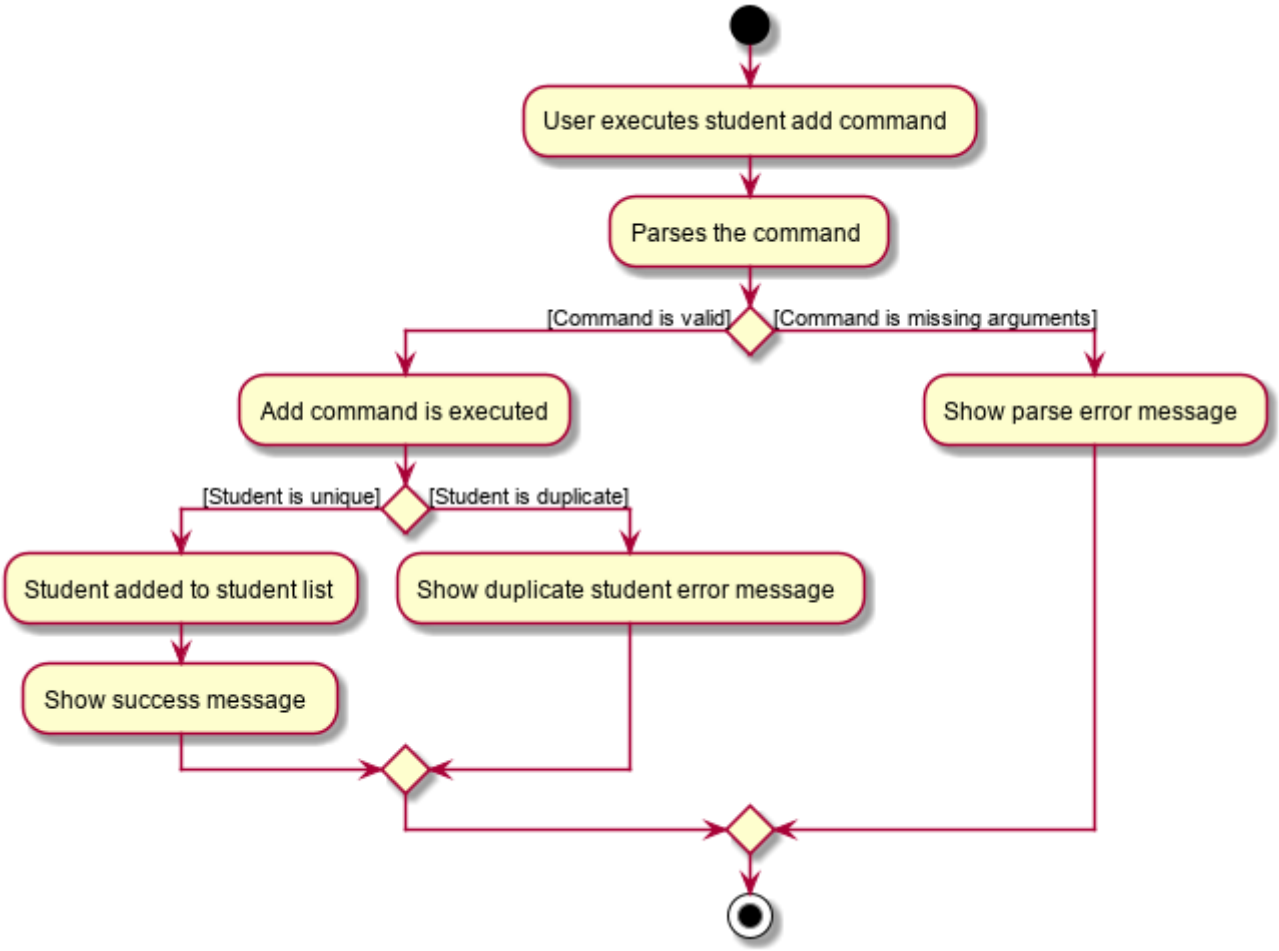


Figure 10. Sequence Diagram for StudentAddCommand

7.1.2. Student Edit command

Implementation

The following is a detailed explanation of the operations which `StudentEditCommand` performs.

1. After the successful parsing of user input, the `StudentEditCommand#execute(Model model)` method is called which checks if the `Index` is defined as an argument when instantiating the `StudentEditCommand(Index, index, EditStudentDescriptor editStudentDescriptor)` constructor. It uses the `StudentEditCommand.EditStudentDescriptor` to create a new edited student.
2. A new `Student` with the newly updated values will be created which replaces the existing `Student` object using the `Model#setStudent(Student target, Student editedStudent)` method.
3. The filtered student list is then updated with the new `Student` with the `Model#updateFilteredStudentList(PREDICATE_SHOW_ALL_STUDENTS)` method.
4. The command box will be reflected with the `StudentEditCommand#MESSAGE_SUCCESS` constant and a new `CommandResult` will be returned with the success message.

7.1.3. Student Delete command

Implementation

The following is a detailed explanation of the operations which `StudentDeleteCommand` performs.

1. After the successful parsing of user input, the `StudentDeleteCommand#execute(Model model)` method is called which checks if the `Index` is defined as an argument when instantiating the `StudentDeleteCommand(Index index)` constructor.

> **NOTE** The `Index` must be within the bounds of the student list.

2. The `Student` at the specified `Index` is then removed from the `UniqueStudentList#students` observable list using the `Model#deleteStudent(Index index)` method.
3. The command box will be reflected with the `StudentDeleteCommand#MESSAGE_SUCCESS` constant and a new `CommandResult` will be returned with the success message.

7.1.4. Student Find command

Implementation

The following is a detailed explanation of the operations which `StudentFindCommand` performs.

1. After the successful parsing of user input, the `StudentFindCommand#execute(Model model)` method is called which checks if the `NameContainsKeywordsPredicate(keywords)` is defined as part of the argument when instantiating the `StudentFindCommand(NameContainsKeywordsPredicate predicate)` constructor
2. The `Student` is then searched through the `UniqueStudentList#students` list using the `Model#hasStudent(Student student)` method to check if the `Student` already exists. If the `Student` does not exist, a `StudentNotFoundException` will be thrown and the `Student` will not be displayed.
3. The existing `UniqueStudentList#internalList` is then cleared and updated using the `Model#updateFilteredStudentList(Predicate predicate)` method.
4. A new `CommandResult` will be returned with the success message.

7.1.5. Student Clear command

Implementation

The following is a detailed explanation of the operations which `StudentFindCommand` performs.

1. After the successful parsing of the user input, the `StudentClearCommand#execute(Model model)` method is called.
2. The `Model#setTeaPet(ReadOnlyTeaPet teaPet)` method is then called which triggers the `TeaPet#resetData(ReadOnlyTeaPet newData)` method and creates a brand new student list to replace the old one.
3. A new `CommandResult` will be returned with the success message.

Design Considerations

Aspect: Command Syntax

- **Current Implementation:**
  - Current implementation of the feature follows just the command word syntax For example, `student`.
- **Alternatives Considered:**
  - We considered using the forward slash `/` before the command word, for example `/add`. However, we realise that it is redundant and will make inputs more tedious and confusing for users.

Aspect: Command Length:

- **Current Implementation:**
  - Commands are shortened as much as possible without much loss in clarity. For example, instead of using `/temperature`, we used `/temp` instead to input the students temperature into the application. Although this may be initially unfamiliar to users, it should be easy to pick up and make it less tedious during input.
- **Alternatives Considered:**
  - We considered using more descriptive arguments such that arguments are clear and succinct. However, this will definitely decrease the user expereince as the command will be too long to type.

7.1.6. Import image feature

This feature was included in TeaPet to help teachers easily identify the students using their pictures instead of just names. This feature utilises the `StudentCard#updateImage` method to update the images of students.

The feature comprises of one command namely, * `DefaultStudentDisplayCommand` - Refreshes the student list to show updated images of students and displays the student list.

This is further illustrated in the following sequence diagram below.

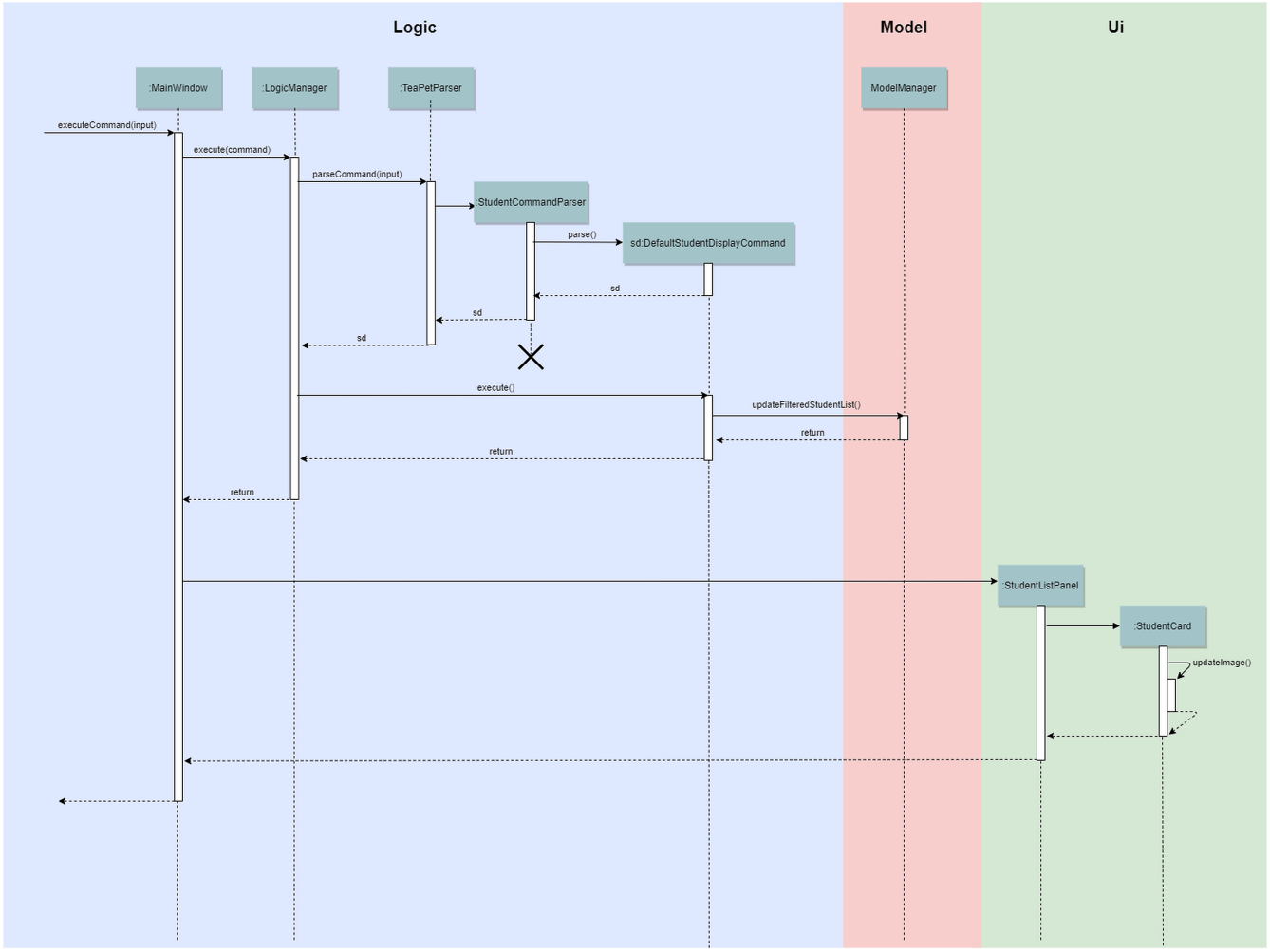


Figure 11. Sequence Diagram for StudentRefreshCommand

Implementation

The following is a detailed explanation of the operations which `DefaultStudentDisplayCommand` performs.

1. After the successful parsing of user input, the `DefaultStudentDisplayCommand#execute(Model model)` method is called. It does not require validation as it does not write into the student list.
2. The `StudentCardDefault#updateImage` method is then called which checks the image folder for the required png file and updates the student card.

> **NOTE** The name of the png file must match the name of the student.

3. If any view other than the student list view is showing on the `MainWindow`, the `MainWindow#handleDefaultStudent()` method will be called and the student list is now visible on the `MainWindow`.

