ExerHealth - Developer Guide

By: `Team ExerHealth` Since: `Sep 2019` Licence: `MIT`

1. Setting up

Refer to the guide here.

2. Design

2.1. Architecture

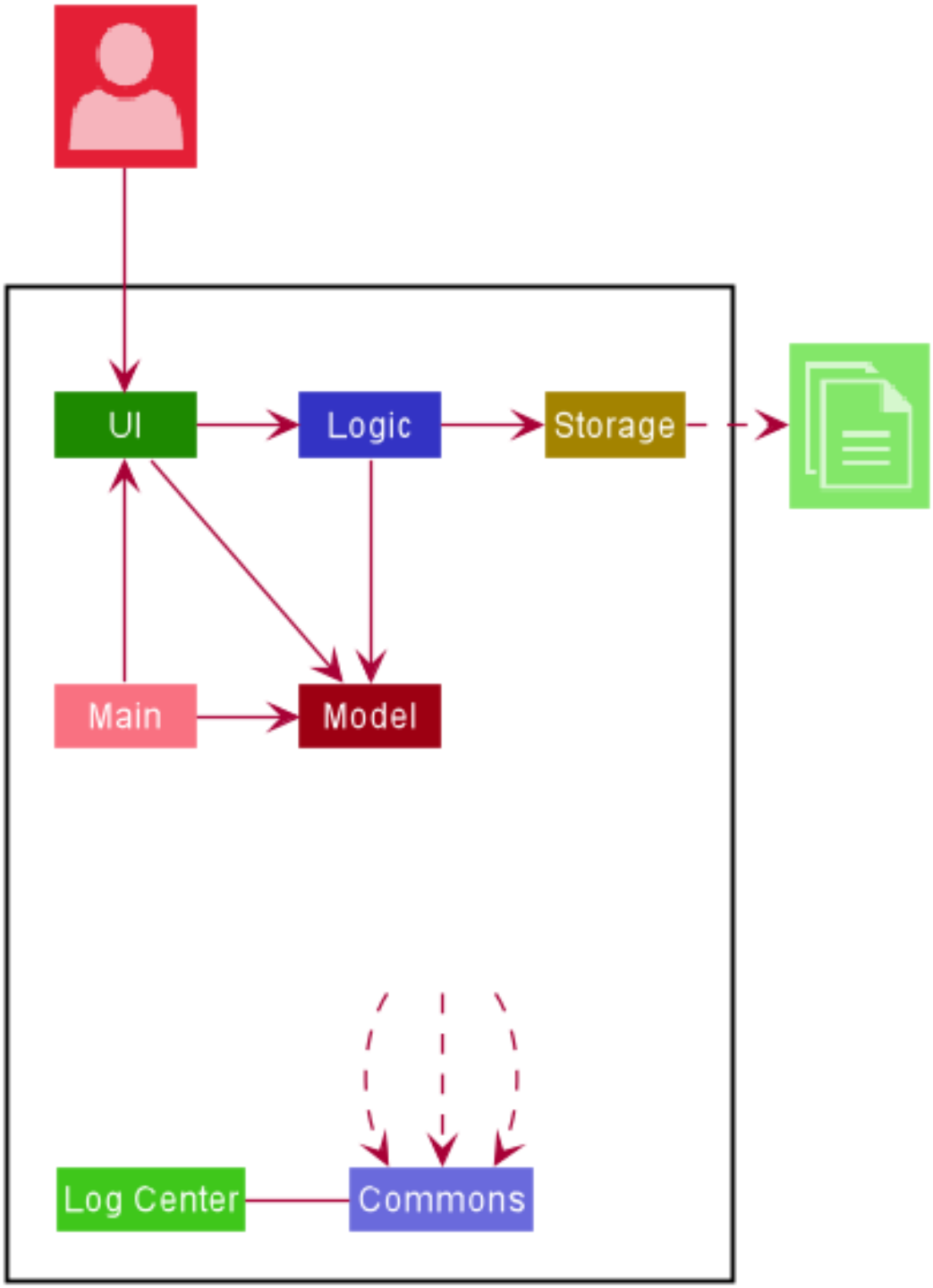

Figure 1. Architecture Diagram

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

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

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

- At app launch: Initializes the components in the correct sequence, and connects them up with each other.

- At shut down: Shuts down the components and invokes cleanup method where necessary.

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

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

The rest of the App consists of 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.

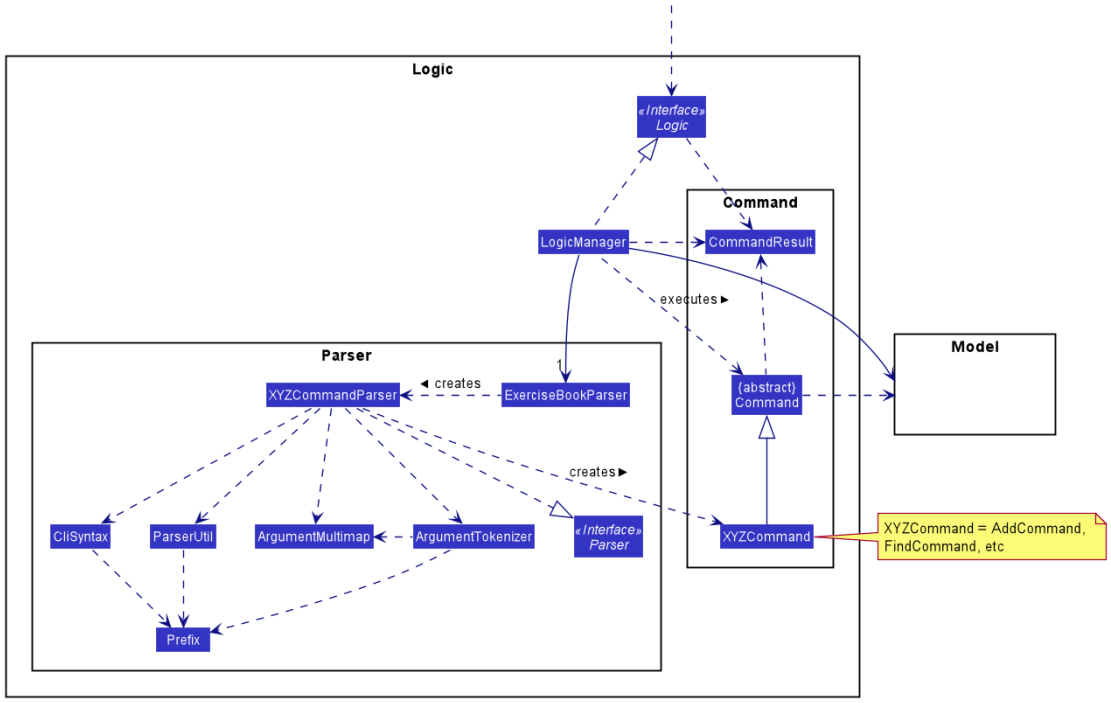

Figure 2. Class Diagram of the Logic Component

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 `delete t/exercise i/1`.

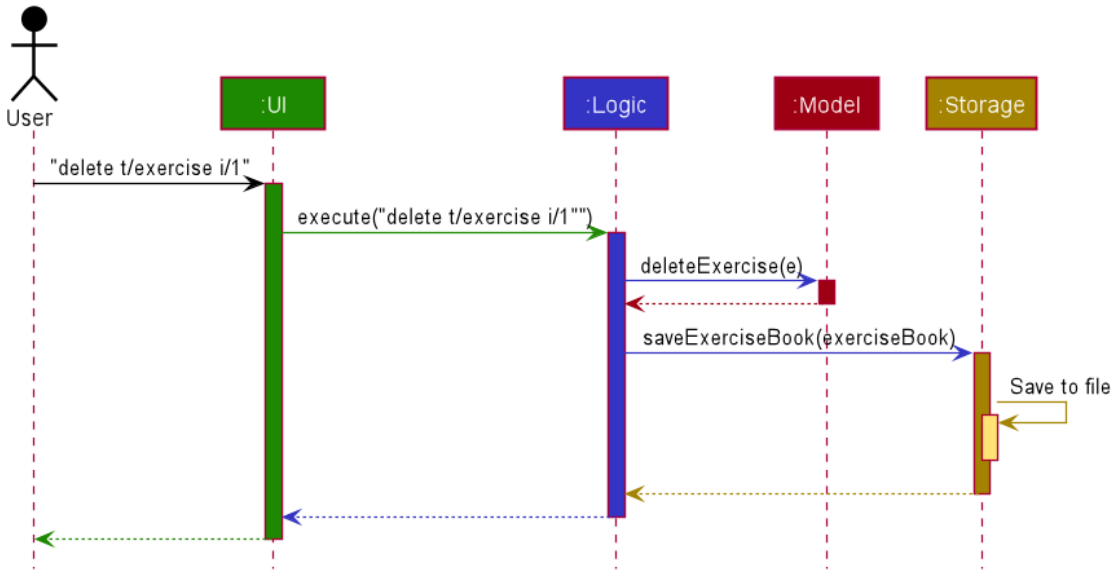

Figure 3. Component interactions for `delete t/exercise i/1` command

The sections below give more details of each component.

2.2. UI component

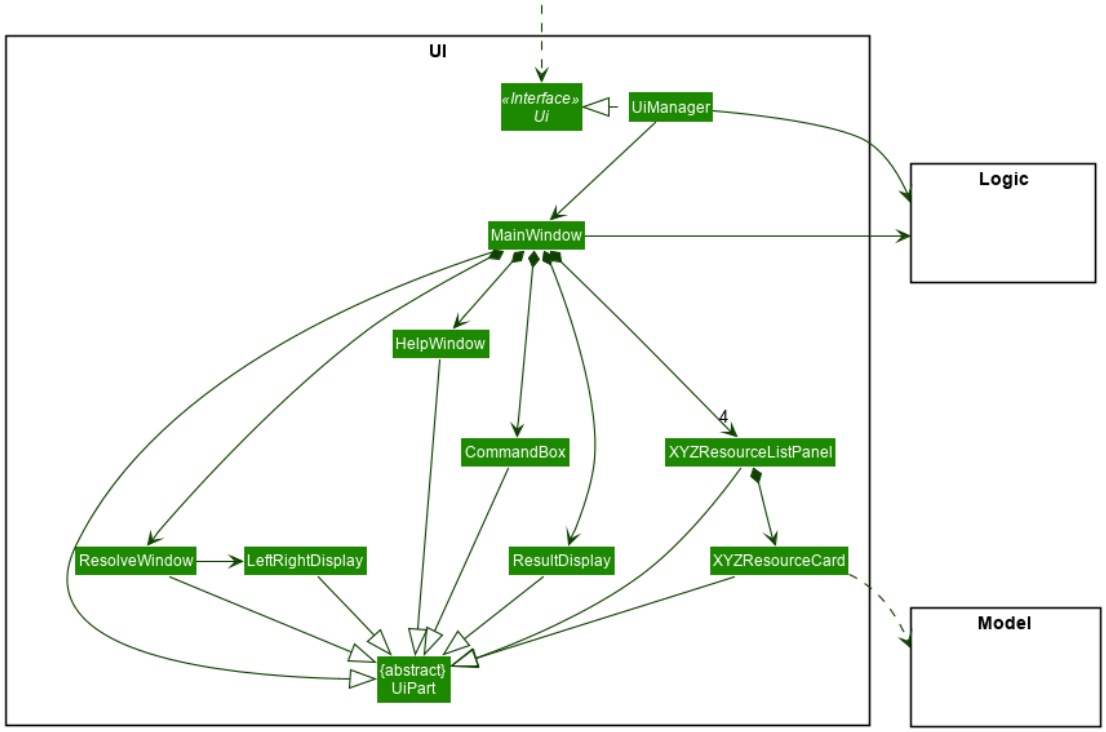

Figure 4. Structure of the UI Component

API : `Ui.java`

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

The `UI` component uses JavaFx UI framework. The layout of these UI parts are defined in matching

`.fxml` files that are in the `src/main/resources/view` folder. For example, the layout of the `MainWindow` is specified in `MainWindow.fxml`

The `UI` component,

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

2.3. Logic component

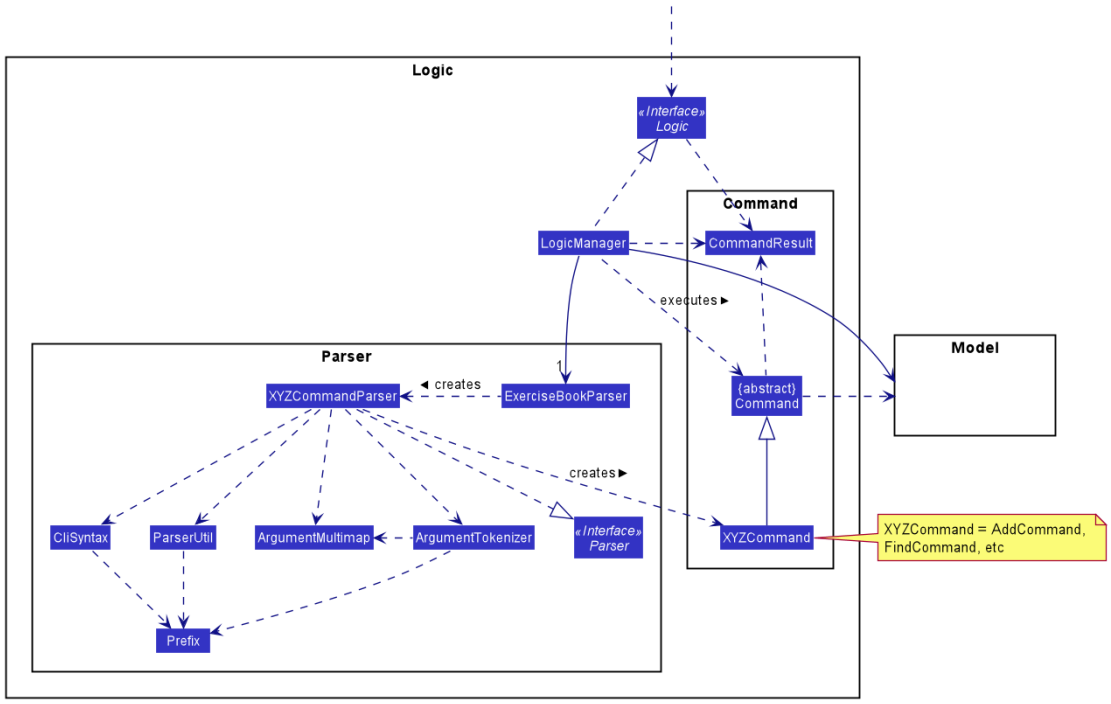

Figure 5. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `ExerciseBookParser` 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 an exercise/regime).
4. The result of the command execution is encapsulated as a `CommandResult` object which is passed back to the `Ui`.
5. In addition, the `CommandResult` object can also instruct the `Ui` to perform certain actions, such as displaying help to the user.