The following activity diagram summarizes what happens when a user executes the `student` command:

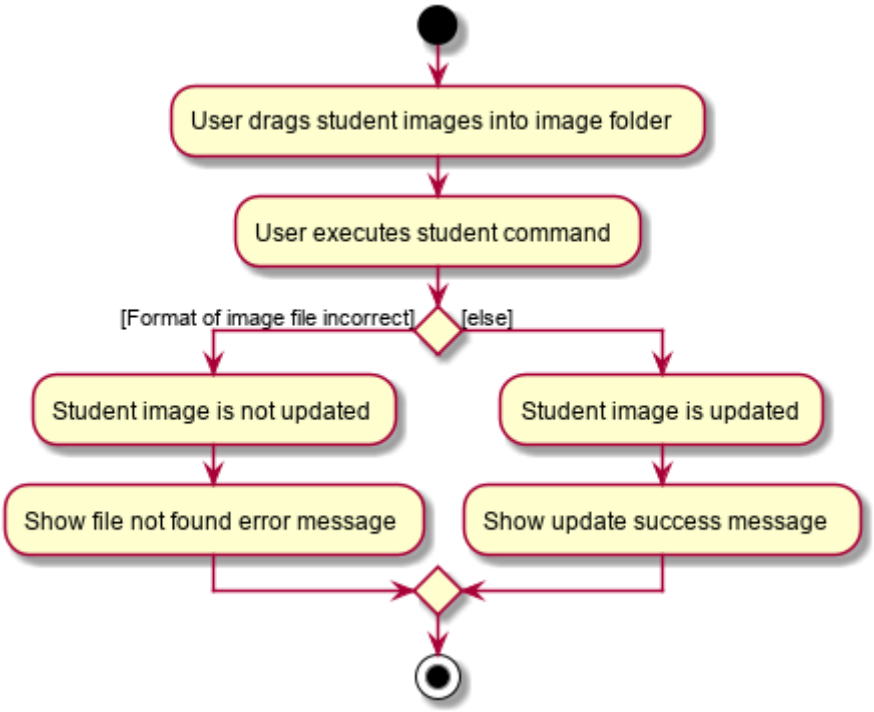


Figure 12. Activity Diagram for StudentRefreshCommand

Design Considerations

Aspect: Command Syntax

- **Current Implementation:**
  - Current implementation of the commands follows the command word syntax, followed by the arguments necessary for execution. For example, `student add/edit/delete/find/refresh`.
- **Alternatives Considered:**
  - We considered using a whole new command, `student refresh` to solely refresh and update images of the students. However, we realised that it would be more convenient for the user if we just add this functionality into the `student` command instead as it is able to both update the images and display the student list concurrently.

7.2. Class administration feature

The class administration feature keeps track of essential student administrative details. The feature comprises of four commands namely.

The structure of the Admin commands are as shown below:

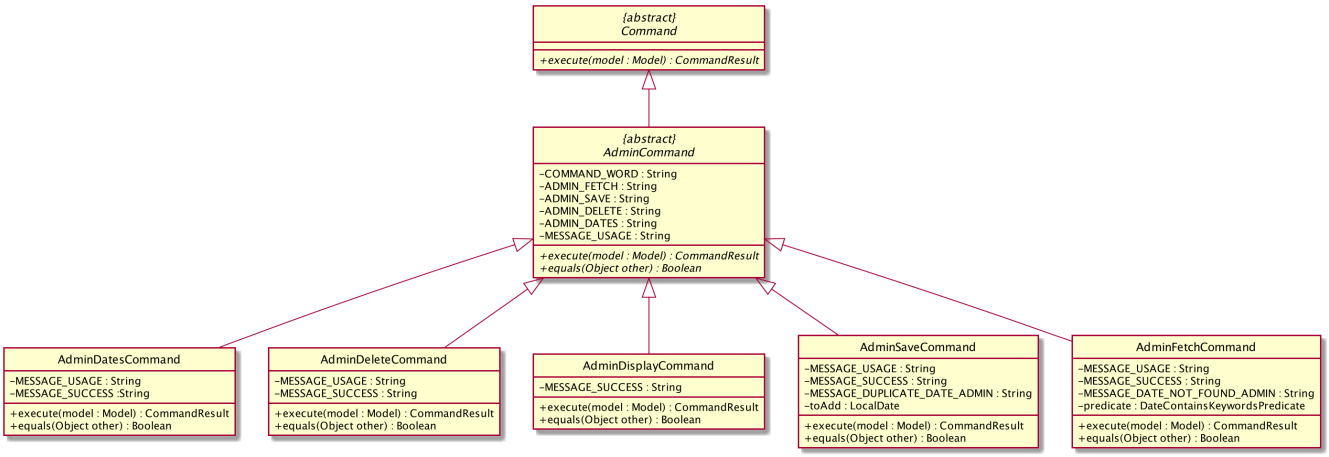


Figure 13. Admin Commands Diagram

These are the various admin commands to try:

- `admin` - Displays the most updated class administrative details.
- `admin dates` - Displays the dates that hold administrative information of the class.
- `admin save` - Saves today's administrative information of the class.
- `admin delete` - Deletes the administrative information of the class at the specified date.
- `admin fetch` - Fetches the administrative information of the class at the specified date.

7.2.1. Admin Display Command

Implementation

The following is a detailed explanation of the operations which `admin` performs.

Step 1. The `AdminDisplayCommand#execute(Model model)` method is executed which does not take in any arguments.

Step 2. The method `Model#updateFilteredStudentList(PREDICATE_SHOW_ALL_STUDENTS)` will then be called to update the filtered student list to show all current students in the student list.

NOTE	If the class list is empty, a blank page will be shown.

Step 3. The command box will be reflected with the `AdminDisplayCommand#MESSAGE_SUCCESS` constant and a new `CommandResult` will be returned with the message.

NOTE	If the wording of the `admin` command contains error(s), an unknown command message will be displayed.

7.2.2. Admin Dates Command

Implementation

The following is a detailed explanation of the operations which `admin dates` performs.

Step 1. The `AdminDatesCommand#execute(Model model)` method is executed which does not take in any arguments.

Step 2. The method `Model#updateFilteredDateList(PREDICATE_SHOW_ALL_DATES)` will then be called to update the filtered date list to show all current dates in the date list.

NOTE	If the date list is empty, a blank page will be shown.

Step 3. The command box will be reflected with the `AdminDatesCommand#MESSAGE_SUCCESS` constant and a new `CommandResult` will be returned with the message.

NOTE	If the format or wording of the `admin dates` command contains error(s), an unknown command or a wrong format message will be displayed.

7.2.3. Admin Save Command

Implementation

The following is a detailed explanation of the operations which `admin save` performs.

Step 1. The `AdminSaveCommand#execute(Model model)` method is executed which takes in today's date as an argument.

Step 2. The method `Model#updateFilteredStudentList(PREDICATE_SHOW_ALL_STUDENTS)` will then be called to update the filtered student list to show all current students in the student list.

Step 3. Sequentially, a date constructor will then called, creating a date object with today's date and `Model#getFilteredStudentList()`

Step 4. The method `Model#addDate(Date date)` will then be called to add the date. This will then trigger the `UniqueDateList#addDate(Date toadd)` method, which will throw `DuplicateDateException` if the date that is been added exists, with the duplicate dates error message.

Step 5. The command box will be reflected with the `AdminSaveCommand#MESSAGE_SUCCESS` constant and a new `CommandResult` will be returned with the message.

NOTE	If the format or wording of saving of a date contains error(s), an unknown command or wrong format error message will be displayed.

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

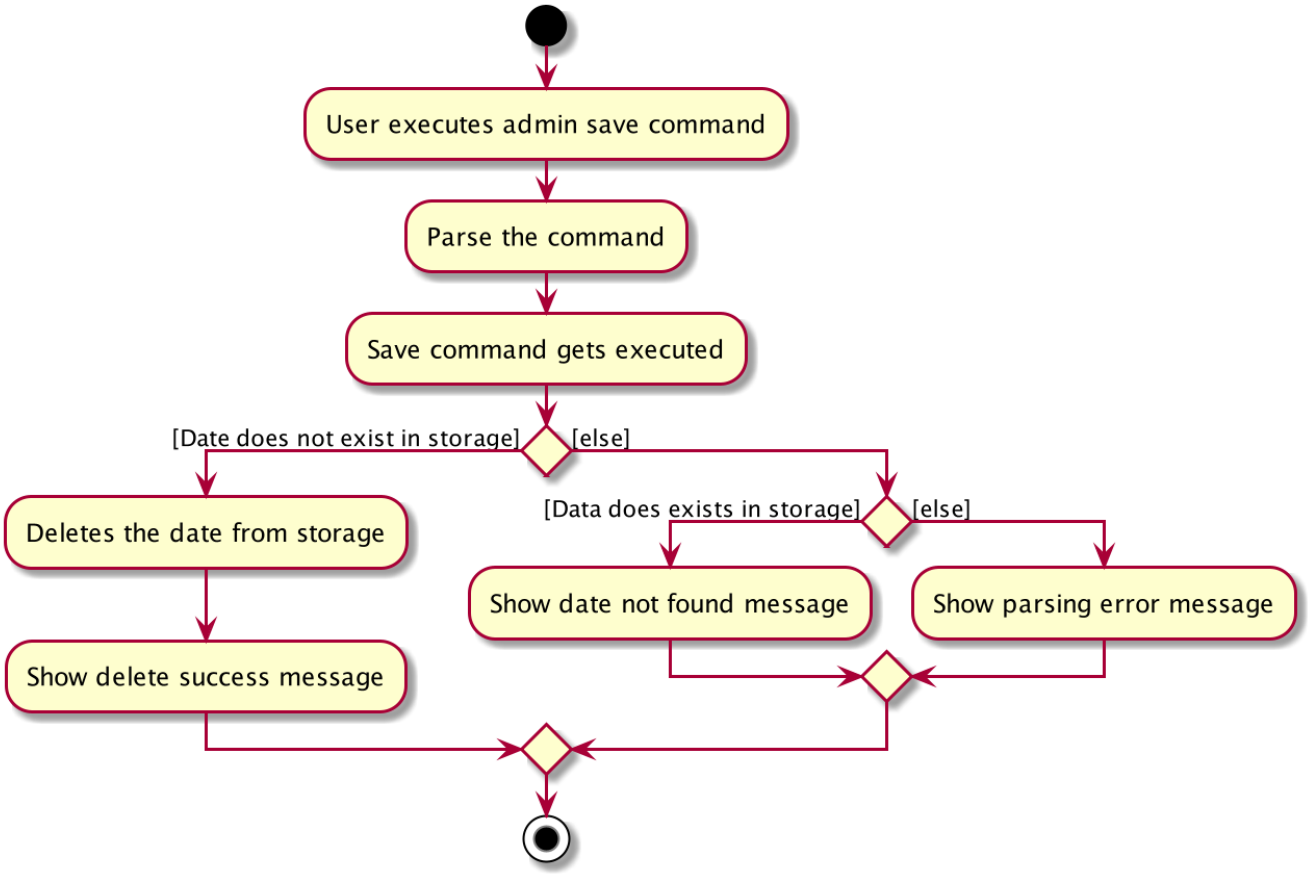


Figure 14. Admin Save Activity Diagram

Design Considerations

Aspect: Which date to save

- **Alternative 1 (current choice):** Saves the most updated administrative list as today's date.
  - Pros: Easy to implement and prevents mutation of dates.
  - Cons: The user will be unable to overwrite dates.
- **Alternative 2:** Saves the most updated administrative list as any date.
  - Pros: The user can mutate any dates as he or she wishes.
  - Cons: Hard to implement, and possible accidental mutation of dates.

Aspect: Allow overwriting of data

- **Alternative 1 (current choice):** Saving a date that exists in the storage is not allowed.
  - Pros: Easy to implement and prevent accidental mutation of data
  - Cons: Hard to implement.
- **Alternative 2:** Saving a date that exists in the storage is allowed.
  - Pros: User can make necessary changes to the dates where errors exists.
  - Cons: Hard to implement and could result in accidental mutation of dates.

7.2.4. Admin Delete Command

Implementation

The following is a detailed explanation of the operations which `admin save` performs.

Step 1. The `AdminDeleteCommand#execute(Model model)` method is executed which takes in a DateContainsKeywordsPredicate object as an argument. User input will be parsed first to a DateContainsKeywordsPredicate object before passing to the `AdminDeleteCommand` constructor.

NOTE	Date is to be entered in YYYY-MM-DD format, or a ParseException will be thrown and an error message will be displayed.

Step 2. The method `Model#updateFilteredStudentList(DateContainsKeywordsPredicate predicate)` will then be called to update the filtered date list to show the date that matches the given predicate. If no such date is found after searching through the `UniqueDateList#dates`, a DateNotFoundException will be thrown with an error message displayed.

Step 3. After the date has been found, the method `Model*deleteDate(Date target)` will then be called to remove the specified date from `UniqueDateList`.