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

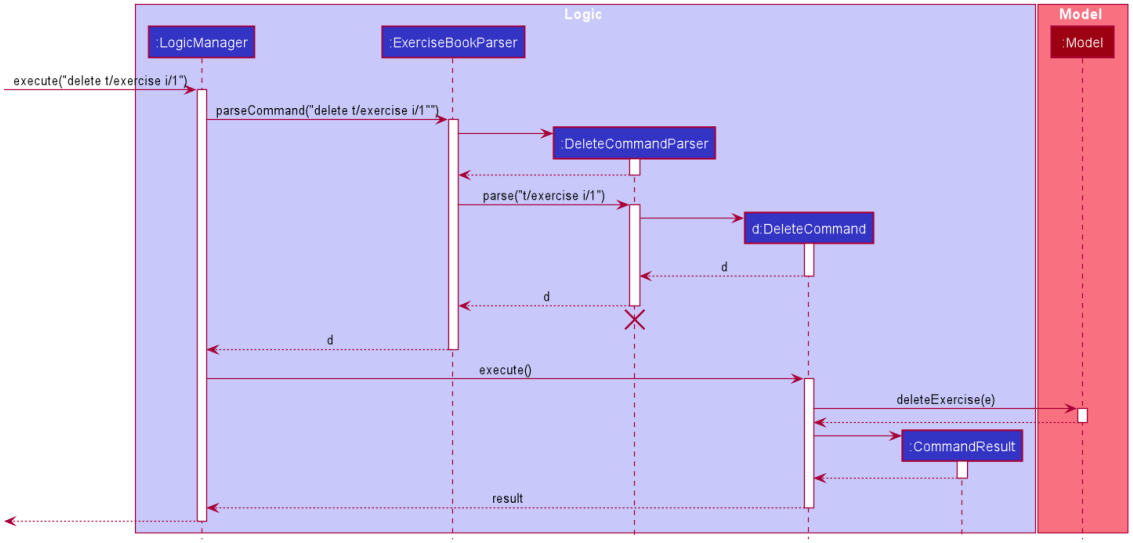

Figure 6. Interactions Inside the Logic Component for the `delete 1` Command

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

2.4. Model component

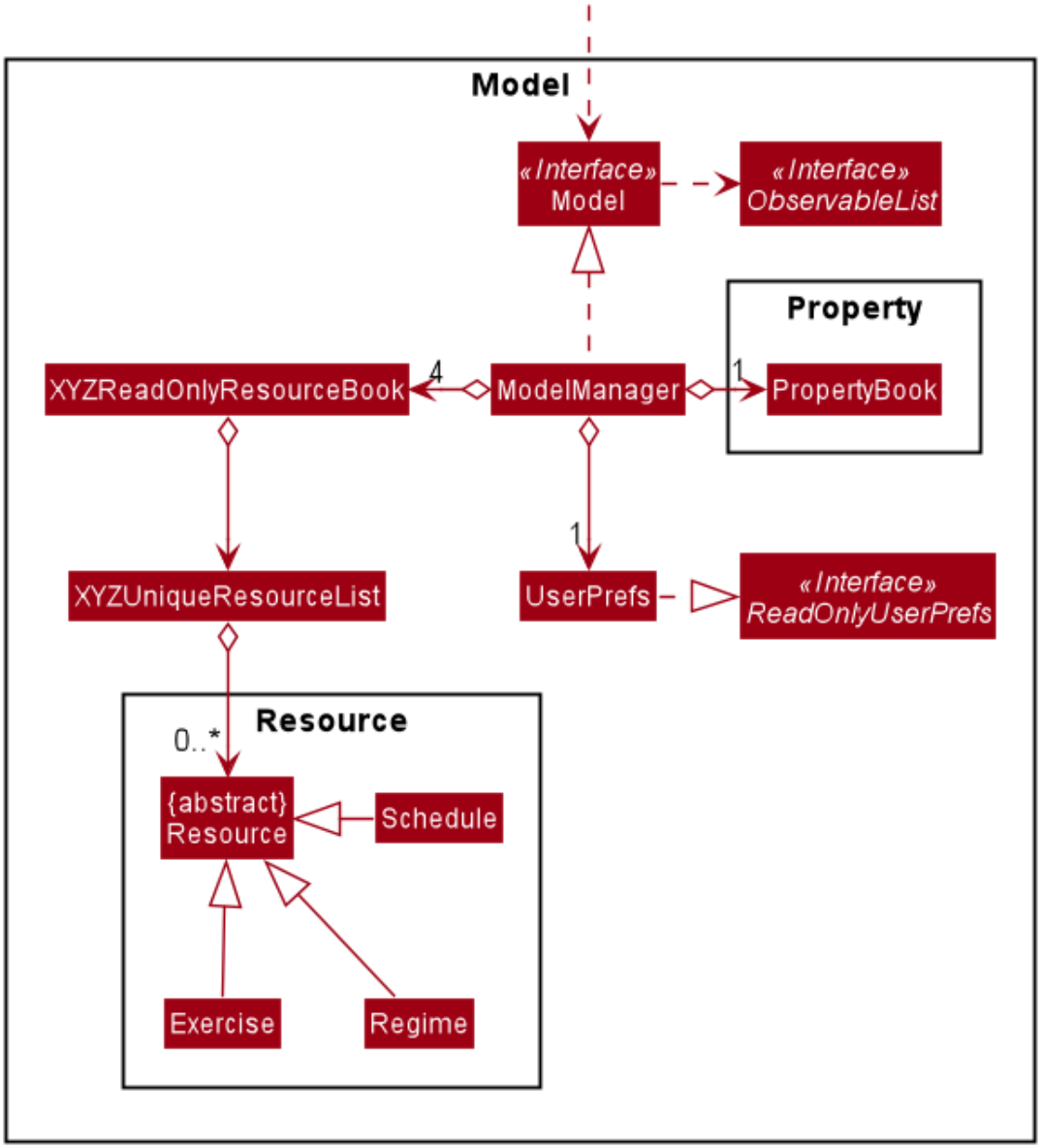

Figure 7. Structure of the Model Component

API : `Model.java`

The `Model`

- stores a `UserPref` object that represents the user's preferences.
- stores a `PropertyBook` object that represents the custom properties defined by the user.
- stores a `ExerciseBook` object that represents the user's exercises being tracked.
- stores a `ExerciseDatabaseBook` object that represents the database of exercises in ExerHealth.
- stores a `RegimeBook` object that represents the user's regimes.
- stores a `ScheduleBook` object that represents the user's schedules.
- exposes an unmodifiable `ObservableList<Exercise>` that can be 'observed' e.g. the UI can be bound to this list so that the UI automatically updates when the data in the list change.
- does not depend on any of the other three components.

2.5. Storage component

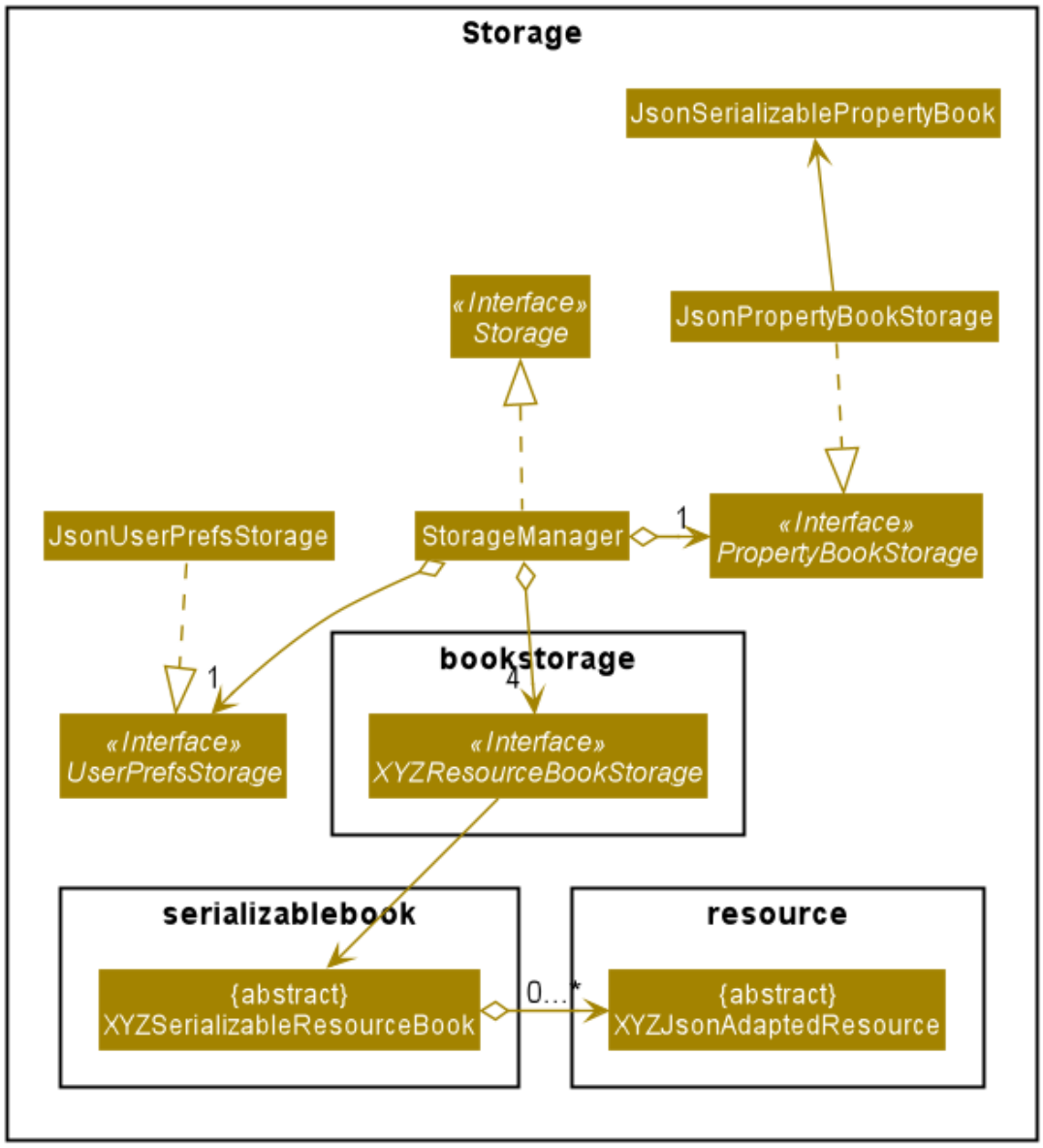

Figure 8. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

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

NOTE	Resource Book data consists of Exercise Book, Regime Book and Schedule Book data

2.6. Common classes

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

3. Implementation

This section describes some noteworthy details on how certain features are implemented. === Undo/Redo feature

3.1. Implementation

The undo/redo mechanism is facilitated by the `events` package consisting of `EventHistory`, `EventFactory`, `EventPayload` and the various `Event` classes.

The `EventHistory` is a singleton class used to store a history of successfully executed commands as `Event` objects. Instances of `Event` are stored in either the `undoStack` or the `redoStack` depending on the user's course of action.

The `EventHistory` class has two primary methods namely `undo(Model model)` and `redo(Model model)`:

- `eventHistory.undo(model)` — Undoes the `Event` at the top of the `undoStack`, executes it, and pushes it to the top of the `redoStack`
- `eventHistory.redo(model)` — Redoes the `Event` at the top of the `redoStack`, executes it, and pushes it to the top of the `undoStack`

These operations are utilised in the `UndoCommand` and `RedoCommand` respectively.

The following steps will describe the steps taken in the execution of an UndoableCommand, and subsequently the UndoCommand and RedoCommand.

Step 1: When an `UndoableCommand` is executed, key information used during the command will be added into a newly initialized `EventPayload`.

NOTE	The `EventPayload` is a wrapper class to store key information about the particular command. For instance, if an `EditCommand` has been executed, the `EventPayload` will store the `originalExercise` as well as the `editedExercise`.

Step 2: The `EventFactory` takes in the `UndoableCommand` and generates an `Event` using the `EventPayload` stored in the `UndoableCommand`. The `Event` is then added to the undo stack of the `EventHistory`.

NOTE	The `EventFactory` checks for the command word of the `UndoableCommand` to decide which specific `Event` object to generate. It will then obtain the `EventPayload` from the `UndoableCommand` and pass it into the constructor of the `Event` so that the `Event` captures the key information of the `UndoableCommand`.

Step 3: To undo the latest `UndoableCommand` the user executes the `UndoCommand` by entering `undo` into the command box.

Step 4: The `UndoCommand` executes `eventHistory.undo(model)`, which prompts the `EventHistory` instance to pop the next `Event` to undo from the undo stack. Once the `Event` is undone, it will be pushed to the top of the redo stack.

Step 5: To redo the command that has been undone, the user executes the `RedoCommand`. This execution behaves similarly to step 4, except that the next `Event` is taken from the top of the redo stack and pushed to the undo stack instead.

NOTE	In steps 4 and 5, if any of the respective stack is empty when undo or redo is called, a `CommandException` will be thrown and an error message will be displayed to indicate there is no undoable or redoable commands.

The following two Sequence Diagrams show a sample flow of the execution when an `EditCommand`, which is an `UndoableCommand`, has been executed and subsequently undone.

The first diagram describes the process of storing an `EditEvent` to `EventHistory` during the execution of the `EditCommand`. The `EventPayload` is only initialized when the `EditCommand` is executed. The `EventPayload` is subsequently used for the initialization of the `EditEvent`.

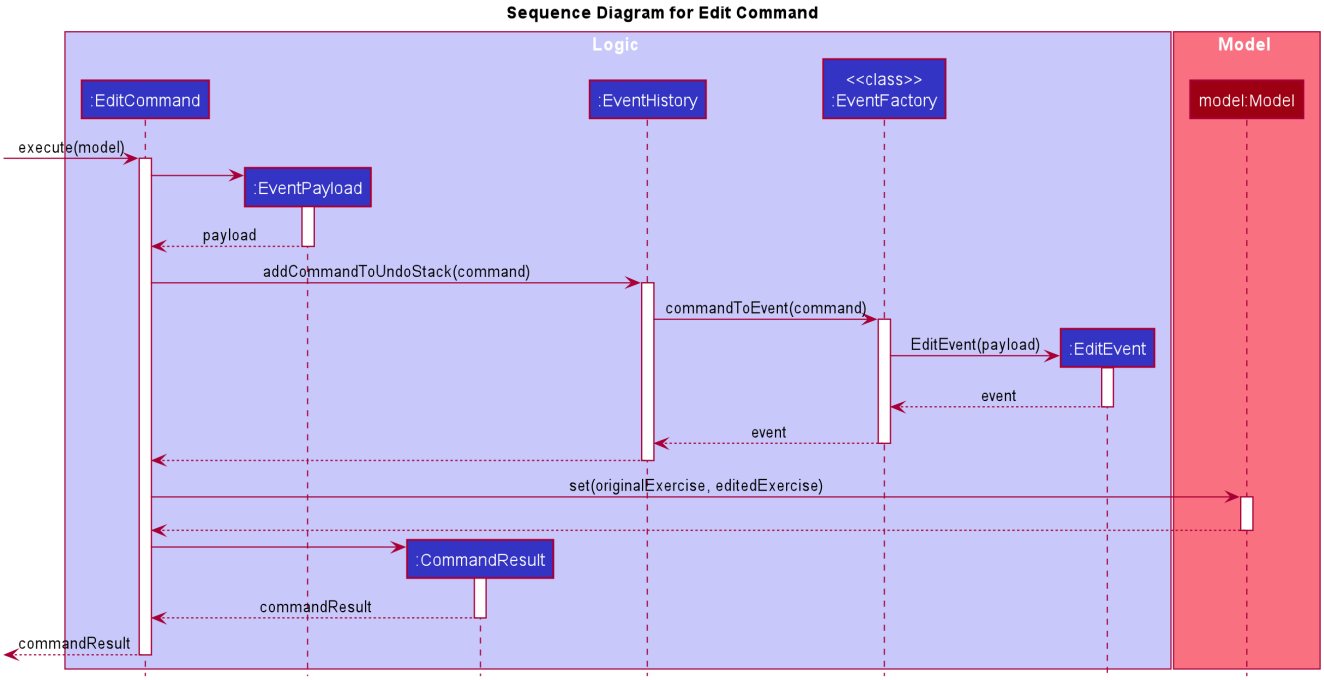

Figure 9. The process of storing an EditEvent to EventHistory

The second diagram here describes the process of undoing the `EditCommand` executed above using the `UndoCommand`. When the `UndoCommand` is executed, the `EventHistory` calls the `undo` method of the next `Event` in the undo stack (i.e. the `EditEvent`).

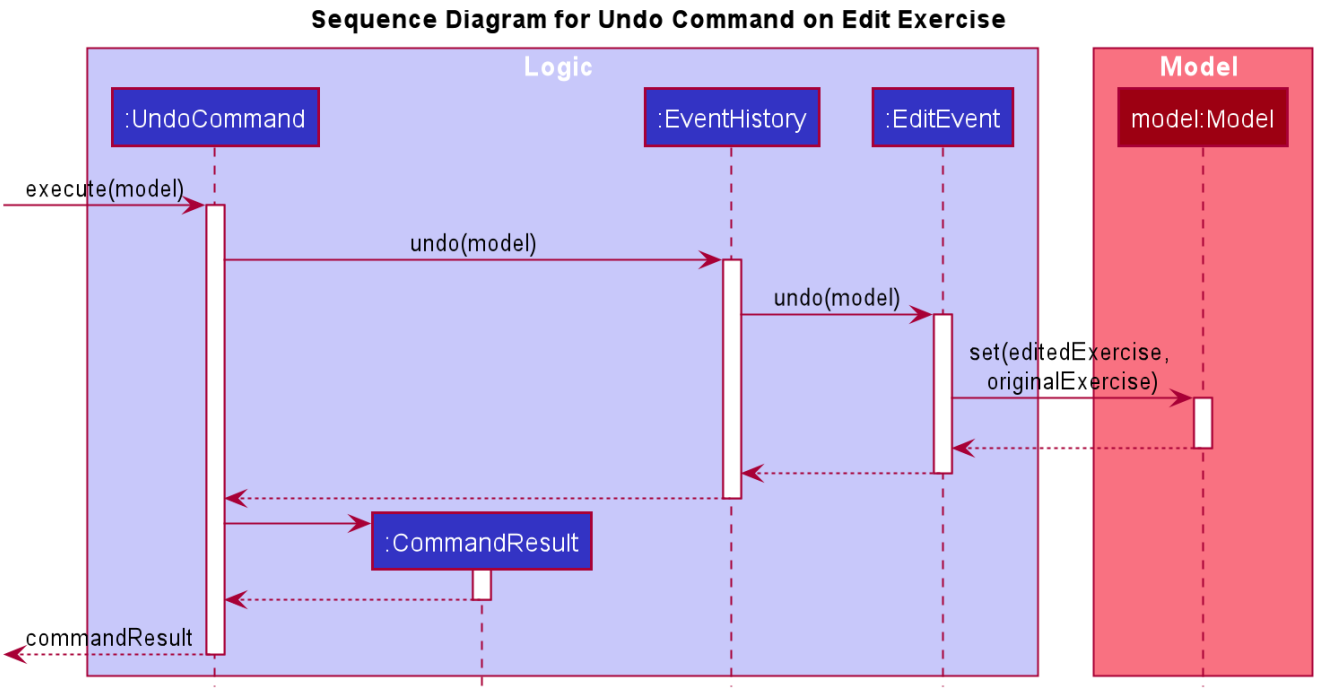

Figure 10. The process of undoing the EditCommand

Given below is a Class Diagram to show the associations between Event, Command and Model. It is specifically designed such that only objects that implement the `Event` and `Command` interface will need to handle the `model` class.

NOTE

The only commands that implements the `UndoableCommand` are `AddCommand`, `DeleteCommand`, `EditCommand`, `ClearCommand`, `ScheduleCommand` and `ResolveCommand`. They each stores an `EventPayload` instance.

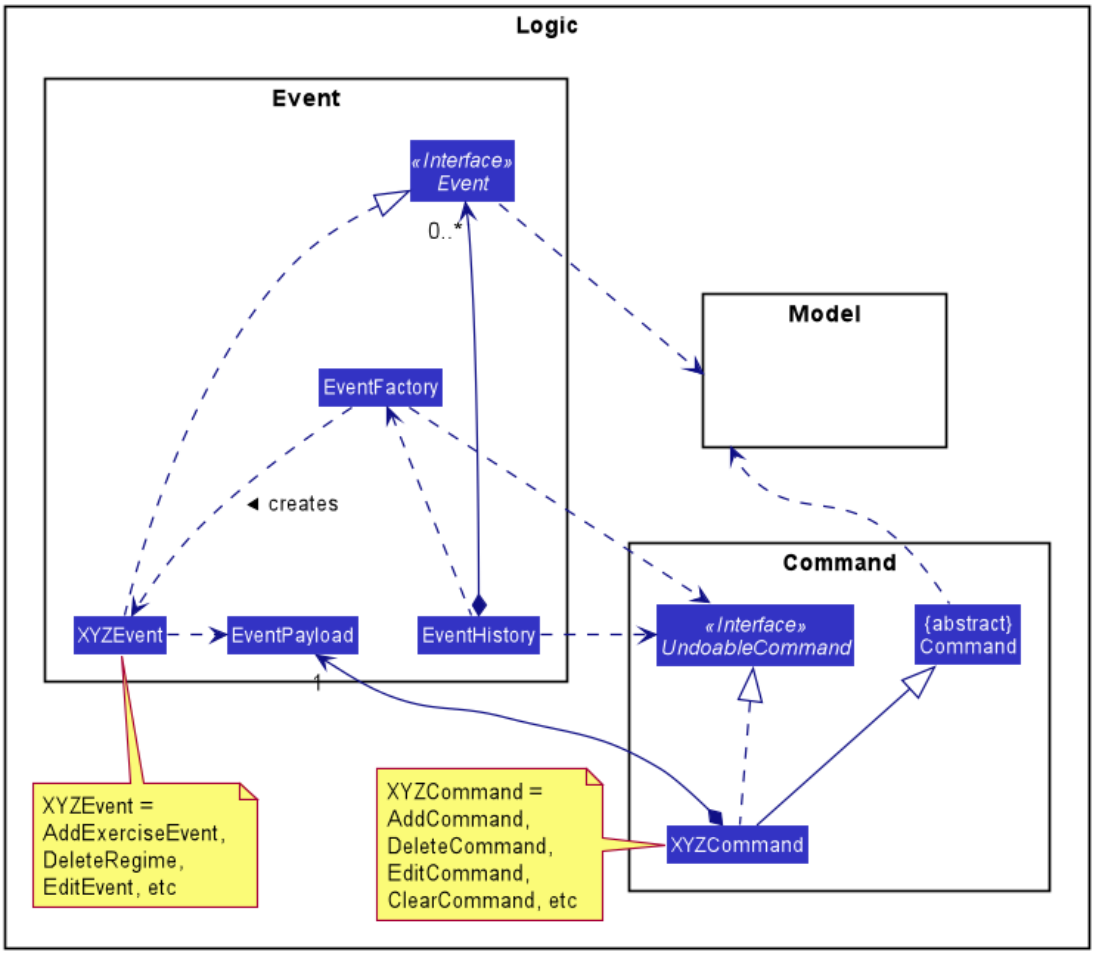

Figure 11. The associations between Event, Command and Model

The following Activity Diagram summarizes what happens when a user enters undoable commands, the undo command and the redo command.