The following sequence diagram shows how the add operation works:

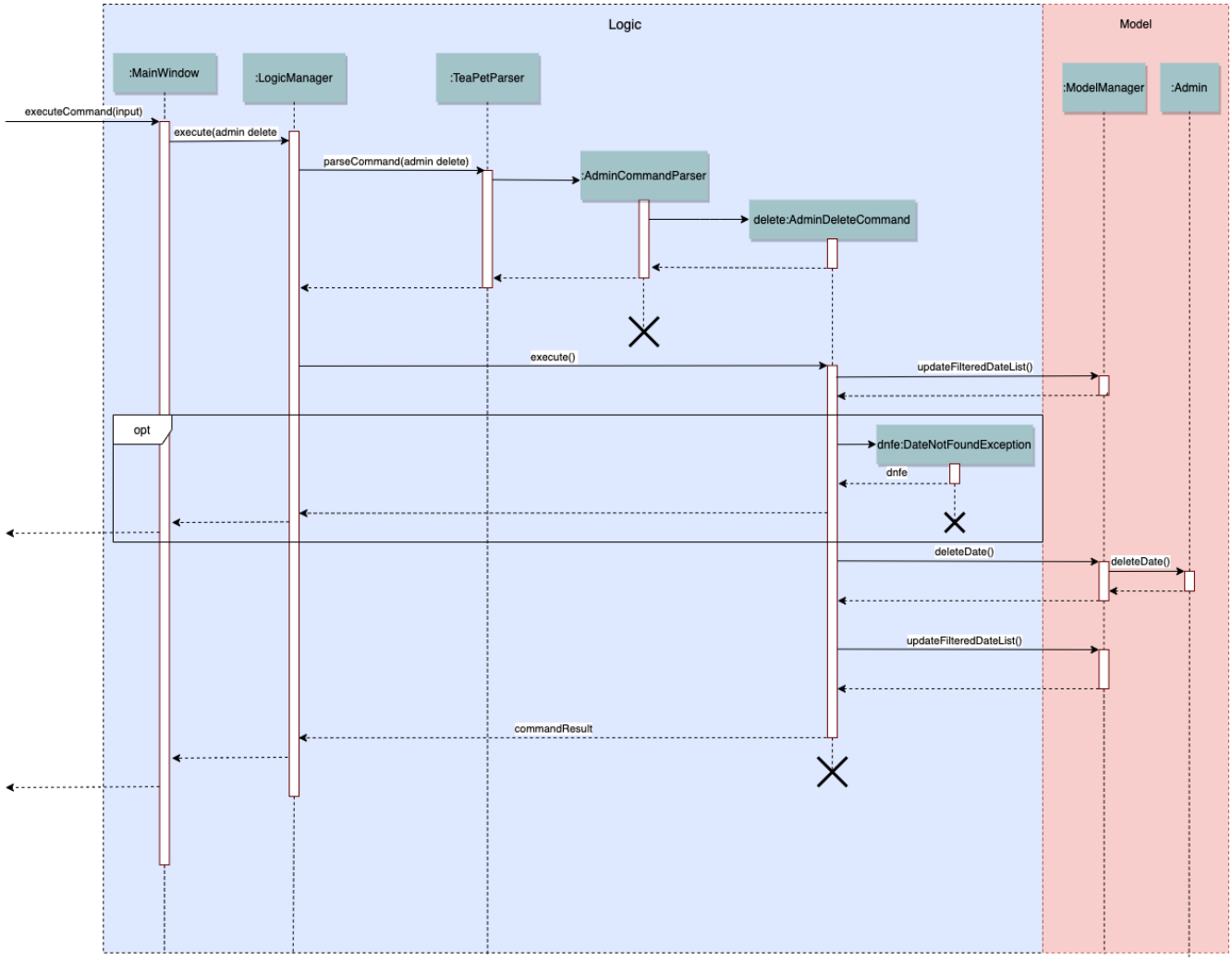


Figure 15. Admin Delete Sequence Diagram

7.2.5. Admin Fetch Command

Step 1. The `AdminFetchCommand#execute(Model model)` method is executed which takes in a DateContainsKeywordsPredicate object as an argument. User input will be parsed first to a DateContainsKeywordsPredicate object before passing to the `AdminFetchCommand` constructor.

NOTE	Date is to be entered in YYYY-MM-DD format, or a ParseException will be thrown and an error message will be displayed.

Step 2. The method `Model#updateFilteredStudentList(DateContainsKeywordsPredicate predicate)` will then be called to update the filtered date list to show the date that matches the given predicate. If no such date is found after searching through the `UniqueDateList#dates`, a DateNotFoundException will be thrown with an error message displayed.

NOTE	The sequence diagram for `admin fetch` command is similar to that of `admin delete` command.

7.3. Student Academics feature

This student academics feature stores and tracks the class' academics progress. The academics feature consists of four commands namely.

- `AcademicsCommand` - Displays the most updated student academics details.
- `AcademicsAddCommand` - Adds a new assessment to the academic list.
- `AcademicsEditCommand` - Edits the details of a particular assessment.
- `AcademicsDeleteCommand` - Deletes the specified assessment from academics.
- `AcademicsSubmitCommand` - Submits students' work to the specified assessment.
- `AcademicsMarkCommand` - Marks students' work of the specified assessment.
- `AcademicsDisplayCommand` - Displays either homework, exam, or the report of student academics.
- `AcademicsExportCommand` - Exports the academics information into a .csv file.

All academics commands share similar paths of execution. The commands when executed, will interface with the methods exposed by the `Model` interface to perform the related operations (See logic component for the general overview).

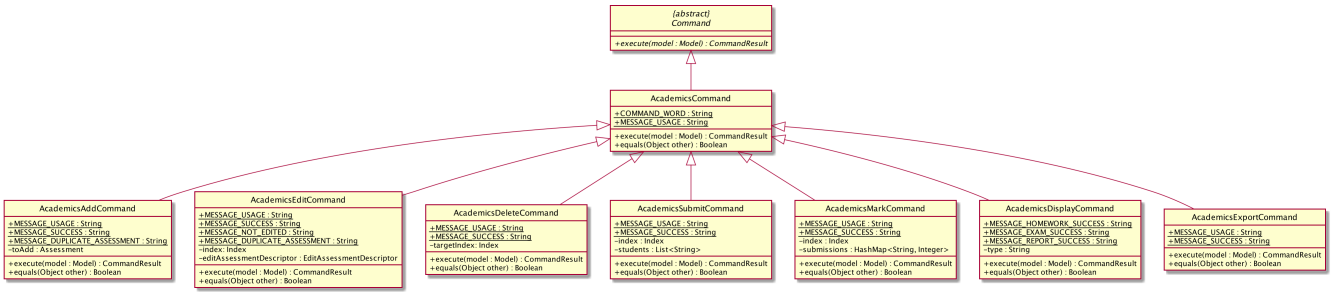


Figure 16. Academics Commands Diagram

7.3.1. Class Overview

The class diagram below will depict the structure of the Academics Model Component.

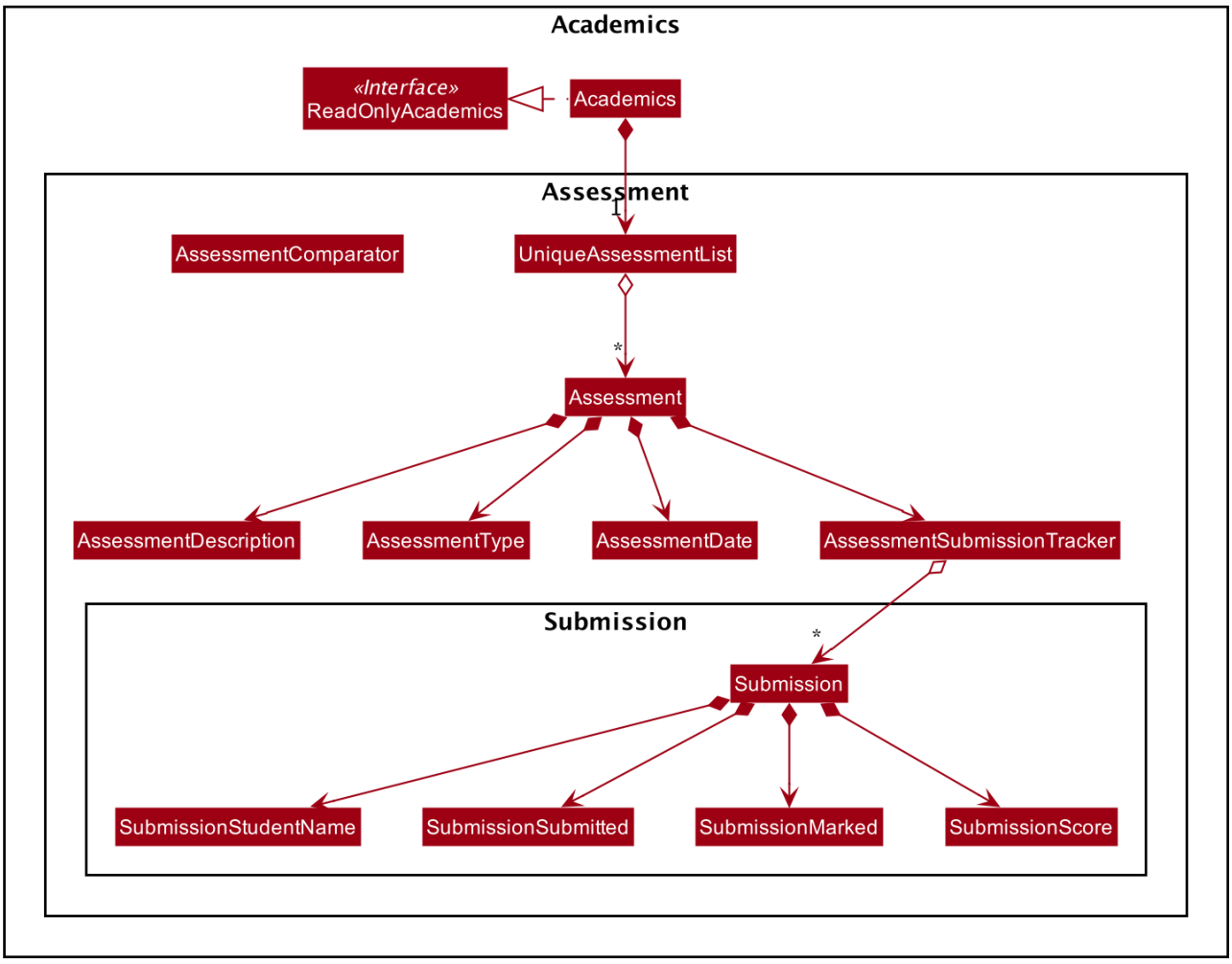


Figure 17. Academics Model Class Diagram

7.3.2. Academics Add Command

Implementation

The following is a detailed explanation of the operations which `AcademicsAddCommand` performs.

Step 1. The `AcademicsAddCommand#execute(Model model)` method is executed which takes in a necessary assessment description, type and date.

NOTE	Format for adding an assessment is `academics add desc/ASSESSMENT_DESCRIPTION type/TYPE date/DATE`.

Step 2. As assessment names should be unique, the `Model#hasAssessment(Assessment assessment)` method will check if the assessment already exists in `UniqueAssessmentList#assessments`. If a duplicate assessment is found, a `CommandException` will be thrown.

Step 3. Subsequently, the `Model#getFilteredStudentList()` method will then be called, to set the student submission tracker for the assessment.

Step 4. The method `Model#addAssessment(Assessment assessment)` will then be called to add the assessment. The command box will be reflected with the `AcademicsAddCommand#MESSAGE_SUCCESS` constant and a new `CommandResult` will be returned with the message.

NOTE	If the format or wording of adding an assessment contains error(s), an unknown command or wrong format error message will be displayed.

The following sequence diagram summarizes what happens when a user keys in an academics add command:

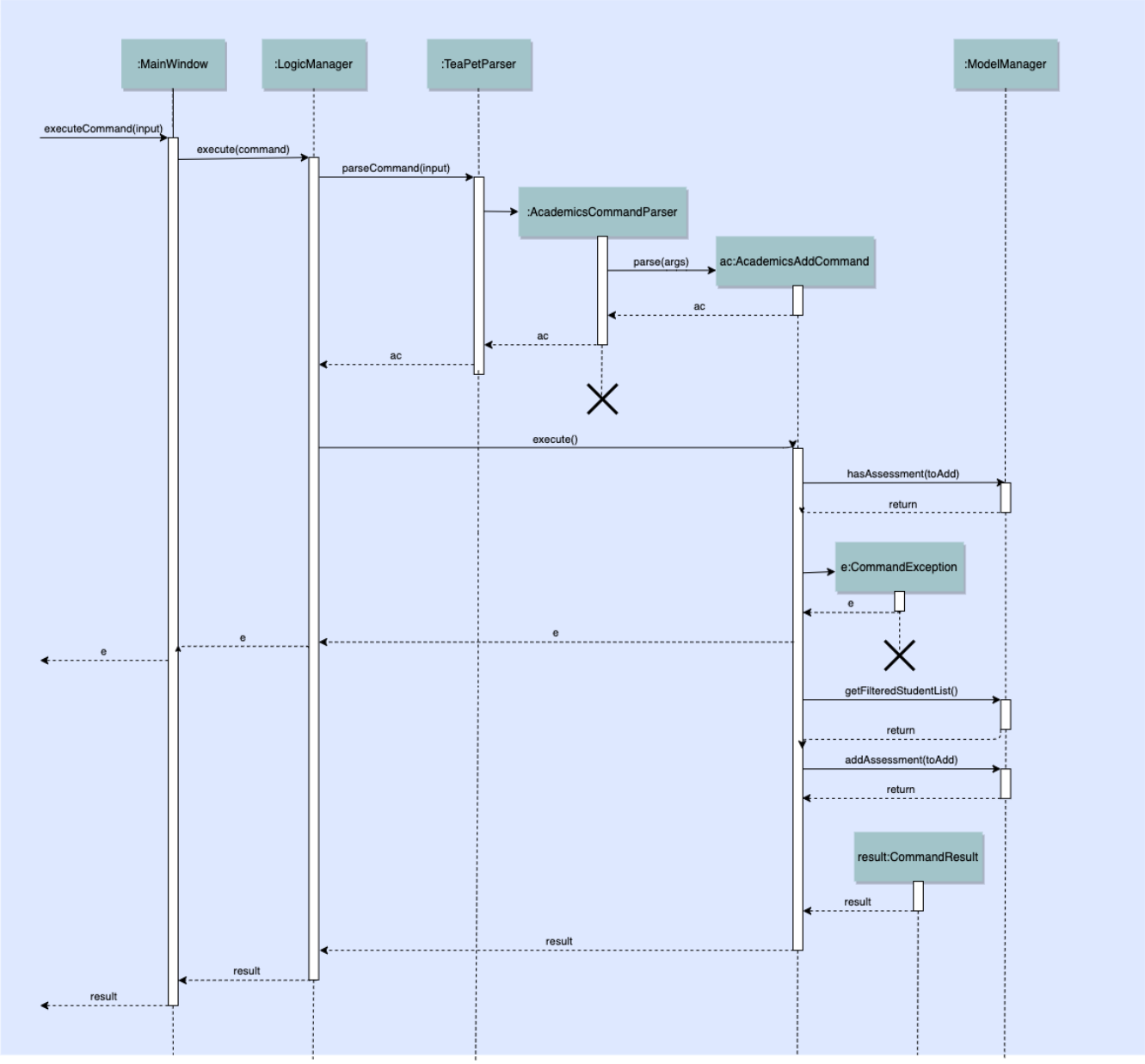


Figure 18. Academics Add Command Sequence Diagram

7.3.3. Academics Edit Command

Implementation

The following is a detailed explanation of the operations which `AcademicsEditCommand` performs.

Step 1. The `AcademicsEditCommand#execute(Model model)` method is executed which edits the details

of the specified assessment. The method checks if the `index` defined when instantiating `AcademicsEditCommand(Index index, EditAssessmentDescriptor editAssessmentDescriptor)` is valid. Since it is optional for users to input fields, the fields not entered will reuse the existing values that are currently stored and defined in the `Assessment` object.

NOTE	User needs to input at least 1 field of assessment to edit.

Step 2. A new `Assessment` with the newly updated values will be created which replaces the existing `Assessment` object using the `Model#setAssessment(Assessment target, Assessment editedAssessment)` method. However, if new assessment results in a duplicate assessment in `UniqueAssessmentList#assessments`, a `CommandException` will be thrown.

Step 4. The command box will be reflected with the `AcademicsEditCommand#MESSAGE_SUCCESS` constant and a new `CommandResult` will be returned with the message.

7.3.4. Academics Delete Command

Implementation

The following is a detailed explanation of the operations which `AcademicsDeleteCommand` performs.

Step 1. The `AcademicsDeleteCommand#execute(Model model)` method is executed which deletes the assessment at the specified index. It checks if the `Index` is defined as an argument when instantiating the `AcademicsDeleteCommand` constructor.

NOTE	The `Index` must be within the bounds of `UniqueAssessmentList#assessments`.

Step 2. The `Assessment` at the specified `Index` is then removed from `UniqueAssessmentList#assessments` observable list using the `Model#delete(Assessment assessment)` method.

Step 3. The command box will be reflected with the `AcademicsDeleteCommand#MESSAGE_SUCCESS` constant and a new `CommandResult` will be returned with the message.

7.3.5. Academics Submit Command

Implementation

The following is a detailed explanation of the operations which `AcademicsSubmitCommand` performs.

Step 1. The `AcademicsSubmitCommand#execute(Model model)` method is executed which submits students' work for the assessment at the specified index. The method checks if the `Index` is defined as an argument when instantiating the `AcademicsSubmitCommand` constructor.

NOTE	The `Index` must be within the bounds of `UniqueAssessmentList#assessments`.

Step 2. Subsequently, the `Model#hasStudentName(String studentName)` method will then check if the given student exists in `UniqueStudentList#students`. Also, `Model#hasStudentSubmitted(String studentName)` method checks if the student has already submitted their work for the specified

assessment. If the student does not exist or has already submitted their work, a `CommandException` will be thrown.

Step 3. The students' `Submission` will then be submitted to the specified `Assessment` using the method `Model#submitAssessment(Assessment assessment, List<String> students)`.

Step 4. The command box will be reflected with the `AcademicsSubmitCommand#MESSAGE_SUCCESS` constant and a new `CommandResult` will be returned with the message.

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

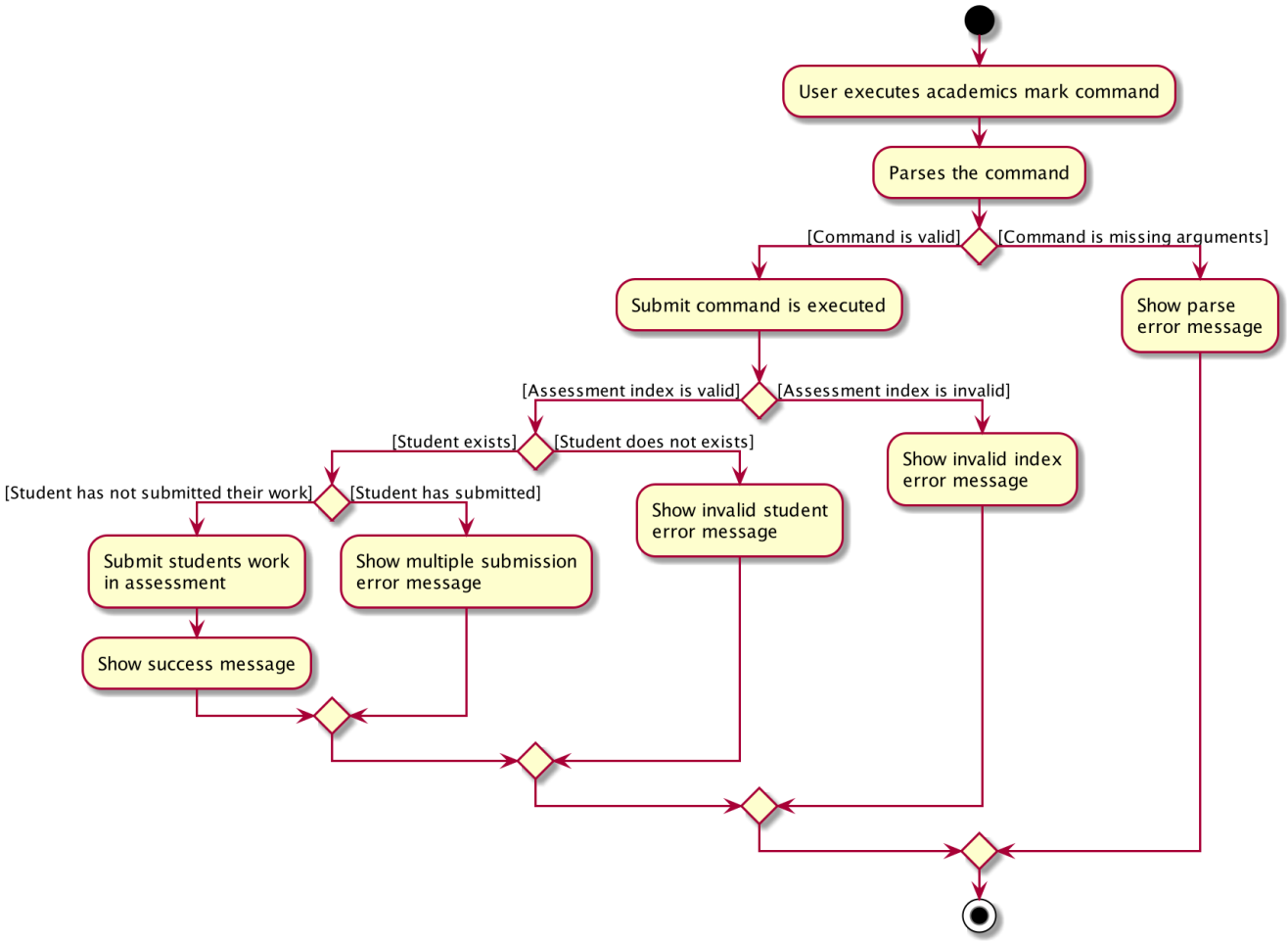


Figure 19. Academics Submit Activity Diagram

7.3.6. Academics Mark Command

Implementation

The following is a detailed explanation of the operations which `AcademicsMarkCommand` performs.

Step 1. The `AcademicsMarkCommand#execute(Model model)` method is executed which marks students' work and stores the students' scores for the assessment at the specified index. The method checks if the `Index` is defined as an argument when instantiating the `AcademicsMarkCommand` constructor.

NOTE	The `Index` must be within the bounds of `UniqueAssessmentList#assessments`.

Step 2. Subsequently, the `Model#hasStudentName(String studentName)` method will then check if the given student exists in `UniqueStudentList#students`. Also, `Model#hasStudentSubmitted(String studentName)` method checks if the student has yet to submit their work for the specified assessment. If the student does not exist or has not submitted their work, a `CommandException` will be thrown. Furthermore, the score should be between 0 and 100 inclusive, otherwise `CommandException` will also be thrown.

NOTE	Format for marking a students' work is `academics mark INDEX stu/STUDENT_NAME-SCORE`.

Step 3. The students' `Submission` will then be marked and its score will be stored in the specified `Assessment` using the method `Model#markAssessment(Assessment assessment, List<String> students)`.

Step 4. The command box will be reflected with the `AcademicsMarkCommand#MESSAGE_SUCCESS` constant and a new `CommandResult` will be returned with the message.

Design Considerations

Aspect: Allow submission along with marking

- **Alternative 1 (current choice):** Marking a submission that has not be submitted is not allowed.
  - Pros: Clearer and prevents confusion in data.
  - Cons: Harder to implement and the user will have to submit students' work separately.
- **Alternative 2:** Marking an unsubmitted work will also submit it for the assessment.
  - Pros: The user can just submit students work using the mark command, giving them less to type.
  - Cons: Prone to confusion of submitting and marking commands.

Aspect: Allow customizable total score of assessments

- **Alternative 1 (current choice):** Setting the total score for a submission is not allowed. (Total score for all submissions will be 100.)
  - Pros: Easy to implement and maintains uniformity of data.
  - Cons: User cannot set different total scores for assessments and have to grade it to a 100 weightage.
- **Alternative 2:** Setting the total score for a submission is allowed.
  - Pros: User can make set different total scores to different assessments according to its requirements.
  - Cons: Hard to implement and could result in inconsistency of data.

7.3.7. Academics Display Command

Implementation

The following is a detailed explanation of the operations which `AcademicsDisplayCommand` performs.

Step 1. The `AcademicsDisplayCommand#execute(Model model)` method is executed which can either take in no arguments or a 1 word argument indicating the type of display to show.

NOTE	Other than the default display (no arguments needed), there are only 3 types of displays: `homework`, `exam`, and `report`. Format: `academics` or `academics DISPLAY_TYPE`

Step 2. Depending on the display type, the command box will reflect its respective `AcademicsDisplayCommand#MESSAGE_SUCCESS` constant and a new `CommandResult` will be returned with the message.
Example. `homework` type display will reflect `AcademicsDisplayCommand#MESSAGE_HOMEWORK_SUCCESS`

NOTE	If the academics list is empty, a blank page will be shown.