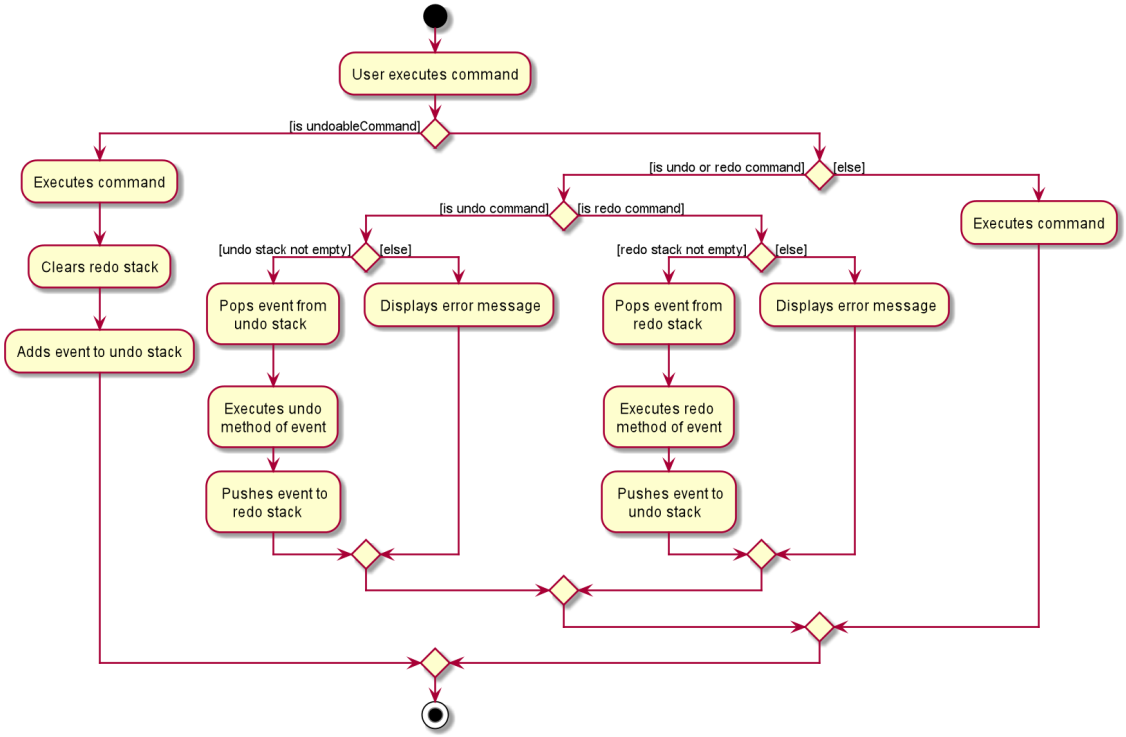

Figure 12. The workflow when a user enters an undoable command and performs undo/redo

3.2. Design Considerations

Aspect: How undo & redo executes

- **Choice 1: (current choice)** Implements undo and redo of each Command in a separate Event object stored in the EventHistory
  - Pros:
    - Uses less memory to store Event objects and payloads as compared to entire copies of the Model object.
    - Open for extensions and close to modifications as the Event interface only contains undo and redo methods, and can be easily implemented when new Undoable commands are introduced.
  - Cons:
    - UndoableCommand objects are forced to depend on EventPayloads when it does not actually use it directly. (e.g. `DeleteCommand` has to store the exercise being deleted despite using it only once).
- **Choice 2:** Individual command knows how to undo/redo by itself.
  - Pros:
    - Uses less memory to store each command as compared to entire copies of the Model object.
  - Cons:
    - Violates Single Responsibility Principle as Commands need to contain specific implementation of the inverse action of itself and also stores data such as the exercise being deleted in a local field.
- **Choice 3:** Saves the entire model consisting of the exercise, regime, schedule and suggestion lists.
  - Pros:
    - Easy to implement.
  - Cons:
    - May have performance issues in terms of memory usage as multiple lists need to be stored (i.e. Exercise list, Regime list, Schedule list)
    - Unnecessary storage of irrelevant details such as suggestion list.

Aspect: Data structure to support the undo/redo commands

- **Choice 1 (current choice):** Use a singleton EventHistory to store stacks of Events generated by a EventFactory.
  - Pros:
    - Ensures only one instance of EventHistory exists
    - The EventFactory relies on the Factory pattern that helps to reduce coupling between EventHistory and each individual Event.

    - Cons:
        - The Singleton pattern may have a chance of breaking if multiple threads initialize the singleton class at the same time, creating multiple instances of EventHistory. However, if this problem arises, the instantiation method can be made "synchronized" to circumvent this issue.
- **Choice 2:** Use a list to store the history of model objects.
    - Pros:
        - Very simple to implement as each step simply requires a deep copy of the model to be created and stored.
    - Cons:
        - Difficult to monitor multiple resource books (e.g. Regime books and Exercise books) as they all manage different types of resources that can be altered by commands. ==== Design Patterns

The Undo/Redo feature implementation is based on the Singleton, Command, and Factory design patterns

- **Singleton**
    - To help ensure that only one instance of `EventHistory` exists during the execution of the program
    - Allows easier access by the various command classes (i.e. the UndoableCommands, UndoCommand and RedoCommand)
- **Command**
    - Extensions of new `Event` is easy and can be done without significant changes to the existing code
- **Factory**
    - Suitable for the context of taking in a particular Command and returning a corresponding Event
    - Reduces coupling between Command classes and Event classes

3.3. Resolve feature

3.3.1. Rationale

There are multiple times where if the user wishes to schedule a regime, they find themselves in trouble over which kind of exercise regime they can fit into their schedule. The motivation behind this feature is so that users can customise their own schedules to their own liking. The alternative of an auto scheduler will restrict users from having the regime of their liking be scheduled. Instead of forcing users to adhere to some pre-generated resolution, we allow the users to make their own choice and choose their own exercise regime to be scheduled.

3.3.2. Implementation

The resolve feature is used when there is a scheduling conflict that happens within ExerHealth. This feature will alter the state of the program. The state is known by `MainApp` and it is either `State.IN_CONFLICT` or `State.NORMAL`. Only when the state is `State.IN_CONFLICT` will `resolve` commands be allowed.

For the implementation of the resolve feature, the `ResolveCommand` will hold a `Conflict` object which is then passed into `Model`. The concrete implementation, `ModelManager` then resolves the conflict that is being held there. Each `Conflict` object will hold 1 conflicting `schedule` and 1 `schedule` that was originally scheduled on the date.

Shown below is the class diagram for the implementation of the `Resolve` feature.

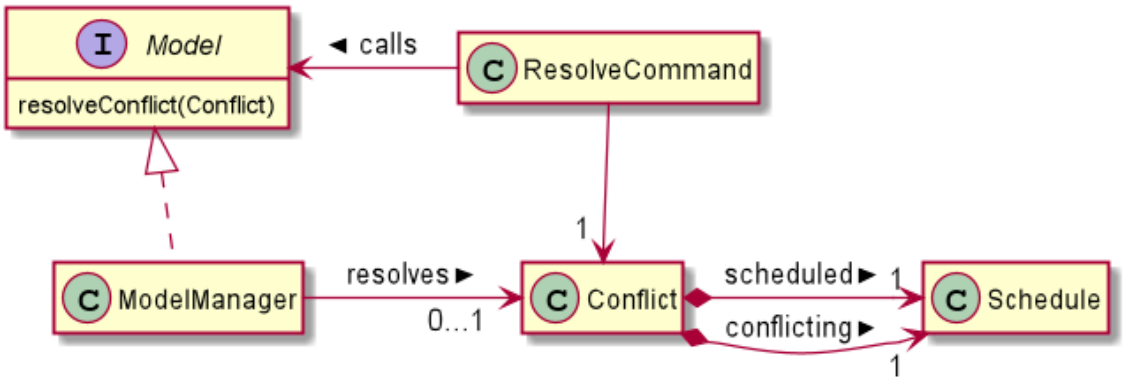

Figure 13. Class diagram for Resolve Command

With regards to the flow of the program for a scheduling conflict, the steps are laid out below:

Step 1. User enters a `schedule` command that will cause a scheduling conflict. The `ScheduleCommand` will change `MainApp` state to `State.IN_CONFLICT`.

NOTE	`schedule` can conflict with another `schedule` when the dates from the 2 schedules are the same. The method `model.hasSchedule()` returns `true` if that happens.

Step 2. A `CommandResult` object is returned to `MainWindow` where the flag `showResolve` is set to `true`.

Step 3. Upon receipt of the object, `MainWindow` will show the resolve window and the user is required to resolve the conflict.

NOTE	The `ResolveWindow` will block all inputs to `MainWindow` and only allow `resolve` command to be entered.

Shown below is the sequence diagram for when a scheduling conflict happens:

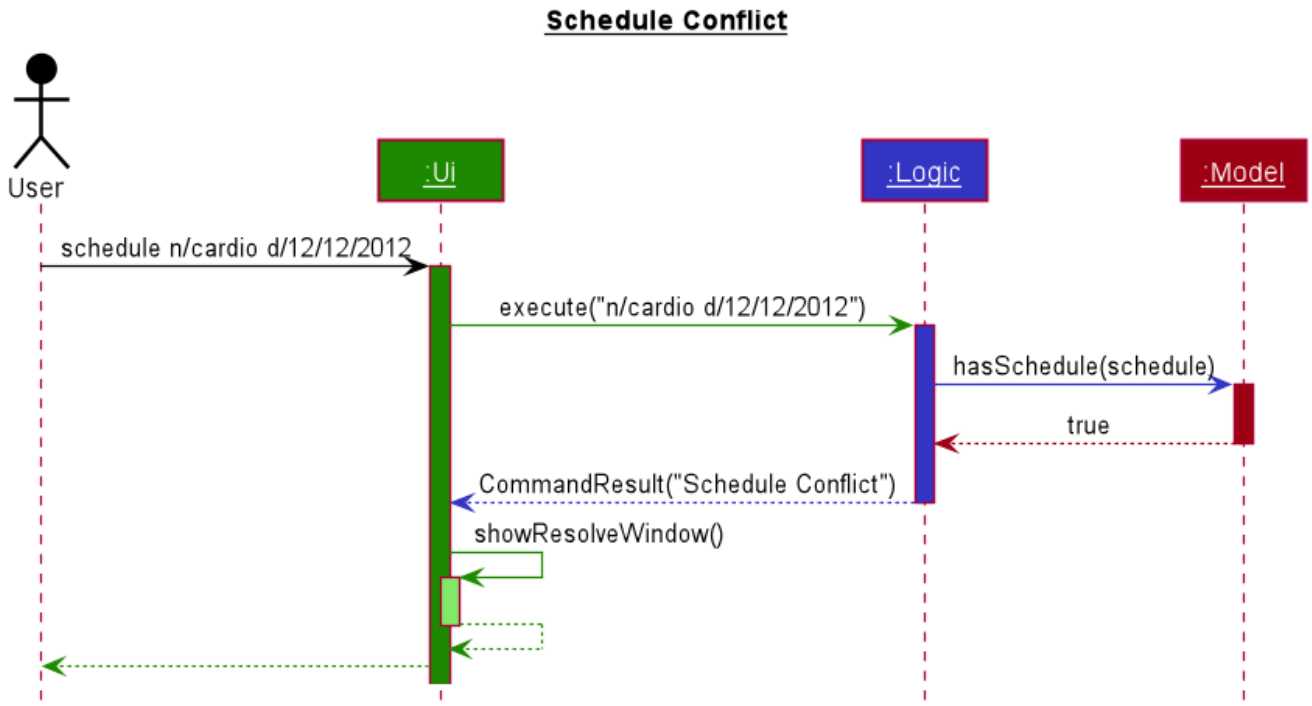

Figure 14. Sequence diagram when a scheduling conflict happens

Step 5. When the user is prompted with the `ResolveWindow`, all the conflicting exercises will be shown in one page. The previously `scheduled regime` on the left and the `conflicting regime` on the right.

Step 6. Once the user issue a `resolve` command correctly, the `model` and `storage` of ExerHealth will be updated to reflect the changes. A new regime will be added for the user from the `resolve`.

NOTE	The `ResolveWindow` will only take one valid `resolve` command and `Ui` will close the `ResolveWindow` immediately after the command finishes. The newly made schedule will result in a new `regime` being added to the user's `RegimeList`, so the name of the `regime` in the `resolve` command cannot have any conflicts with current names in `RegimeList`.

Step 7. The `ResolveWindow` then closes upon successful `resolve` and the application continues.

The following activity diagram summarizes what happens when a user enters a `schedule` command:

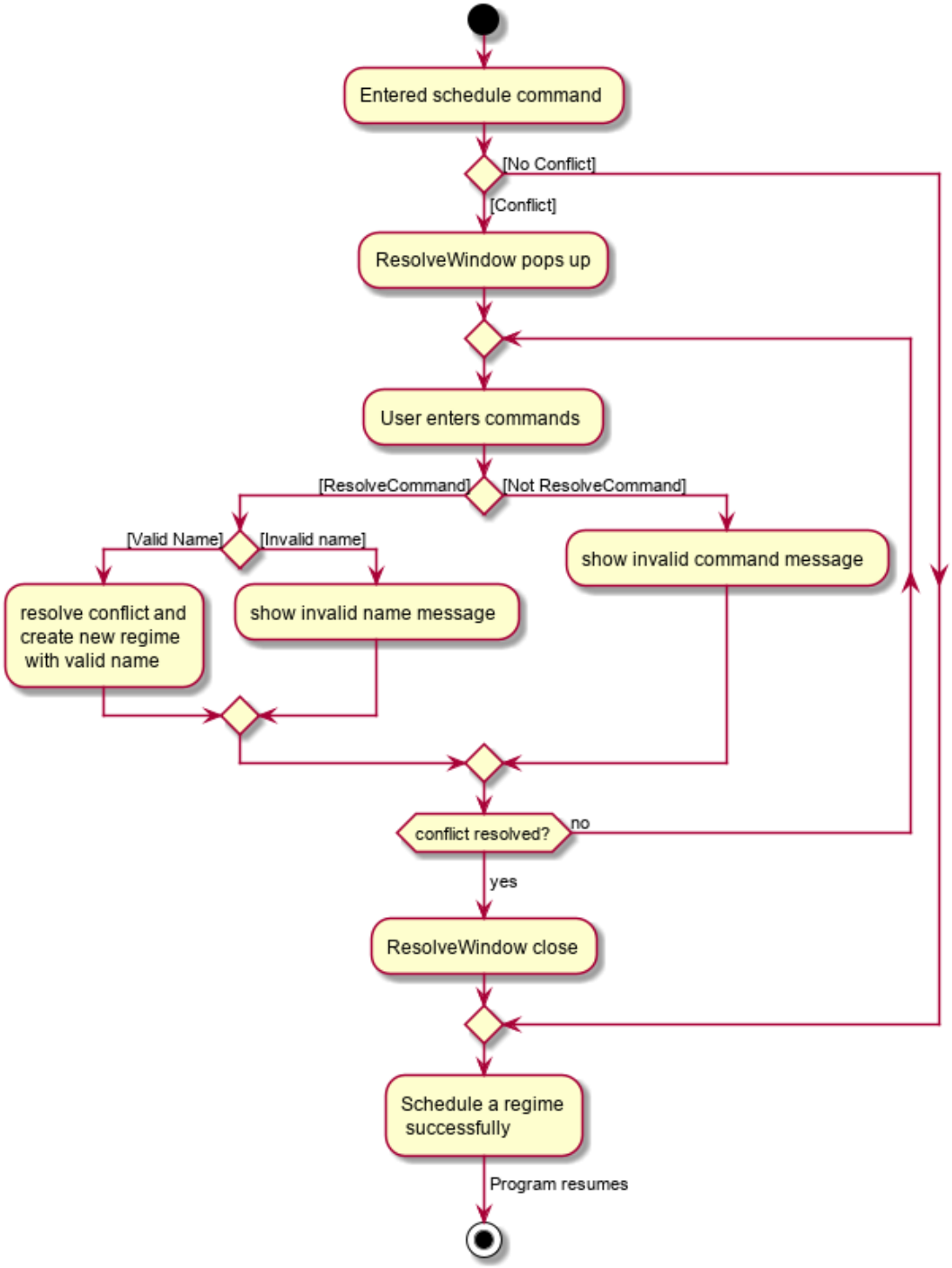

Figure 15. Activity diagram for when a user enters a `schedule` command

3.3.3. Design Considerations

Aspect: Signalling Schedule Conflict

- **Choice 1 (current choice):** Using `CommandResult` object
  - Pros:
    - Makes use of existing objects in codebase making it easier to implement
    - UI does not have to handle logic when encountering schedule conflicts. It only has to show the `ResolveWindow` and pass the data accordingly.