NOTE	If the wording of the `academics` command contains error(s), an unknown command message will be displayed.

7.3.8. Academics Export Command

Implementation

The following is a detailed explanation of the operations which `AcademicsExportCommand` performs.

Step 1. The `AcademicsExportCommand#execute(Model model)` method is executed which exports the content of Academics into a csv file in the data folder.

NOTE	Format of the command is: `academics export`.

Step 2. The command box will be reflected with the `AcademicsExportCommand#MESSAGE_SUCCESS` constant and a new `CommandResult` will be returned with the message.

Step 3. Subsequently, the `CommandResult` will be processed by the`MainWindow` in the UI component and generate a studentAcademics.csv in the data folder of the current directory.

7.4. Notes feature

TeaPet application comes with an in-built notes feature, which serves to allow Teachers to record administrative or behavioural information of his/her students. Each note is tagged to a specific student and acts as a lightweight, digital 'Post It'.

The notes feature comprises of 6 main functionalities represented by 6 commands. They are namely:

- `NotesCommand` - Displays all notes and help on the notes feature.
- `NotesAddCommand` - Adds a new note into TeaPet.
- `NotesEditCommand` - Edits the details of a note in TeaPet.
- `NotesDeleteCommand` - Deletes a note in TeaPet.

- `NotesFilterCommand` - Filters the list of notes in TeaPet based on user-input keywords.
- `NotesExportCommand` - Exports all notes information into a .csv file

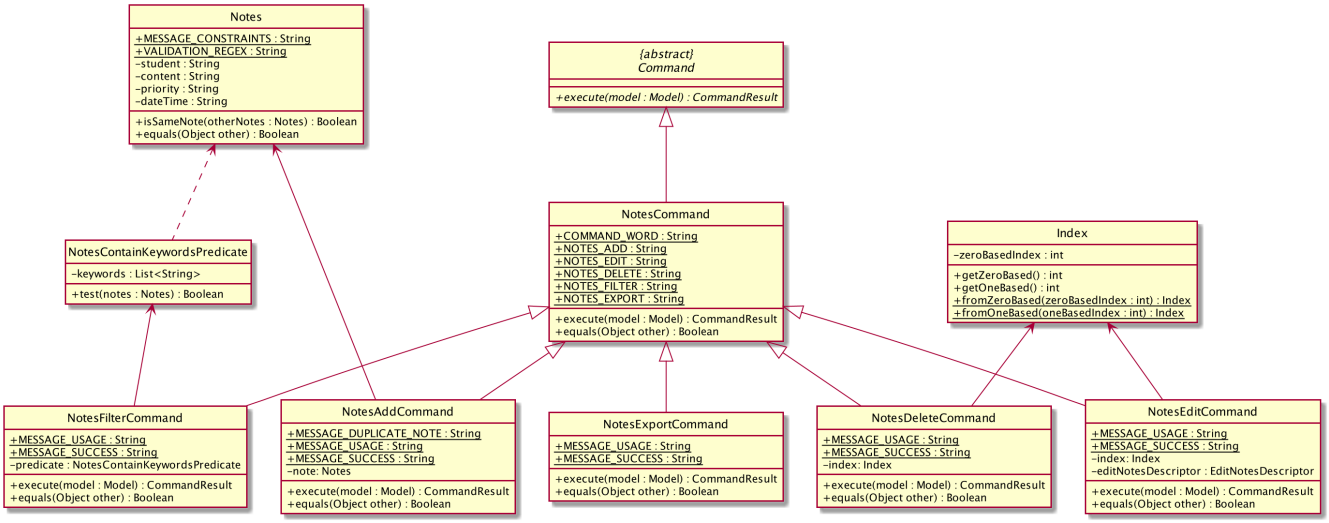


Figure 20. Class Diagram for Notes Commands.

7.4.1. Structure of Notes Class

Notes object is made up of 4 fields. They are namely: Student Name, Content, Priority and DateTime. The class diagram below depicts the structure of the Notes Model Component.

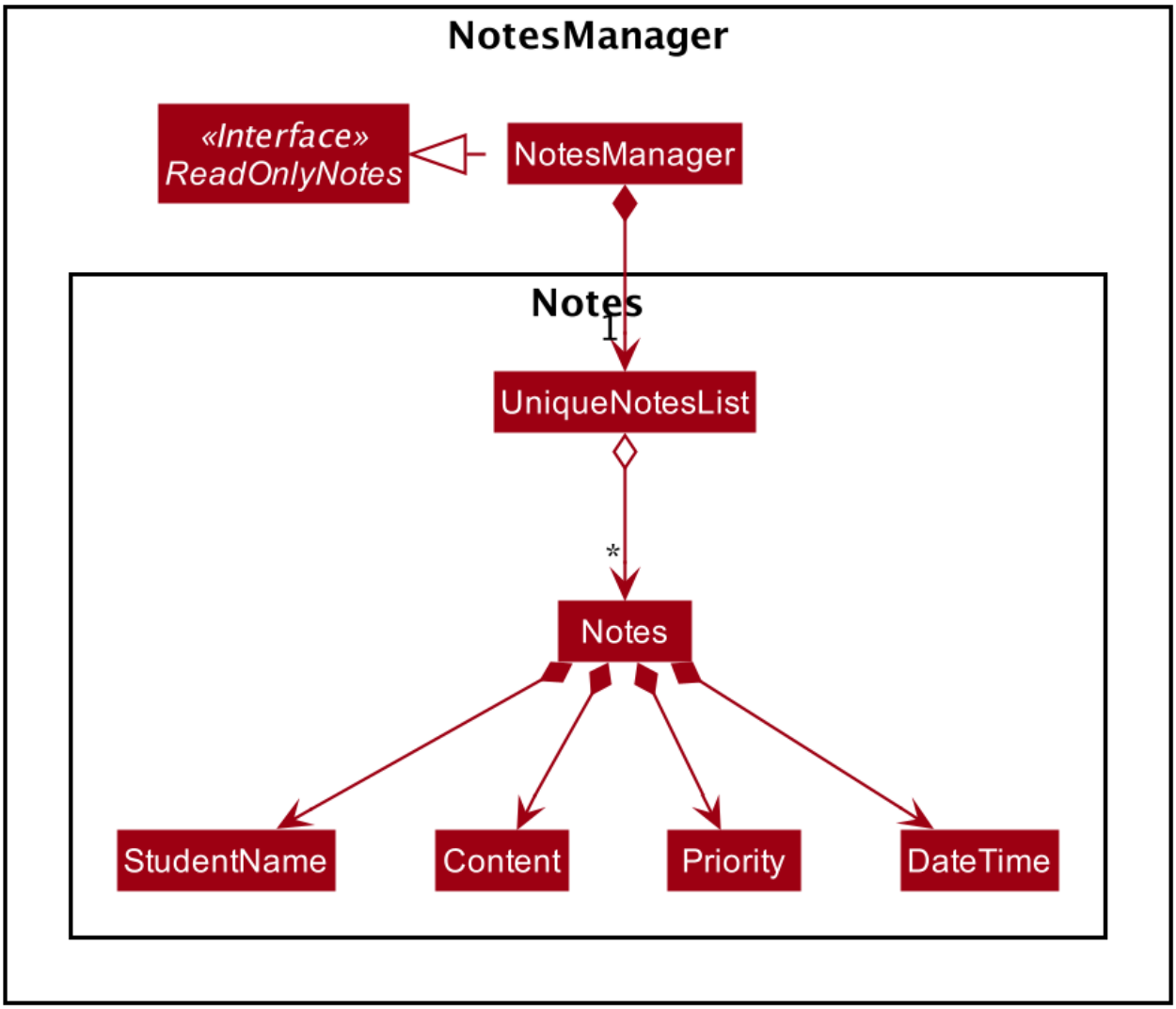


Figure 21. Class Diagram for Notes Model Component.

7.4.2. Add Note

The following is a detailed elaboration how `NotesAddCommand` operates.

NOTE	Format for adding a Note is `notes add name/STUDENT_NAME cont/CONTENT pr/PRIORITY`.
NOTE	Priority can only be either LOW, MEDIUM or HIGH, case-insensitive.

Step 1. The `NotesAddCommand#execute(Model model)` method is executed which takes in a necessary student name, content and priority as input

Step 2. The note is then searched through the `UniqueNotesList#notes` list using the `Model#hasNote(Notes note)` method to check if the note already exists. If the note exists, the `CommandException` will be thrown with the duplicate note error message.

Step 3. The method `Model#addNote(Notes note)` will then be called to add the note. The command box will be reflected with the `NotesAddCommand#MESSAGE_SUCCESS` constant and a new `CommandResult` will be returned with the message.

NOTE

If the format or wording of adding a student contains error(s), the behaviour of TeaPet will be similar to step 2, where either a unknown command or wrong format error message will be displayed.

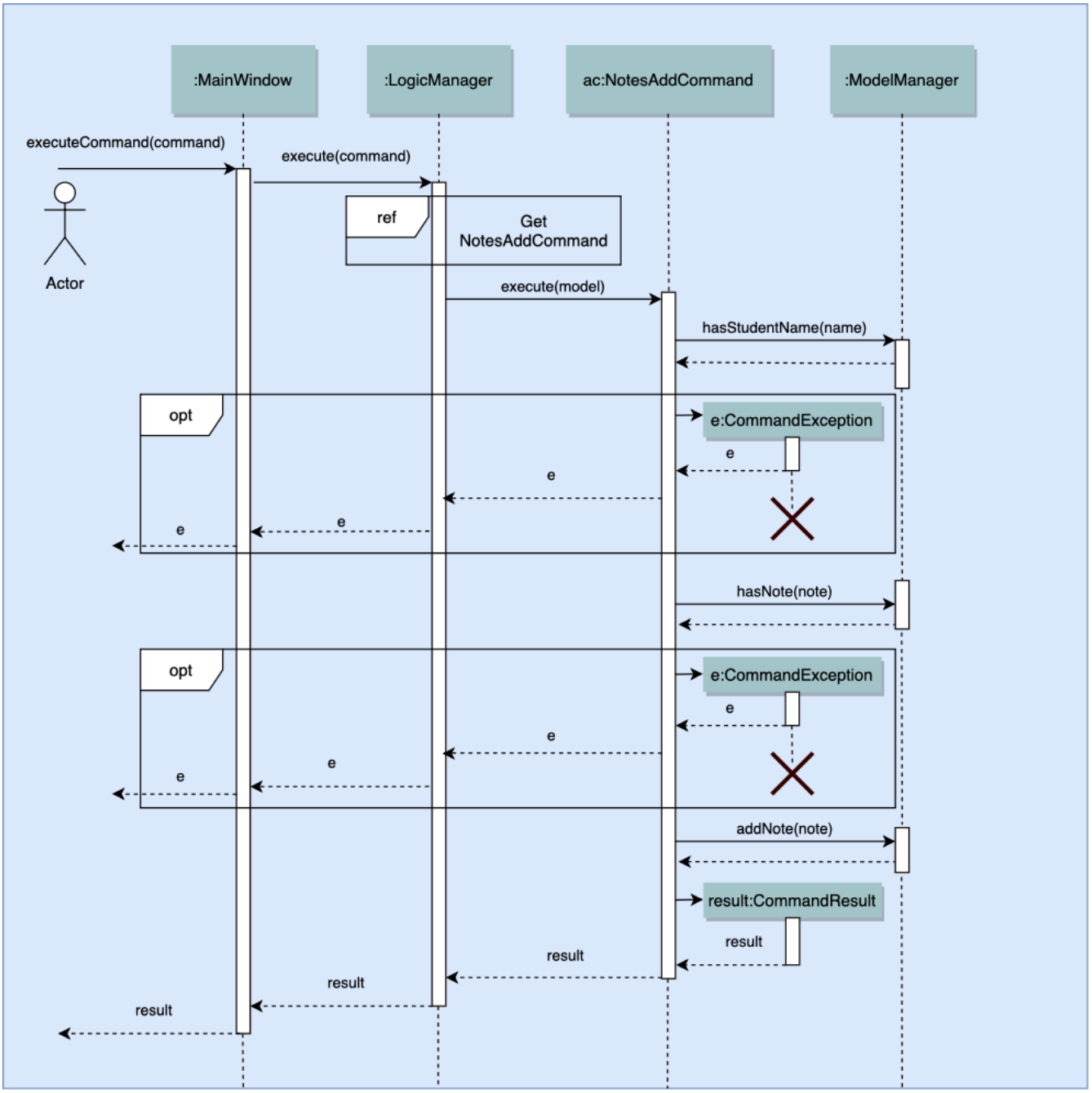


Figure 22. Sequence Diagram for Adding Notes.

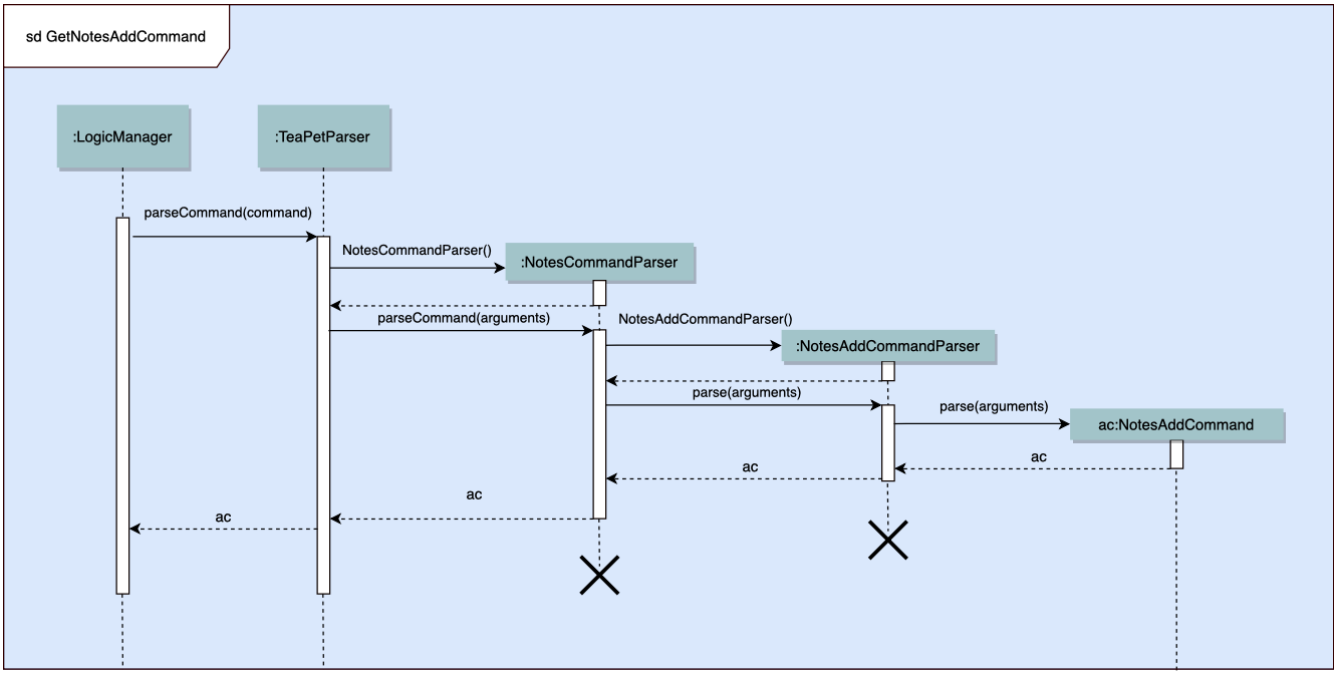


Figure 23. Supplementary Frame for Sequence Diagram.

7.4.3. Edit Note

The following is a detailed explanation of the operations which `NotesEditCommand` performs.

NOTE	Format for adding a Note is `notes edit INDEX [name/UPDATED_STUDENT_NAME] [cont/CONTENT] [pr/PRIORITY]`.

NOTE	Priority can only be either LOW, MEDIUM or HIGH, case-insensitive. Enclosing [] braces indicate optional fields. At least one of the three fields must be present.

Step 1. The `NotesEditCommand#execute(Model model)` method is executed which edit attributes of the selected note. The method checks if the `index` defined when instantiating `NotesEditCommand(Index index, EditNotesDecriptor editNotesDescriptor)` is valid. Since it is optional for users to input fields, the fields not entered will reuse the existing values that are currently stored and defined in the `Notes` object.

Step 2. A new `Notes` with the updated values is created and it is then searched through the `UniqueNotesList#notes` list using the `Model#hasNote(Notes note)` method to check if the note already exists. If the note exists, the `CommandException` will be thrown with the duplicate note error message.

Step 3. The newly created `Notes` will replace the old one through the `Model#setNote(Notes toBeChanged, Notes editedNote` method.

Step 4. The command box will be reflected with the `NotesEditCommand#MESSAGE_SUCCESS` constant and a new `CommandResult` will be returned with the message.

7.4.4. Delete Note

The following is a detailed explanation of the operations which `NotesDeleteCommand` performs.

1. After the successful parsing of user input, the `NotesDeleteCommand#execute(Model model)` method is called which checks if the `Index` is defined as an argument when instantiating the `NotesDeleteCommand(Index index)` constructor.

NOTE	The `Index` must be within the bounds of the student list.

2. The `Notes` at the specified `Index` is then removed from the `UniqueNotesList#notes` observable list using the `Model#deleteNote(Index index)` method.
3. The command box will be reflected with the `NotesDeleteCommand#MESSAGE_SUCCESS` constant and a new `CommandResult` will be returned with the success message.

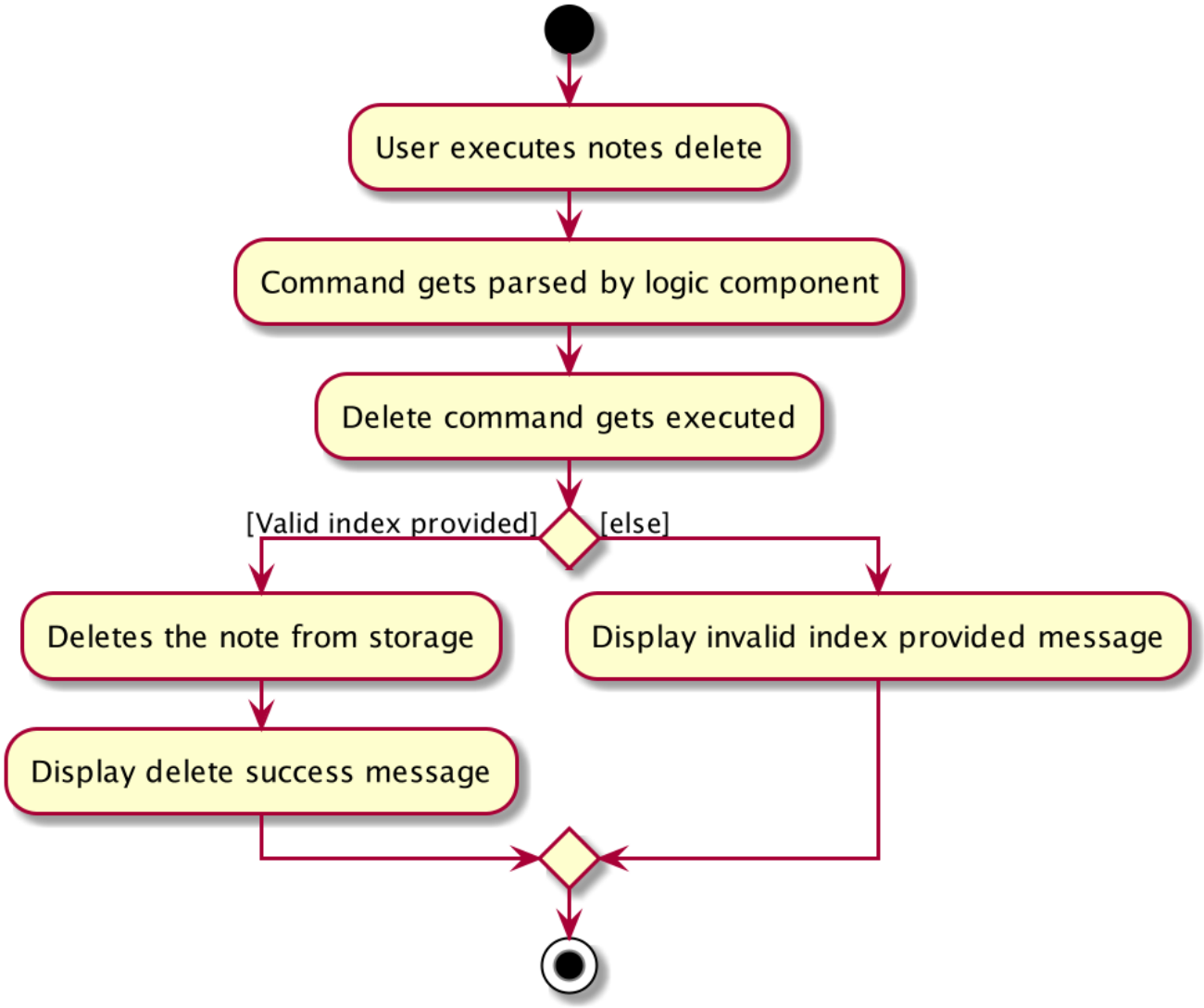


Figure 24. Activity Diagram for Deleting Note

7.4.5. Filter Notes

The following is a detailed explanation of the operations which `NotesFilterCommand` performs.

1. After the successful parsing of user input, the `NotesFilterCommand#execute(Model model)` method is called which checks if the `NotesContainsKeywordsPredicate(keywords)` is defined as part of the argument when instantiating the `NotesFilterCommand(NotesContainsKeywordsPredicate predicate)` constructor

2. The `Notes` is then searched through the `UniqueNotesList#notes` list... ...
3. The existing `UniqueNotesList#internalList` is then cleared and updated using the `Model#updateFilteredNotesList(Predicate predicate)` method.
4. A new `CommandResult` will be returned with the success message.

7.4.6. Export Notes

The following is a detailed explanation of the operations which `NotesExportCommand` performs.

1. After the successful parsing of user input, the `NotesExportCommand#execute(Model model)` method is called.
2. The command box will be reflected with the `NotesExportCommand#MESSAGE_SUCCESS` constant and a new `CommandResult` will be returned with the success message.
3. The `MainWindow` of the UI component will process the `CommandResult` and create a studentNotes.csv in the data folder of the current directory.

Design Considerations

- Alternative 1 (Current Choice): Intuitive, simple syntax and user-friendly
  - Pros: It is easy for the Teacher to use the feature.
  - Cons: Not as powerful and less utility for advanced users.
- Alternative 2: Many additional fields including special tags, reminders, etc.
  - Pros: Powerful, many interesting features that advanced users can use.
  - Cons: It contradicts with the initial goal of the Notes feature which is to enable quick and easy note-taking.

7.5. Schedule feature

7.5.1. Overview

The schedule feature enables teachers to add, delete, edit and view events in their personal scheduler. This feature is built based on the Jfxtras iCalendarAgenda library. The iCalendarAgenda object is used on the UI side to render VEvents. The VEvent object takes in data such as event name, start date time, end date time, recurrence of events, etc.

NOTE	VEvent object is used primarily throughout the application as it is the required object type for the iCalendarAgenda library. Hence, at the storage level, the Event objects are mapped to VEvents for reading purposes and vice versa for saving purposes.

The feature comprises of the the following commands:

- `EventAddCommand` - Creates a new event.

To add:

7.5.2. Class Overview

The class diagram below shows the interactions between events classes in the `Model`. Notice how the `EventHistory` class depends on the `Event` class in its constructor but only has a `VEvent` attribute. This is because an `Event` object will always be mapped to a `VEvent` within the `EventHistory` class. Some methods of `EventHistory` has been omitted for brevity as they are mostly `VEvent` based, which then again highlights that the interactions with the `Logic` and `UI` components are mostly done using the `VEvent` type class. Only the `Storage` component works with `Event` type class.