- Cons:
  - If we have to signal different types of outcomes to the UI, the `CommandResult` class will become bloated.

- **Choice 2:** throw `ScheduleException`
  - Pros:
    - Easy to implement. `ScheduleCommand` just has to throw an exception and `UI` catches it.
  - Cons:
    - `UI`'s execute methods will contain multiple `try/catch` which acts like a control flow mechanism which increases code smell.
    - If there is a need to pass down information from executed Commands, an exception is unable to convey any sort of complex information that the `UI` can act on. Thus, encapsulating information in an object will be more open to extension compared to throwing an exception.

3.4. Custom feature

3.4.1. Rationale

A quick conversation with a few of our friends revealed that there are many properties which they intend to keep track for exercises. However, it is unlikely that we can implement all of these properties for the exercises as there may be too much overhead and we can never be certain that we have met all of the users' needs.

3.4.2. Overview

This feature is facilitated by both `PropertyBook` and `CustomProperty`. Whenever a user adds a newly defined custom property, a `CustomProperty` object will be created which is stored in `PropertyBook`. Its corresponding prefix and full name will be tracked by `PropertyBook` to avoid clashes in their uses.

3.4.3. Current Implementation

`CustomProperty` encapsulates a single custom property that the user defines. It contains information such as name, prefix and parameter type of the custom property. The parameter type is supported by an enumeration class `ParameterType` and is restricted to one of the following 3 types: `Number`, `Text`, `Date`.

`PropertyBook` serves as a singleton class that helps to manage all of the custom properties that have been defined by the user. This class acts as an access point for any information relating to the creation or deletion of custom properties.

To keep track of the custom properties and its relevant information, the following are used:

1. `customProperties`: A set containing all of the `CustomProperty` objects that have been created.
2. `customPrefixes`: A set containing all of the `Prefix` objects associated with existing custom properties.

3. `customFullNames`: A set containing the full names of the existing custom properties.
4. `defaultPrefixes`: A set containing all of the `Prefix` objects associated with default properties and parameter types.
5. `defaultFullNames`: A set containing all of the full names of default properties.

Custom names and prefixes are separated from its default counterparts to ensure that the default names and prefixes will always be present when the `PropertyBook` is first initialised.

To help facilitate `PropertyBook` in its custom properties management, the following main methods are implemented:

1. `PropertyBook#isPrefixUsed(Prefix)`: Checks if the given prefix has been used by a default or custom property.
2. `PropertyBook#isFullNameUsed(String)`: Checks if the given name has been used by a default or custom property.
3. `PropertyBook#isFullNameUsedByCustomProperty(String)`: Checks if the given name has been used by a custom property
4. `PropertyBook#addCustomProperty(CustomProperty)`: Adds the new custom property. Each time a custom property is added, the prefix set in `CliSyntax` is also updated.
5. `PropertyBook#removeCustomProperty(CustomProperty)`: Removes a pre-defined custom property. Its associated prefix is also removed from the prefix set in `CliSyntax`.

All of the crucial associations mentioned above are summarised in the next class diagram.

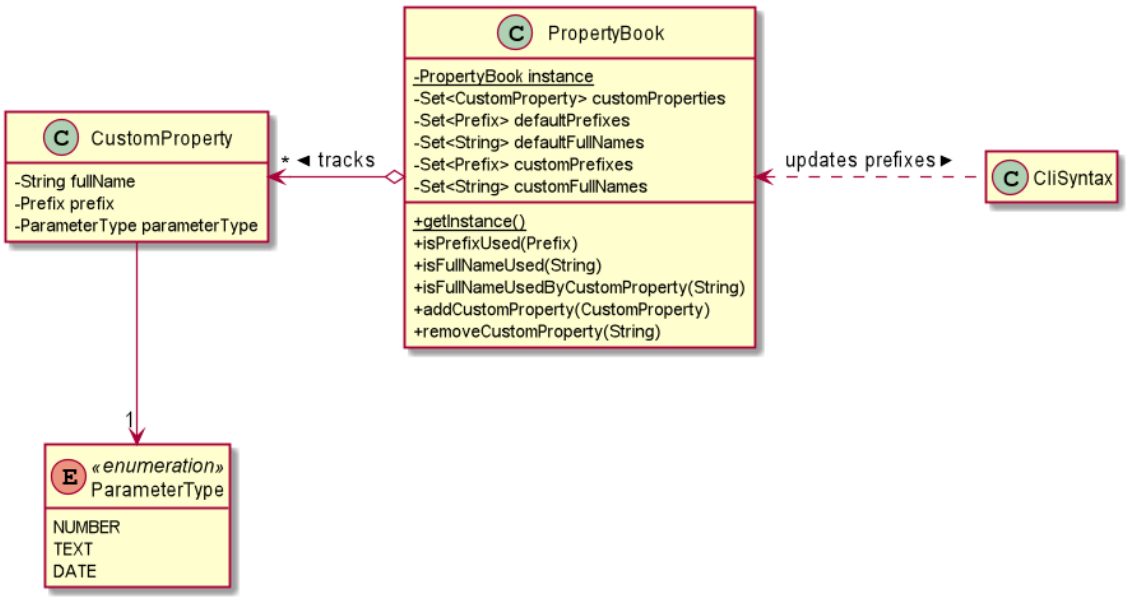

Figure 16. Class diagram of the associations of `PropertyBook` and `CustomProperty`

Adding Custom Properties

To add a new custom property for the exercises, the user can do it through the command `custom s/PREFIX_NAME f/FULL_NAME p/PARAMETER_TYPE`. Examples include `custom s/r f/Rating p/Number` and `custom s/ed f/Ending Date p/Date`.

The following sequence diagram will illustrate how the custom operation works when a custom property is **successfully added**.

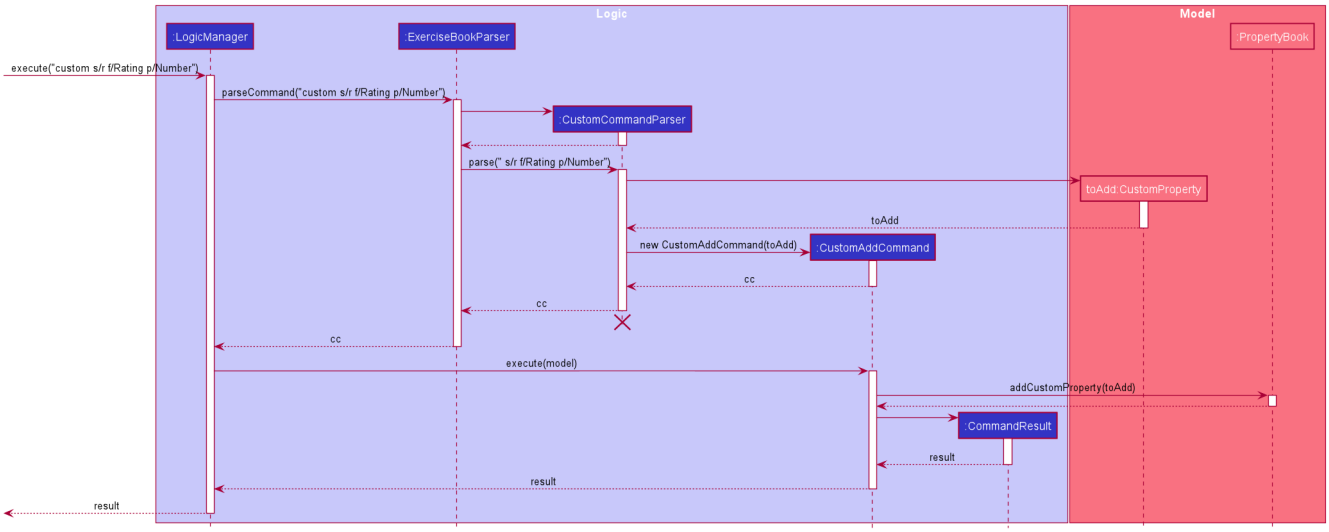

Figure 17. Sequence diagram of a successful addition of a custom property

For further clarity, one can identify the above diagram with the following sequence of steps:

Step 1: User first defines the custom property they wish to add for the exercises.

Step 2: The custom property will be parsed by the app's parser and a new `CustomProperty` object is created.

Step 3: This `CustomProperty` object will be returned together with a newly created `CustomAddCommand` object.

Step 4: The `execute` method of the `CustomAddCommand` method will be called and the `CustomProperty` object will be added to `PropertyBook`.

Step 5: Finally, a `CommandResult` object will be created and returned.

The above steps illustrate the main success scenario. However, not all additions of a custom property will be successful. The next activity diagram shows the workflow when a new custom property is defined.

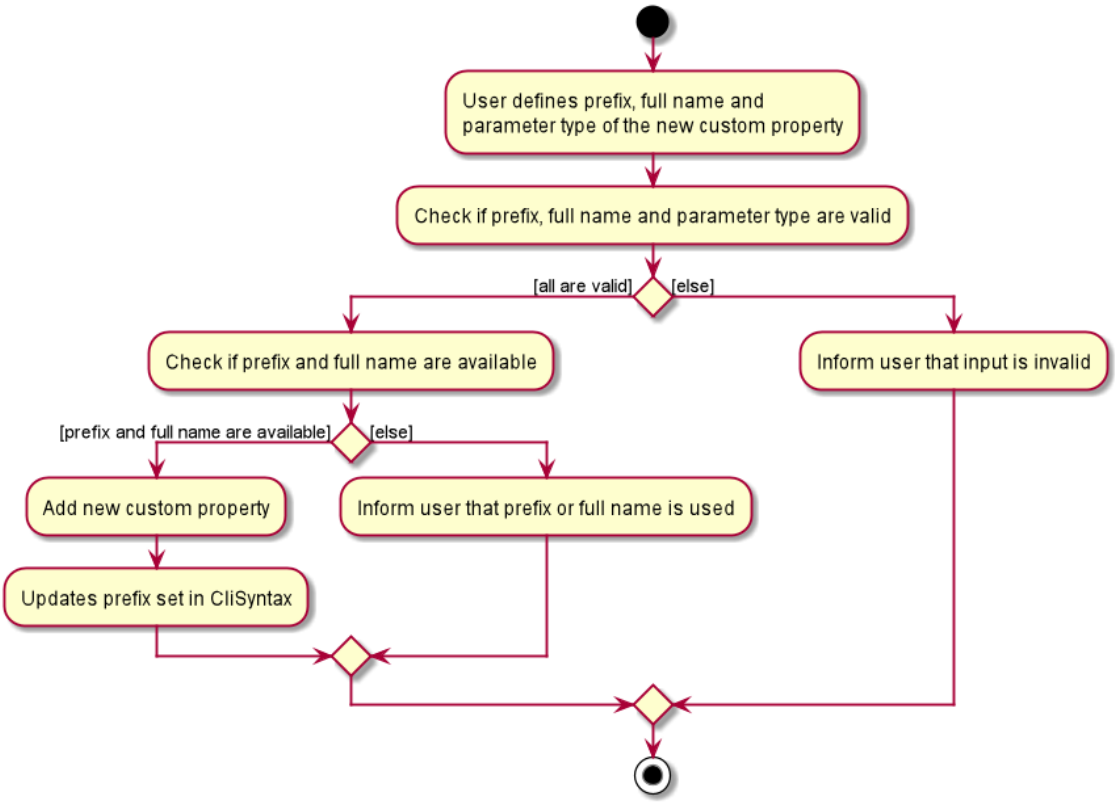

Figure 18. Activity diagram of the workflow when a new custom property is added

Once a custom property is successfully added into `PropertyBook`, the user can use the prefix of the custom property in `add` or `edit` command.