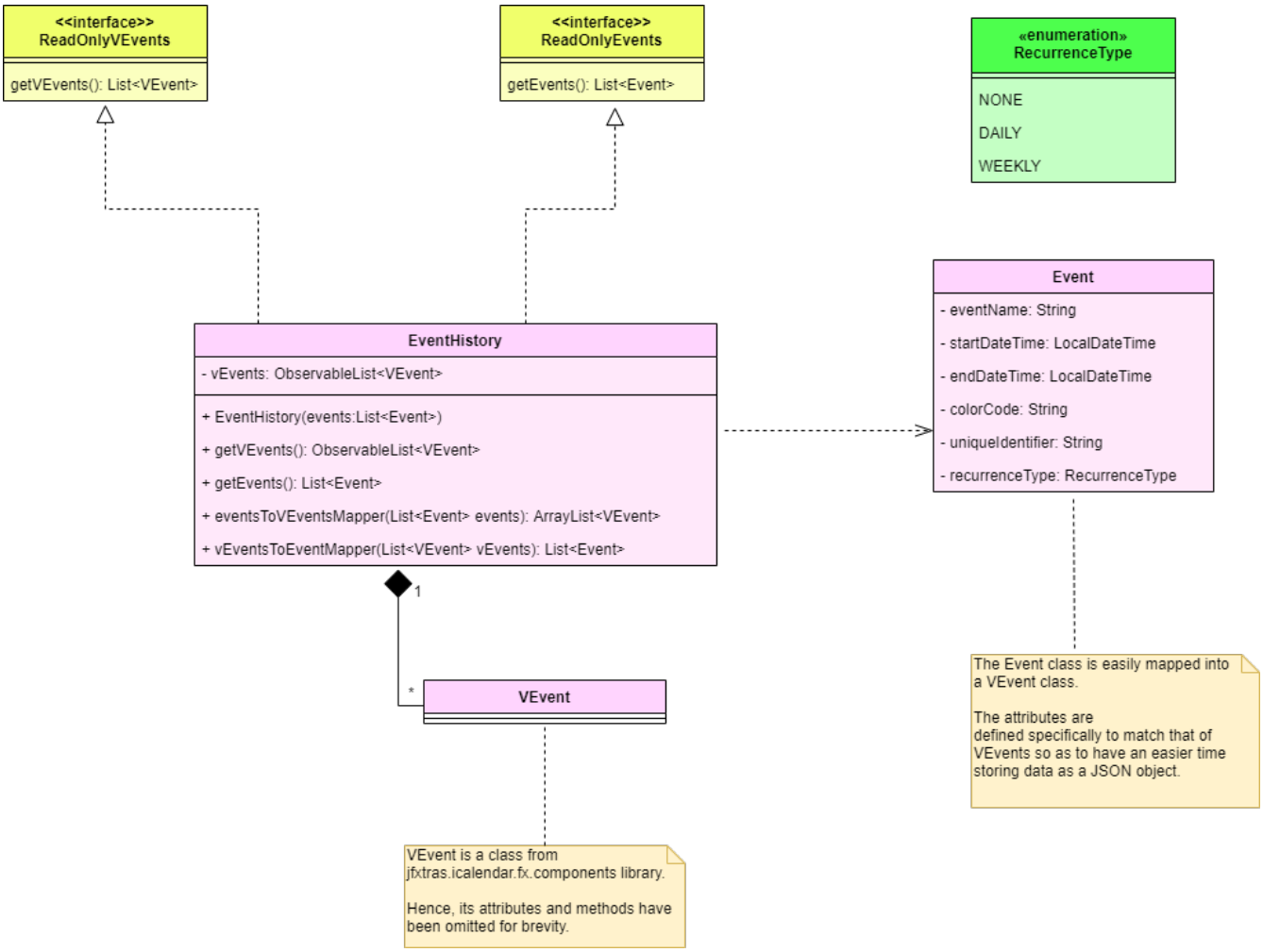


Figure 25. Schedule Class Diagram

7.5.3. Schedule Add Command

Implementation

The following is a detailed explanation which `schedule add` performs.

Step 1: The `EventAddCommand#execute(Model model)` method is called which validates if the `VEvent` object from the parser is valid.

Step 2: The method `Model#addVEvent(VEvent vEvent)` is then called which adds the new `VEvent` to the `EventHistory`. The `VEvent` is validated to check if it is unique using the `EventUtil#isEqualVEvent(VEvent vEvent)` method.

Step 3: If the event is invalid, a `CommandException` will throw an error message. Else, a new

`CommandResult` will be returned with the success message.

Step 4: The `LogicManager` then calls the `Storage#saveEvents(ReadOnlyEvents readOnlyEvents)` which saves the `EventHistory` in JSON format after serializing it using the `JsonEventStorage`.

NOTE	The `ReadOnlyEvents` and `ReadOnlyVEvents` interfaces are an abstraction of the implementation of the `EventHistory` from other layers of the application.

The following activity diagram summarizes what happens when a user executes the `schedule add` command:

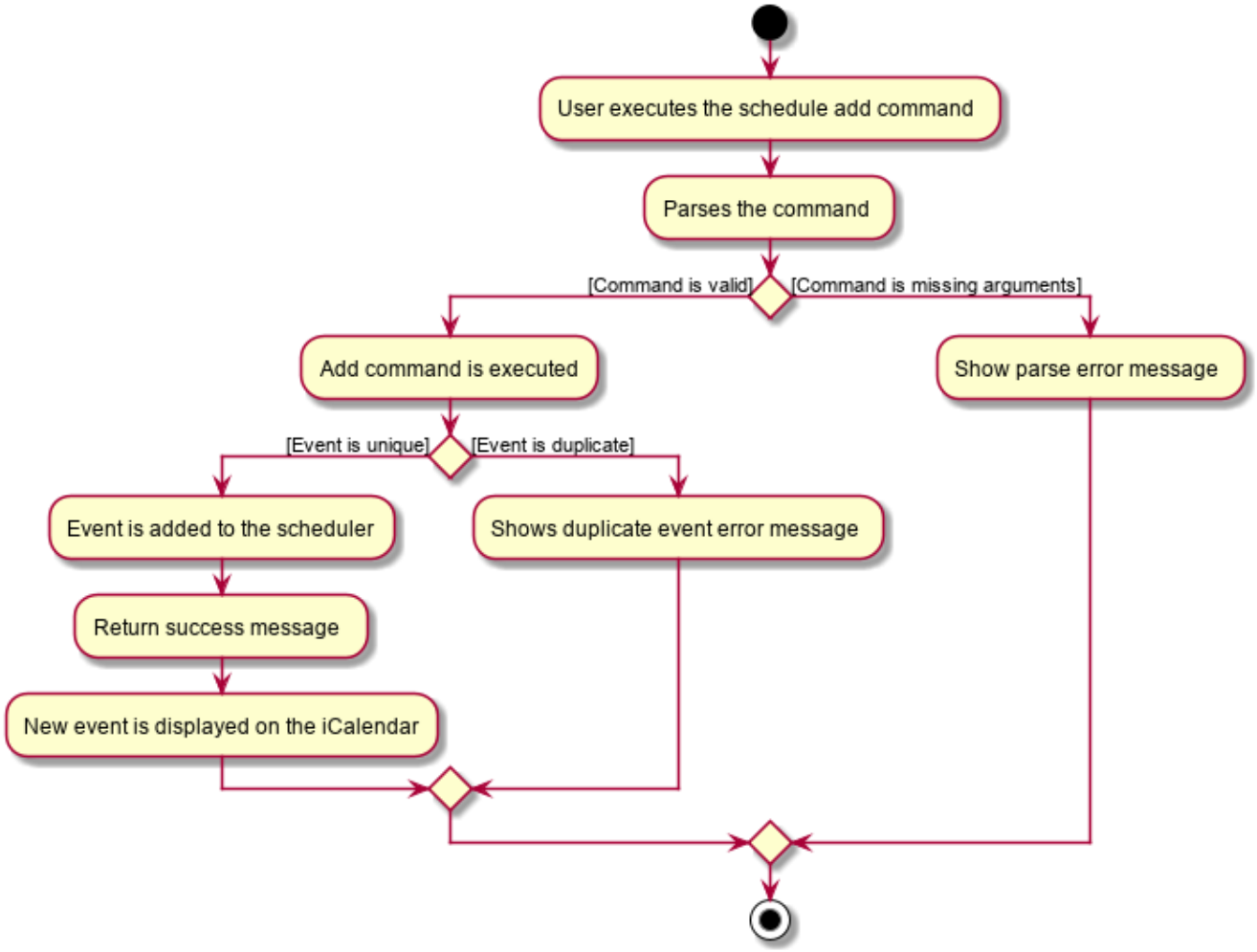


Figure 26. Schedule Add Activity Diagram

Design Considerations

Aspect: Command Clarity

- **Current Implementation:**
  - `schedule add eventName/Consultation startDateTime/2020-04-08T09:00 endDateTime/2020-04-08T11:00 recur/none color/13`
  - We currently have full names for prefixes such as eventName/ instead of name/, as well as slightly lengthier prefixes such as startDateTime/ and endDateTime/. Although this may be slightly more tedious, we believe that it is clearer as there are other very similar prefixes in our other features such as name/ and date/.

- **Alternatives Considered:**
  - `schedule add name/Consultation startDateTime/2020-04-08T09:00 endDateTime/2020-04-08T11:00 recur/none color/13`
  - By doing this, users may be confused as the Academics feature, Student feature and Notes feature require name as a prefix as well. Furthermore, the name required here is not the name of the student but the name of the event.

7.6. Logging

This section describes how TeaPet record it's logs.

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 7.7, "Configuration")
- The `Logger` for a class can be obtained using `LogsCenter.getLogger(Class)` which will log messages according to the specified logging level
- Currently log messages are output through: `Console` and to a `.log` file.

Logging Levels

- `SEVERE` : Critical problem detected which may possibly cause the termination of the application
- `WARNING` : Can continue, but with caution
- `INFO` : Information showing the noteworthy actions by the App
- `FINE` : Details that is not usually noteworthy but may be useful in debugging e.g. print the actual list instead of just its size

7.7. Configuration

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

8. Documentation

Refer to the guide here.

9. Testing

Refer to the guide here.

10. Dev Ops

Refer to the guide here.

Appendix A: Product Scope

Target user profile:

- form teacher of a class
- has a need to manage a significant number of students
- has a need to take the attendance of students
- wants to be able to track the homework and progress of students
- wants to be able to keep a schedule of his/her classes and events
- wants to be able to keep track of students' behavior in class
- prefer desktop applications over other types
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition: Ability to manage students administration and personal commitments better than a typical mouse/GUI driven application. Overall increase in productivity.

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…
* * *	new user	see usage instructions	refer to instructions when I forget how to use the App
* * *	form teacher	take the attendance of my students	know who is present for my class
* * *	form teacher	have a schedule tracking my events	know what I need to attend/do in a day

Priority	As a …	I want to …	So that I can…
* * *	form teacher	maintain of a list of students who have completed my homework	know who has not submitted my homework
* * *	form teacher	take down notes for student's behavior	track the behaviour of my students
* * *	form teacher	see the scores of my class	track the academic progress of my class
* * *	form teacher	add students	add new students to the class list
* * *	form teacher	delete a student	remove students that I no longer need
* * *	form teacher	find a student by name	locate details of students without having to go through the entire list
* * *	form teacher	sort students by alphabetical order	locate a student easily
* * *	form teacher	update the details of my students	make necessary changes to my student's particulars
* * *	form teacher	maintain emergency contacts of my students	know who to contact in case of emergency
* *	form teacher	specify if a student is late or absent for class	know why my student is absent

Priority	As a …	I want to …	So that I can…
* *	user	hide private contact details by default	minimize chance of someone else seeing them by accident
* *	form teacher	keep track of the sitting arrangement of the class	students who change their seats unknowingly
* *	form teacher	record the temperature of students	track the health of my students
*	form teacher	get feedback from other teachers teaching the students of my class	better understand the progress of the class

{More to be added}

Appendix C: Use Cases

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

Use case: UC01 - Add student

MSS

1. User enters a student name, followed by optional attributes such as emergency contacts, through the command line.
2. TeaPet adds the student and his/her attributes to the class list.
3. TeaPet displays feedback to the user that a new student is being added.

   Use case ends.

Extensions

1a. Student is invalid.

1a1. TeaPet shows an error message.

Use case ends.

1b. Particulars are invalid.

1b1. TeaPet shows an error message.

Use case ends.

Use case: UC02 - Edit student

MSS

1. User specifies which student, using the name, and what particulars he/she wants to edit in the command line.
2. TeaPet edits the student's particulars in the class list as instructed by the commands.
3. TeaPet displays feedback to the user that the student has been edited, followed by the changes made.

Use case ends.

Extensions

1a. Student is invalid.

1a1. TeaPet shows an error message.

Use case ends.

1b. Particulars are invalid.

1b1. TeaPet shows an error message.

Use case ends.

Use case: UC03 - Delete student

MSS

1. User specifies which student, using the index, he/she wants to remove.
2. TeaPet removes the student from the class list.
3. TeaPet displays feedback to the user that the student is being removed.

Use case ends.

Extensions

1a. Student name entered is invalid.

1a1. TeaPet shows an error message.

Use case ends.

Use case: UC04 - Add event

MSS

1. User keys in an event.
2. TeaPet adds the event to the schedule.
3. TeaPet feedback the event has been added.

   Use case ends.

Extensions

1a. Event entered is invalid.

   1a1. TeaPet shows an error message.

   Use case ends.

Use case: UC05 - Display Schedule

MSS

1. User keys in the command to display events.
2. TeaPet displays the events in chronological order.

   Use case ends.

Extensions

1a. Command is invalid.

   1a1. TeaPet shows an error message.

   Use case ends.

Use case: UC06 - Display default class list.

MSS

1. User enters the command to display the default version of the class list.
2. TeaPet displays the class list with the students' tags, mobile number, email, and notes.

   Use case ends.

Extensions

1a. Command is invalid.

   1a1. TeaPet shows an error message.