Removing Custom Properties

Should a user wish to remove a custom property from all of the exercises, he/she can simply make use of the command `custom rm/FULL_NAME`. A custom property that has been removed from the `PropertyBook` can be re-added back if the user chooses to. Alternatively, if the user wishes to remove a custom property just from a single exercise, he/she can choose to enter `custom rm/FULL_NAME i/INDEX` instead.

The next sequence diagram illustrates what happens when a custom property is removed from the `PropertyBook`. If a custom property is removed from a single exercise instead, only the selected exercise will be updated.

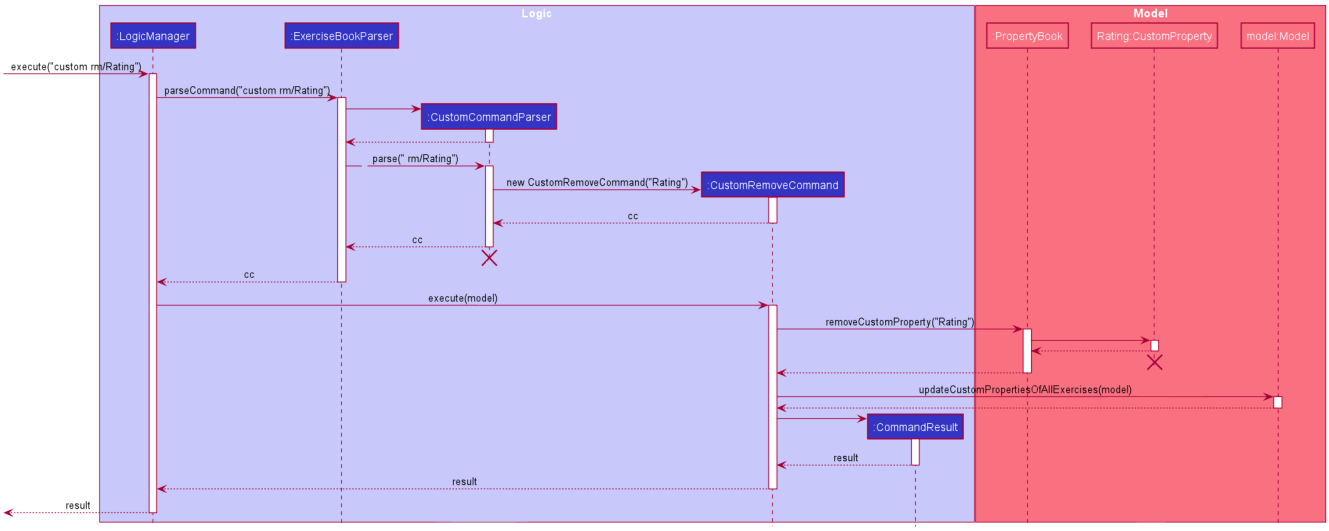

Figure 19. Sequence diagram of a successful removal of a custom property from all exercises

3.4.4. Design Considerations

Aspect: `PropertyBook` **design**

- **Choice 1 (Current choice)**: Represent `PropertyBook` as a singleton class that will act as the only access point for the addition and removal of custom properties.
  - Pros: Having a singleton helps to provide more utility for methods that rely on the `CustomProperty` objects that have been created.
  - Cons: It makes testing much difficult as the results from the previous test cases are carried over. Furthermore, it increases coupling across the code base.
- **Choice 2**: Represent `PropertyBook` as a usual Java object that can be instantiated many times.
  - Pros: This reduces coupling and makes testing easier as a new `PropertyBook` object independent of the other tests can be created for different tests.
  - Cons: There could be situations where 2 instances of `PropertyBook` objects are created and the addition of a custom property is done to only one instance and not in the other.

After much consideration, Choice 1 was implemented with the following reasons:

1. `AddCommandParser` and `EditCommandParser` have to gain access to the `CustomProperty` in order to ensure that the values entered for the custom properties in the add/edit commands are valid. However, as the `ExerciseBookParser` in the original code base only takes in a `String` as a parameter, there has to be another way of retrieving the custom properties. While we can change the `ExerciseBookParser` to take in a data structure containing `CustomProperty` objects, this does not seem good as its responsibility is just to ensure that a predefined command is entered and is passed to the correct command parser.A slightly better choice in this case is to make the data structure holding the `CustomProperty` objects a static variable and parsers that require it can access it directly.
2. If the data structure holding the `CustomProperty` object is to be made static, it means that this information is shared among all of the `PropertyBook` instances if Choice 2 was implemented. Thus, `PropertyBook` is acting like a singleton and so, a singleton class will be appropriate.

3.5. Suggest

3.5.1. Rationale

Beginners now have a plethora of choices, which may overwhelm them when they are deciding on what exercises to do. Thus, we decided to provide users with sample exercise routines to reduce the inertia of starting this lifestyle change. On the other hand, regular gym goers may face a repetitive and mundane exercise routine or may want to experiment with different exercises. As such, to put it briefly, we decided to give users the ability to discover exercises based on the characteristics they are interested in.

This feature presents a cohesive function that all users can benefit from. It also makes our application well-rounded so that users can better achieve their fitness goals.

3.5.2. Overview

The sample exercise routines are currently implemented in ExerHealth's database as a hard-coded set of exercises. More importantly, the `SuggestPossible` command which caters to more experienced gym goers utilises the exercises that the user has already done, in addition to ExerHealth's database. Hence, we allow users to search for suggestions based on `Muscle` and `CustomProperty`.

3.5.3. Current Implementation

The `SuggestBasic` command displays a list of exercises from our database to the user. The `SuggestPossible` command is created by parsing the user's inputs to form a `Predicate` before filtering ExerHealth's database and the user's tracked exercises.

The following activity diagram summarizes what happens when a user enters a `SuggestPossible` command:

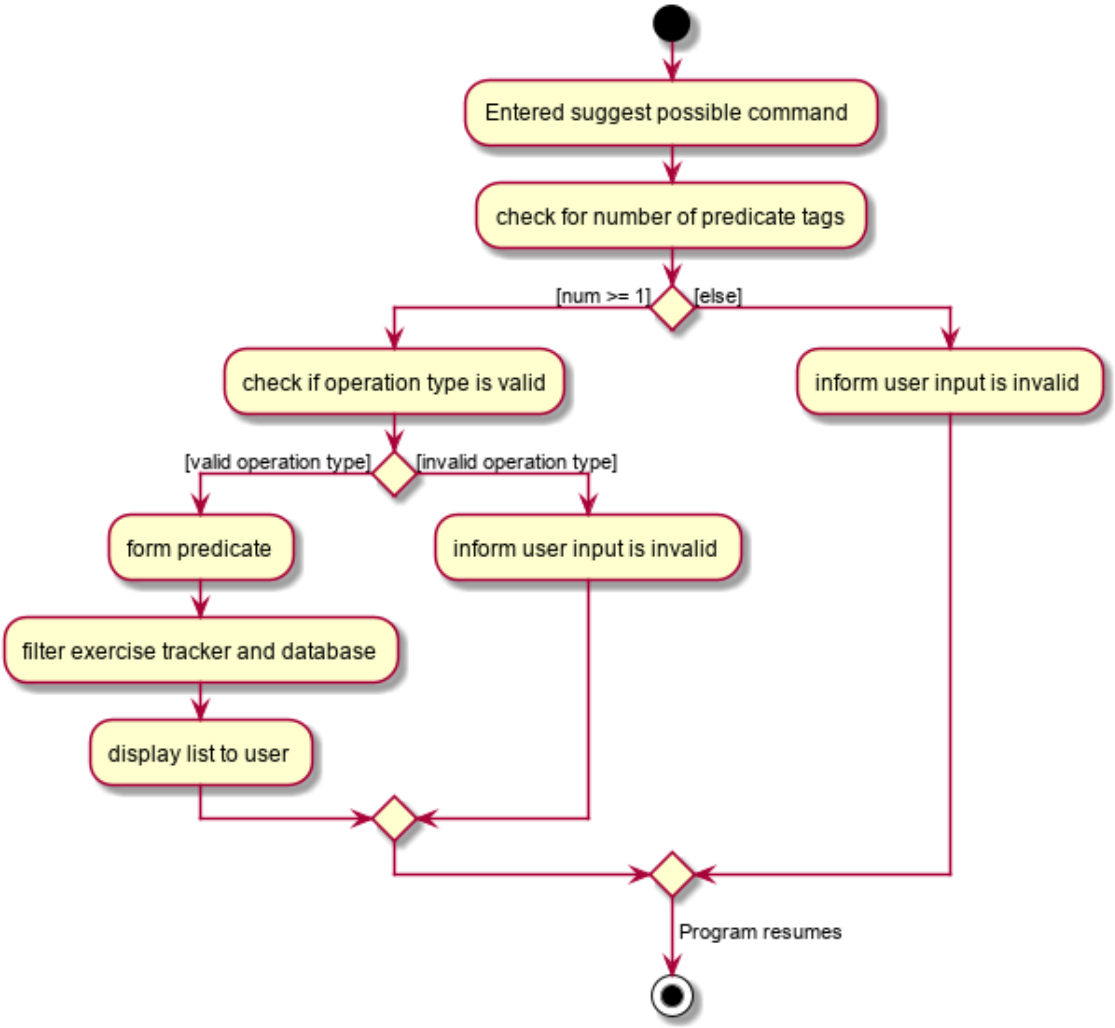

Figure 20. Activity diagram showing the workflow of a `SuggestPossible` command

In detail, when a `SuggestPossible` command is entered, the `Logic` component is responsible for parsing the inputs into a `Predicate`. The `Predicate` is then used to instantiate a `SuggestPossible` command, and later used to filter a list of `Exercise` when the command is executed. The interactions between the multiple objects can be captured using a sequence diagram.

The following sequence diagram shows the sequence flow when a user enters a valid `SuggestPossible` command:

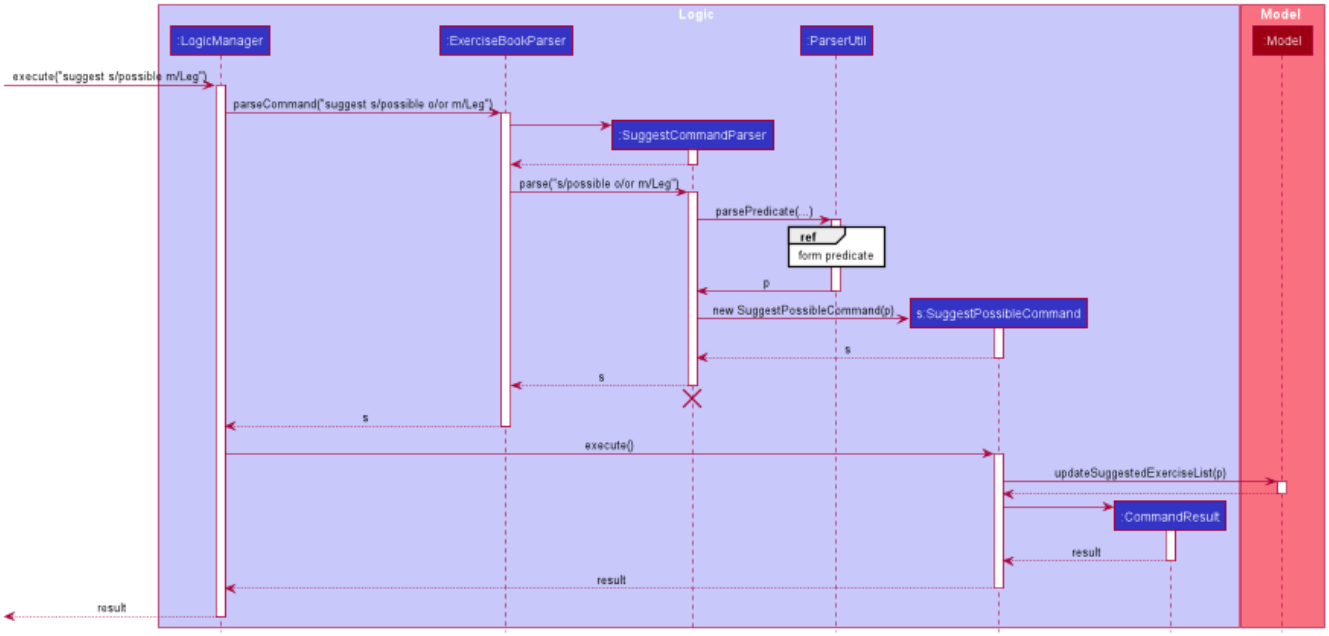

Figure 21. Sequence diagram of a `SuggestPossibleCommand`.

From the sequence diagram:

1. When the `LogicManager` receives the `execute` command, it calls the `parseCommand` method of `ExerciseBookParser`.
2. `ExerciseBookParser` will receive `suggest` as the command type and instantiate `SuggestCommandParser` to further parse the command.
3. `SuggestCommandParser` will receive `s/possible` as the suggest type and calls the `parsePredicate` method of `ParserUtil` to parse the user input to create an `ExercisePredicate` object (named `p` in the diagram).
4. `SuggestCommandParser` will instantiate `SuggestPossibleCommand` with the `ExercisePredicate` as the constructor parameter.
5. The `SuggestPossibleCommand` object is then returned to `SuggestCommandParser`, followed by `ExerciseBookParser`, and lastly back to `LogicManager` to execute.
6. `LogicManager` will proceed to `execute SuggestPossibleCommand`.
7. `SuggestPossibleCommand` then calls the `updateSuggestedExerciseList` method in `ModelManager`, passing in the predicate to filter the list of suggest exercises.
8. `SuggestPossibleCommand` creates a new `CommandResult` to be returned.

In step 3, the process in which the `ExercisePredicate` object is created can be explored deeper.

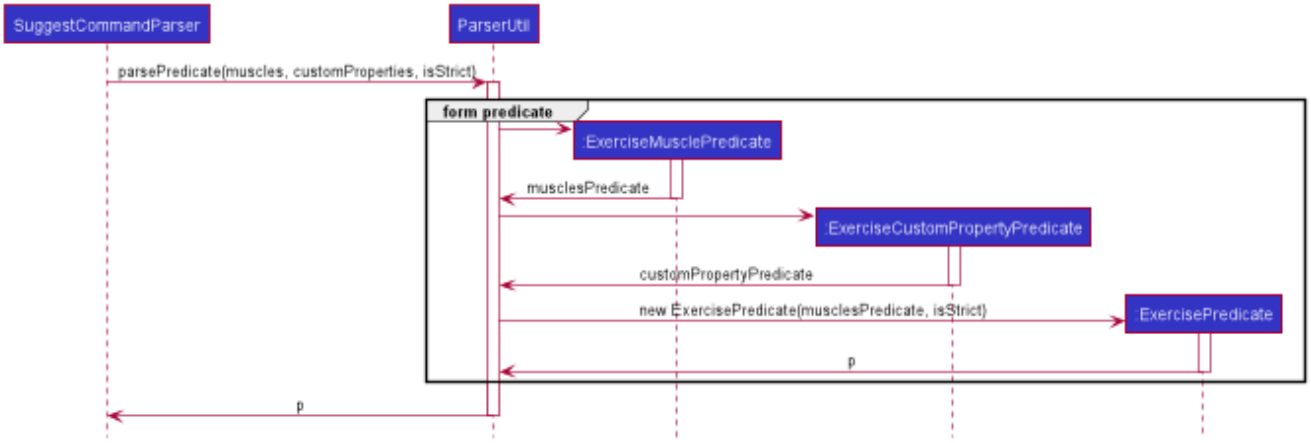

Figure 22. Sequence diagram of how an `ExercisePredicate` is created

From the sequence diagram above:

1. `ParserUtil` creates `ExerciseMusclePredicate` and `ExerciseCustomPropertyPredicate` with the input parameters.
2. Since there were no CustomProperty tags to filter, `ParserUtil` creates `ExercisePredicate` with only the `musclesPredicate` and the boolean `isStrict`.
3. The resulting `ExercisePredicate` is then returned to `ParserUtil`, followed by `SuggestCommandParser`.

The diagram below shows the structure of a `ExercisePredicate` object. A `ExercisePredicate` contains a `list` of `BasePropertyPredicate`, where each contains a `Collection` of `Muscle` or `CustomProperty`.

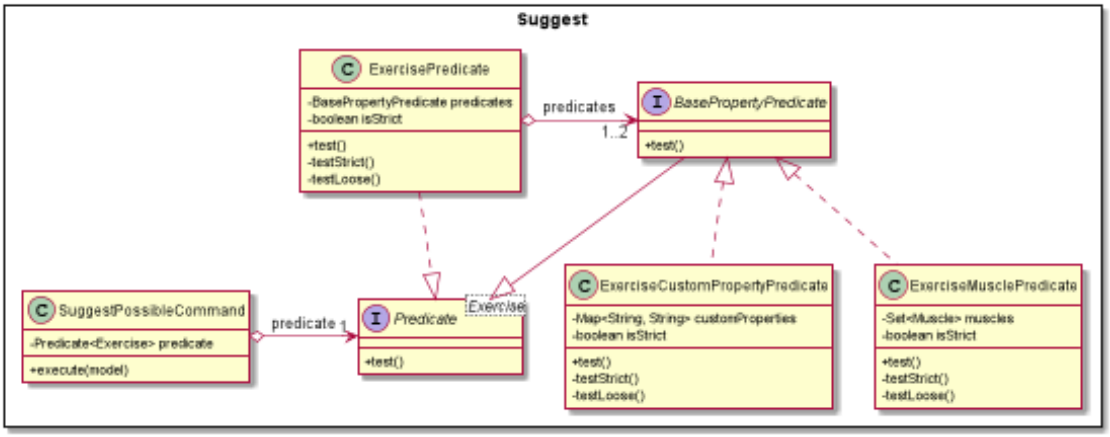

Figure 23. Class diagram of the classes behind the suggest possible feature

Creating classes such as `ExerciseCustomPropertyPredicate` and `ExerciseMusclePredicate` allows us to conduct better testing because we can compare the `Collection` of `Muscle`/`CustomProperty` that is being considered.

3.5.4. Design Considerations

Aspect: Implementation of predicate creation

- **Choice 1:** `SuggestPossibleCommand` to handle the predicates.
  - Pros:
    - Easy to implement and understand. The class `SuggestPossibleCommand` contains the parsing and creation of the predicate all in one place as it stores the tags, and creates the predicate and filters the list of exercises.
  - Cons:
    - Violation of Single Responsibility Principle (SRP) as `SuggestPossibleCommand` updates the model and creates the predicate.
- **Choice 2 (current choice):** Predicate class to handle all predicates.
  - Pros:
    - Adheres to SRP and Separation of Concern (SoC).
  - Cons:
    - Increases the complexity of the code as more classes are needed, and also increases the lines of code written.

3.6. Statistics

3.6.1. Implementation

Statistics of exercises will be displayed in charts. Supported chart types are Pie Chart, Line Chart and Bar Chart. StatsFactory will create Statistic using given parameters. The figure below shows the class diagram of statistics:

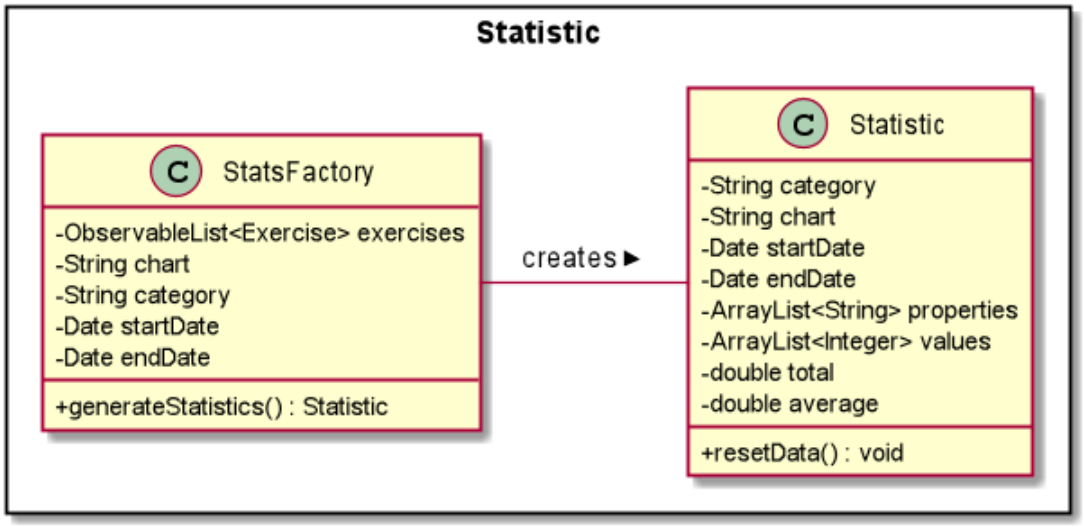

Figure 24. Class diagram of the classes behind the statistics feature

The next figure shows the activity diagram when user enter a `stats` command:

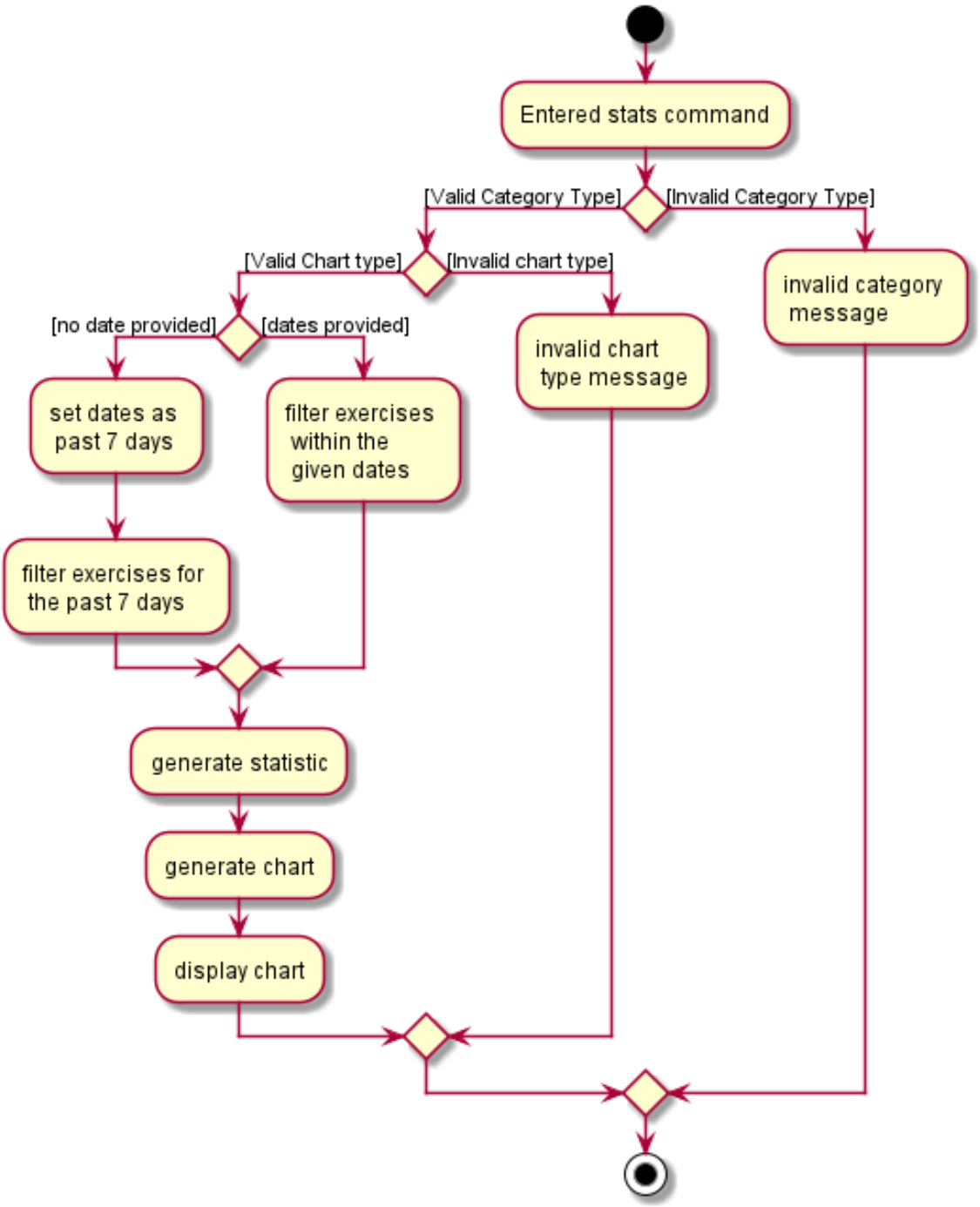

Figure 25. Workflow when a user enters a `stats` command

Given below is an example usage scenario of statistics feature.

Step 1: User enters a `stats` command to see statistics of exercises.

Step 2: `ExerciseBookParser` will receive command from `LogicManager` and pass command to `StatsCommandParser`.

Step 3: `StatsCommandParse` will parse the command and creates a `StatsCommand`.

Step 4: `StatsCommand` calls `Model#getExerciseBookData` to get data of all exercises.

Step 5: `StatsCommand` creates a `StatsFactory` and pass exercises data, chart and category to `StatsFactory`.

Step 6: `StatsFactory` will then generate `Statistic` and return to `StatsCommand`.

Step 7: `StatsCommand` then calls `Model#setStatistic` to set the `Statistic` in `Model`.

Step 8: `StatsCommand` creates a new `CommandResult` and return to `LogicManager`.