Use case ends.

Use case: UC07 - Display admin class list.

MSS

1. User enters the command to display the administrative version of the class list.
2. TeaPet displays the class list with the students' attendance status and temperature.

Use case ends.

Extensions

1a. Command is invalid.

1a1. TeaPet shows an error message.

Use case ends.

Use case: UC08 - Deleting a date from the list of dates.

MSS

1. User enters the command to delete a specific date from the list of dates.
2. TeaPet searches through the list of dates for the date provided by the user.
3. TeaPet removes that date from the date list.
4. TeaPet displays that the date has been deleted successfully.

Use case ends.

Extensions

1a. Command is invalid.

1a1. TeaPet shows an error message.

Use case ends.

1b. Command is in incorrect format.

1b1. TeaPet shows an error message displaying the correct format for the command.

Use case ends.

1c. Date is in incorrect format.

1c1. TeaPet shows an error message displaying the correct format for date.

Use case ends.

2a. Date provided is not in the list of dates.

2a1. TeaPet shows an error message displaying date is not found.

Use case ends.

Use case: UC09 - Display detailed class list.

MSS

1. User enters the command to display the detailed version of the class list.
2. TeaPet displays the class list with all of the students' attributes.

Use case ends.

Extensions

1a. Command is invalid.

1a1. TeaPet shows an error message.

Use case ends.

Use case: UC10 - Display students' academic progress

MSS

1. User enters the command to display academic progress of students.
2. TeaPet displays the academic progress in chronological order.

Use case ends.

Extensions

1a. Command is invalid.

1a1. TeaPet shows an error message.

Use case ends.

Use case: UC11 - Add note for specific student

MSS

1. User enters command, together with a student and note’s content.
2. TeaPet displays feedback that a new note is now tagged to the student specified.
3. TeaPet’s note panel will display the updated list of notes.

Use case ends.

Extensions

1a. Command is invalid.

1a1. TeaPet shows an error message.

1b. Student is invalid.

1b1. TeaPet shows an error message.

Use case ends.

C.1. Use case: UC12 - Edit note of specific student

MSS

1. User enters command, together with an index of the note to edit, and fields to update.
2. TeaPet displays feedback that the specific note chosen has been edited.
3. TeaPet's note panel will display the updated list of notes.

Use case ends.

Extensions

1a. Command is invalid.

1a1. TeaPet shows an error message.

1b. Student is invalid.

1b1. TeaPet shows an error message.

1c. Index is invalid.

1c1. TeaPet shows an error message.

Use case ends.

C.2. Use case: UC13 - Delete note for specific student

MSS

1. User enters command, together with an index of the note to delete.
2. TeaPet displays feedback that the specifed note is deleted.
3. TeaPet's note panel will display the updated list of notes.

Use case ends.

Extensions

1a. Command is invalid.

1a1. TeaPet shows an error message.

1b. Index is invalid.

1b1. TeaPet shows an error message.

Use case ends.

C.3. Use case: UC14 - Add event into schedule

MSS

1. Teacher wishes to add an event into the scheduler
2. Teacher enters the event details.
3. TeaPet saves the item and displays it on the scheduler.

Use case ends.

Extensions

2a. Item is missing details

- 2a1. Teapet displays an error message.
  Use case resumes at step 2.

Use case ends.

C.4. Use case: UC15 - Delete event from schedule

Preconditions 1. Event exists in scheduler.

MSS

1. Teacher lists the events in calendar (UC12)
2. Teacher wishes to delete an event.
3. Teacher confirms the deletion.
4. TeaPet deletes the event from the scheduler.

Use case ends.

Extensions

2a. Teacher reconsiders and chooses not to remove the event.

Use case ends.

C.5. Use case: UC16 - Update student profile picture

Preconditions 1. Png files in image folder is in correct format.

MSS

1. Teacher wants to update image of students.
2. Teacher adds in the respective images into the image folder.
3. TeaPet confirms the process.
4. TeaPet updates the profile pictures of students in the student list.

   Use case ends.

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 500 students without a noticeable sluggishness in performance for typical usage.
3. A teacher with above average typing speed for regular English text (i.e. not code, not system admin commands) should be able to accomplish majority of the tasks faster using commands than using the mouse.
4. TeaPet should be used only for a teacher handling his/her own form class, not by any other teachers.
5. TeaPet should be able to work without internet access.
6. The teacher should be able to familiarise himself/herself within half an hour of usage.

{More to be added}

Appendix E: Glossary

Attributes

The information of a student. For example, phone number, address or next-of-kin contact details.

Class List

Class list of students

CLI

Command Line Interface

GUI

Graphical User Interface

Mainstream OS

Windows, Linux, Unix, OS-X

Private contact detail

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

Schedule

TeaPet's schedule that stores all of the teacher's events t

Appendix F: Product Survey

TeacherKit

Pros:

- Functionality
  - Ease of data tracking
  - Tracks attendance and grades
- Non-funtional requirements
  - Attractive looking GUI
  - Cross platform

Cons:

- Functionality
  - Unable to tag notes to students
  - Unable to track behavioural score
  - Unable to show statistics on exam assessment
- Non-functional requirements
  - Requires internet access
  - Some features are blocked by advertisements and pop ups
  - GUI-reliant, many buttons have to be pressed and many screens traversed to perform a task

Author: Simon Lam

Product information can be found at https://www.teacherkit.net/

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.

G.2. Manual Test: Student particulars

1. Adding a student from class list with the specific name entered by user.
    a. Test case: `student add name/John Tan Jun Wei phone/83391329 email/john@gmail.com temp/36.0 addr/Punggol Street 22 nok/James-father-93259589`
    Expected: Student John Tan Jun Wei has been added to the class list.
    b. Test case: `student add name/John Tan Jun Wei phone/83393129 e/john@gmail.com temp/3@.5 addr/Punggol Street 22 nok/James-father-93259589`
    Expected: No student is added. Error details shown in the status message. Status bar remains the same.
    c. Test case: `student add name/John Tan Jun Wei phone/839 email/john@gmail.com temp/36.0 addr/Punggol Street 22 nok/James-father-93259589`
    Expected: An error message is shown as the phone number of student is invalid.
    d. Test case: `student add name/John Tan Jun Wei phone/83391329 email/john@gmail.com temp/36.0 addr/Punggol Street 22 nok/James-dad-93259589`
    Expected: An error message is shown as the relationship of NOK is not recognised.
    e. Test case: `student add name/John333 phone/83391329 email/john@gmail.com temp/36.0 addr/Punggol Street 22 nok/James-father-93259589`
    Expected: An error message is shown as the name of student cannot contain numbers.
2. Deleting a student from class list with the specific name entered by user.
    a. Test case: `student delete 1`
    Expected: The student at the first index is deleted from the list. Status message displays that the specified student has been deleted.
    b. Test case: `delete Tan John Wei Jun`
    Expected: No student is deleted. Error details shown in the status message. Status bar remains the same.
    c. Other incorrect delete commands to try: `delete`, `delete 10` (where the specified student is not a student in the class list due to the index being out of bounds.) *{give more}*
    Expected: Similar to test case b.

G.3. Manual Test: Class Admin Particulars

1. Saving a date
    a. Test case: `admin save`
    Expected: Date has either been saved if today's date is not in the list of dates, or date not saved if today's date is already in the list of dates. Status message either displays that today's date has been saved or displays that current date already exists in the current list of dates respectively.
    b. Other incorrect delete commands to try: `save`, `admin save DATE` where `DATE` is in YYYY-MM-DD format.
    Expected: Status bar displays invalid command and incorrect command format message respectively.
2. Deleting a date
    a. Test case: `admin delete DATE` where DATE is in YYYY-MM-DD format.
    Expected: Date has either been deleted if the date provided exists in the list of dates, or no dates will be deleted if the date provided is not in the list. Status message either displays that the specific date has been deleted or displays that current date already exists in the current list of dates respectively.
    b. Other incorrect delete commands to try: `admin delete`, `delete DATE`, `admin delete `DATE`, where `DATE` is in YYYY-MM-DD format and `` `DATE `` is in the wrong `DATE` format.
    Expected: Status bar displays invalid command and incorrect command format message.

G.4. Manual Test: Scheduler

1. Adding an event to the scheduler
    a. Prerequisites: The scheduler already contain an event with name "Coffee Break", startDateTime "2020-04-04T12:00", endDateTime "2020-04-04T13:00". The recurrence type and color do not matter as long as they are valid.
    b. Test case: `schedule add eventName/Consultation startDateTime/2020-04-10T10:00 endDateTime/2020-04-10T12:00 recur/none color/5`
    Expected: An event with name Consultation is added to the scheduler
    c. Test case: `schedule add eventName/Coffee Break startDateTime/2020-04-04T12:00 endDateTime/2020-04-04T13:00 recur/none color/5`
    Expected: An error message is shown due to duplicate events being created.
    d. Test case: `schedule add eventName/ `
    Expected: An error message is shown due to invalid command.
    e. Test case: `schedule add eventName/Consultation startDateTime/2020-04-10T10:00 endDateTime/2020-04-10T12:00 recur/none color/24`
    Expected: An error message is shown due to an invalid color code.
    f. Test case: `schedule add eventName/Consultation startDateTime/2020-04-10T10:00 endDateTime/2020-04-10T12:00 recur/fortnightly color/5` Expected: An error message is shown due to invalid recurrence type.

g. Test case: `schedule add eventName/Consultation startDateTime/2020-04-10T13:00 endDateTime/2020-04-10T12:00 recur/none color/5` Expected: An error message is shown due to the invalid date time range.

G.5. Notes

NOTE	In order to add or edit a specific note belonging to a student, the student must first be in the class-list. In order for optimal manual testing, please create the student first before creating a note which belongs to him/her. You can refer to Section 7.1.1, "Student Add command" to find out more on how to add a student to the class-list.

TIP	Delete all previous notes. Be sure two students named **Freddy Zhang** and **Lee Hui Ting** are in the class-list. Conduct these tests sequentially from first to last, for the most effective testing experience.

1. Adding a note to the notes panel.
   a. Test case: `notes add name/Freddy Zhang cont/Reminder to print his testimonial pr/HIGH`
   Expected: New note added to the notes panel. This note should be red (high priority) in colour, belonging to Freddy Zhang. The timestamp in the note should be the current date and time.
   b. Test case: `notes add name/Freddy Zhang cont/Freddy was late for class for the second day in a row. pr/LOW`
   Expected: New note added to the notes panel. This note should be yellow (low priority) in colour, belonging to Freddy Zhang. The timestamp in the note should be the current date and time.
   c. Test case: `notes add name/Lee Hui Ting cont/Hui Ting left school for an interview. pr/MEDIUM`
   Expected: New note added to the notes panel. This note should be orange (medium priority) in colour, belonging to Lee Hui Ting. The timestamp in the note should be the current date and time.
   d. Test case: `notes add name/Freddy_Zhang_ cont/He exemplified an improvement in behaviour pr/LOW`
   Expected: An error message is shown as the student name should be alpha-numeric.
   e. Test case: `notes add name/Freddy Zhang cont/He exemplified an improvement in behaviour pr/Not High`
   Expected: An error message is shown as priority indicated must be either **LOW**, **MEDIUM** or **HIGH**, case-insensitive.
2. Editing a note in the notes panel.
   a. Test case: `notes edit 1 cont/Reminder to print his testimonial and gradebook`
   Expected: First note in the panel is updated with new content. Only the content should be modified, and nothing else.
   b. Test case: `notes edit 1 name/Lee Hui Ting cont/Reminder to print her testimonial and gradebook`
   Expected: First note in the panel is updated with new name and content. Note should reflect

student Lee Hui Ting's name, with modfied content. Nothing else should be different.

c. Test case: `notes edit 1 pr/MEDIUM`
Expected: First note in the panel is updated new priority. Note colour should change from red to orange.

3. Deleting a note in the notes panel.

a. Test case: `notes delete 1`
Expected: First note in the panel should be removed.

b. Test case: `notes delete 0`
Expected: An error message is shown as the integer provided must be greater than zero.

c. Test case: `notes delete 4`
Expected: An error message is shown as the integer provided as the number of notes in the notes panel is less than 4. Integer provided is out of range.

4. Filtering the list of notes to search for specific notes.

a. Test case: `notes filter Freddy`
Expected: Only notes containing the word Freddy should be displayed in the notes panel.

b. Test case: `notes filter low`
Expected: Only notes containing the word low should be displayed in the notes panel.

5. Exporting notes in the notes panel into a .csv file.

a. Test case: `notes export`
Expected: A file named studentNotes.csv should be created in the data folder of the same directory in which TeaPet is installed.

b. Test case: `notes export 3`
Expected: An error message is shown as there should be no arguments passed into notes export command.