Shown below is the sequence diagram when user enters a valid `stats` command:

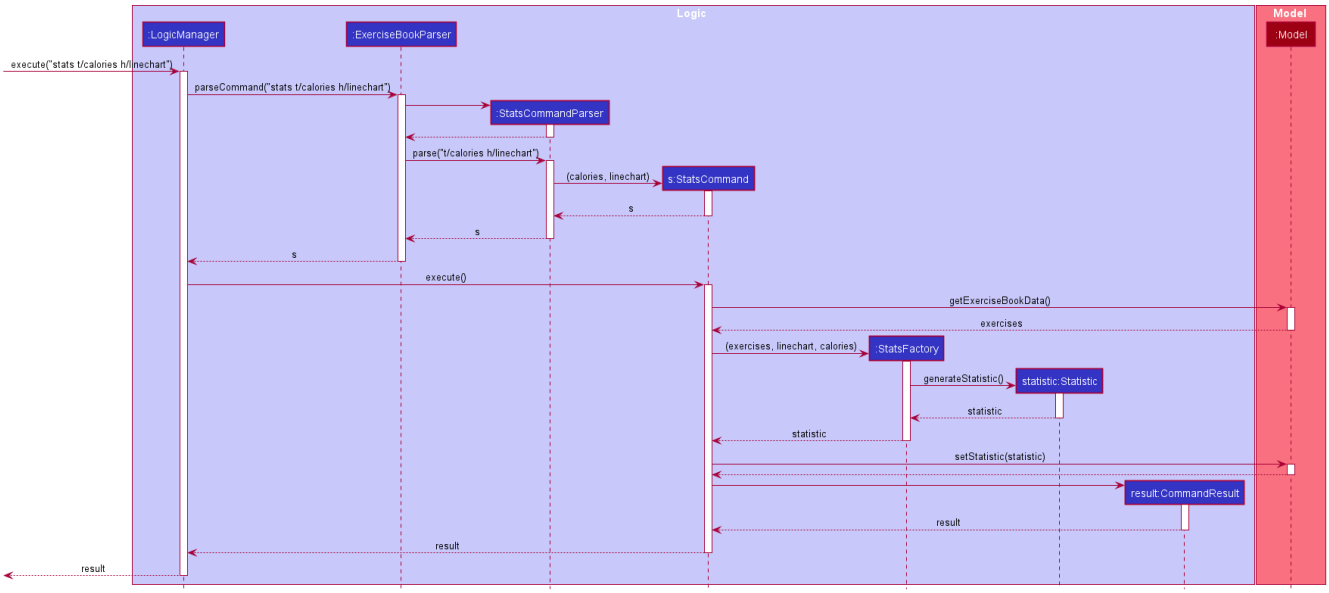

Figure 26. Sequence diagram of a `stats` command

3.7. Logging

We are using `java.util.logging` package for logging. The `LogsCenter` class is used to manage the logging levels and logging destinations.

- The logging level can be controlled using the `logLevel` setting in the configuration file (See Section 3.8, “Configuration”)
- The `Logger` for a class can be obtained using `LogsCenter.getLogger(Class)` which will log messages according to the specified logging level
- Currently log messages are output through: `Console` and to a `.log` file.

Logging Levels

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

3.8. Configuration

Certain properties of the application can be controlled (e.g user prefs file location, logging level)

through the configuration file (default: `config.json`).

4. Documentation

Refer to the guide here.

5. Testing

Refer to the guide here.

6. Dev Ops

Refer to the guide here.

Appendix A: Product Scope

Target user profile:

- exercises on a regular basis
- actively monitors exercise records
- develops exercise regimes for the future
- prefers desktop apps over other types
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition:

- provides an integrated platform to track and access past exercise records
- shows more complex data analytics than the statistics a standard tracking app provides
- allows flexible and conflict-free scheduling of planned exercises
- provides exercise suggestions based on past activities

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…
* * *	student who exercises	monitors the types and quantity of the exercises I have completed	remember and have different variations to my exercises

Priority	As a …	I want to …	So that I can…
* * *	athletic student	have a way to store all my exercises and their relative intensities	make reference to past exercises while scheduling future exercises
* * *	frequent gym-goer	keep track of my rep counts	know how hard I have pushed and how far I am from my target reps
* * *	student who wants to get stronger	keep track of my the muscles my gym exercise works on	plan what kind of muscle groups I should target to get stronger
* * *	amateur at exercising	have the app come up with exercises for me based on my user profile	better plan future regimes based on my previous attempt
* * *	student who just got into exercising	have some sample training plans	have a starting point for my exercise regime
* * *	frequent gym-goer with targets	see my progression for every exercise and the date I completed them	see how much I have improved
* * *	Student who loves visual data	visualise my exercise statistics	understand all my relevant data immediately
* * *	student who is very busy	have the app detect clashes in my exercising schedules	reschedule some of my exercises somewhere else
* * *	person who likes customization	add in new attributes for exercises	tailor the app for my personal use
* * *	careless athletic student	be able to have a way to undo my actions	easily undo my command when I accidentally delete one of my training plans
* * *	careless athletic student	be able to have a way to redo my actions	simply redo my undone command when I realize I undid an important exercise
* * *	athletic student who has a fixed training plan	have a way to store this training plan permanently	save some trouble of constantly updating the app whenever I want to begin on that training plan

Priority	As a …	I want to …	So that I can…
* *	student who is impatient	have simple commands	input new entries quickly
* *	health-conscious student	keep track of my daily calories burnt	monitor my calorie count over a specific duration
* *	student who wants to get stronger	Know what kind of muscles I have been training for the past month	take note of which muscles I have been focusing for training
* *	student who wants to track exercises quickly and efficiently	be able to add exercises from history	add the same thing without having to type it all out
* *	student who wants a balanced exercise regime	have the app auto suggest some forms of exercise	easily find new exercises to do
* *	athletic student	be able to modify my current training schedule	easily adapt my previous training plans into better plans that can help improve my physique
* *	athlete who wants to improves	save notes from my previous session	reflect and modify my training regime accordingly to suit my pace
* *	athletic student who loves to do things my way	be able to define my own command syntax	type the commands for the various features much easily and quickly
*	athletic student who uses the app often	have an auto-complete or input suggestion feature	easily add in reused exercises conveniently without having to type them out fully again

Priority	As a …	I want to …	So that I can…
*	student who likes to keep things neat	be able to archive my older exercises	be more focused on recent exercises
*	student who just got into gyming	receive some tips on good gym practices	avoid injuring myself during training
*	student who just got into sports	Understand the most important tips on good exercise habits	maximise the benefits of my exercises
*	student who wants to get stronger	be advised on how much increment I should make for each exercise	train progressively
*	athletic student	be able to keep track of my recovery period	avoid doing harm to my body from excessive training
*	forgetful student	be reminded of when i have to exercise	set aside time for exercising
*	frequent gym-goer	be reminded of my exercise schedules	remember to go for my sessions
*	athletic student	monitor the list of equipment I need for each session	remember what I need for subsequent exercise sessions of the same kind
*	frequent gym-goer	store my workout music playlist	access my favourite gym workout playlist conveniently when gyming
*	student with a busy schedule	be able to export my files	to resolve conflicts between my exercise and work schedule
*	student who is very lazy	be able to mass import all my exercises data from other platforms	save the trouble of inputting an entire list of existing entries one by one
*	student who uses mobile running apps	import data from other application	avoid the time-consuming process of adding all exercises manually

Appendix C: Use Cases

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

UC01: Statistics (bar chart)

System: ExerHealth

Actor: user

MSS

1. User adds multiple exercises to the ExerHealth tracker
2. User requests to see a bar chart of the most frequently done exercises within a range of date
3. ExerHealth shows user the breakdown of exercises and their respective frequency for the date range

   Use case ends.

UC02: Schedule

System: ExerHealth

Actor: user

MSS

1. User requests for the list of exercises.
2. ExerHealth displays the list of exercises it is tracking.
3. User adds 1 or more exercises to a regime
4. ExerHealth adds the regime to the user's list of regime and display successful addition
5. User schedules a regime at a date
6. ExerHealth schedules regime at the date and displays successful scheduling

   Use case ends.

Extensions

- 5a. ExerHealth detects more than one regime at the date
  - 5a1. ExerHealth displays resolve window to user
  - 5a2. User enters which exercises they wish to schedule at the date from the conflicting regimes
  - 5a3. ExerHealth schedules the newly made regime at the date and closes resolve window

Use case ends

UC03: Suggest

System: Exerhealth

Actor: user

MSS

1. User asks for suggestions
2. System searches database for previous exercises done
3. System creates a suggestion based on search and request type

   Use case ends

UC04: Custom

System: ExerHealth

Actor: user

MSS

1. User requests to add in a new user-defined property for exercises
2. ExerHealth adds in the user-defined property for all exercises
3. User adds a new exercise with the newly specified prefix and argument for the property

   Use case ends

Extensions

- 1a. ExerHealth detects that the prefix name/full name of the user-defined property is a duplicate of another property/parameter for add / edit command.
  - 1a1. ExerHealth informs the user that the prefix name/full name of his/her new property is a duplicate of a current property/parameter for add / edit command.

    Use case ends

UC05: Undo

System: ExerHealth

Actor: user

MSS

1. User executes an undoable command
2. ExerHealth performs the change
3. User undoes the latest command
4. ExerHealth undoes the latest change

   Steps 3-4 can be repeated for as many times as required until there is no undoable command left to undo

   Use case ends

Extensions

- 3a. The undo history is empty
  - 3a1. ExerHealth informs user that undo is not allowed at this point

    Use case ends

UC06: Redo

System: ExerHealth

Actor: user

MSS

1. User undoes the latest command
2. ExerHealth undoes the latest change
3. User redoes the latest undoable command that was undone
4. ExerHealth redoes the command again

   Steps 3-4 can be repeated for as many times as required until there are no more undoable command left to redo

   Use case ends

Extensions

- 3a. There is no action to redo as the user has not executed undo before
  - 3a1. ExerHealth informs user that redo is not allowed at this point

    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 1000 exercises without a noticeable sluggishness in performance for typical usage.
3. A user with above average typing speed for regular English text (i.e. not code, not system admin commands) should be able to accomplish most of the tasks faster using commands than using the mouse.
4. Should work without requiring an installer.
5. Should not depend on a remote server.
6. Should be for a single user i.e. (not a multi-user product).

Appendix E: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X.

Regime

A specific set of exercises that are to be done together. For example, a **Legs** regime at the gym can include multiple exercises such as squats, hamstring curl and calf raises.

Schedule

Planning of an exercise on a later day.

Property

An attribute of an exercise item. Pre-defined attributes include name, quantity, units and calories.

Prefix

The term that comes before each parameter in the command. For example, the prefix in `p/Number` is `p/`.

Prefix Name

The word that comes before `/` in the prefix. For example, the prefix name of `p/` is `p/

7. 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.

7.1. 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 exercises. 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.

7.2. Deleting an exercise

1. Deleting an exercise while all exercises are listed

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

   b. Test case: `delete t/exercise i/1`
   Expected: First exercise is deleted from the list. Details of the deleted exercise shown in the status message.

   c. Test case: `delete t/exercise i/0`
   Expected: No exercise is deleted. Error details shown in the status message. Status bar remains the same.

   d. Other incorrect delete commands to try: `delete t/exercise`, `delete t/exercise i/x` (where x is larger than the list size)
   Expected: Similar to previous.

7.3. Deleting an exercise in regime

1. Deleting an exercise in regime while all regimes are listed.

   a. Prerequisites: List all regimes using the `list` command. Regime named `Level 1` has multiple exercises and is in the list.

   b. Test case: `delete t/regime n/Level 1 i/1`
   Expected: The first exercise in `Level 1` regime is deleted.

   c. Test case: `delete t/regime n/Level 1 i/0`
   Expected: No exercise is deleted in `Level 1` regime.

   d. Other incorrect delete commands to try: `delete t/regime`, `delete t/regime i/x` (where x is larger than the list size)
   Expected: Similar to previous.

7.4. Deleting a regime

1. Deleting a regime while all regimes are listed.

   a. Prerequisites: List all regimes using the `list` command. Regime named `Power set` is in the list.

   b. Test case: `delete t/regime n/Power set`

Expected: The regime named `Power set` is deleted from the list.

c. Test case: `delete t/regime n/power set`
Expected: `Power set` regime is not deleted as name is case-sensitive. Error details shown in the status message.

7.5. Scheduling a regime

1. Scheduling an exercise regime

   a. Prerequisites: Have an exercise regime of name `cardio`.

   b. Test case: `schedule n/cardio d/12/12/2019` with no other schedule on `12/12/2019`
   Expected: Regime `cardio` is now scheduled on `12/12/2019`. Details of schedule should be shown in the center information panel and the left panel should switch to show schedule list.

   c. Test case: `schedule n/cardio d/12/12/2019` with a conflicting schedule on `12/12/2019`
   Expected: Scheduling conflict exist and the resolve window should pop up showing the already scheduled regime on the left panel and the conflicting `cardio` schedule on the right panel.

7.6. Resolving scheduling conflict

1. Resolves a scheduling conflict by taking one whole regime

   a. Prerequisites: Resolve window should be shown on scheduling conflict

   b. Test case: `resolve n/conflicting`
   Expected: The conflicting schedule on the right panel should be taken as the resolved schedule and the resolve window should close. The conflicting schedule should now be scheduled on the conflicting date. Details of the schedule is shown on the center information panel.

2. Resolves a scheduling conflict by taking some exercise from both regime

   a. Prerequisites: Resolve window should be shown on scheduling conflict and `new cardio` should not exist in the user's regime list

   b. Test case: `resolve n/new cardio i/1 r/2`
   Expected: A new regime is created called `new cardio` with the exercises from scheduled regime's first index and conflicting regime's second index. Resolve window should close. The newly made regime is now scheduled on conflicting date. Details of the schedule shown on the center information panel.

7.7. Suggest

1. Suggest basic exercises

   a. Test case: `suggest s/basic` Expected: A list of basic exercises displayed on the

2. Suggest possible exercises

   a. Prerequisites: There is at least an exercise being tracked or in the database tagged with a

`muscle`.
For example, `add t/exercise n/Run d/03/11/2019 c/200 q/10 u/km m/Legs`.

i. Test case: `suggest s/possible m/Legs`
Expected: This exercise, along with database's exercises tagged with `Legs` are displayed.

ii. Test case: `suggest s/possible o/and m/Legs`
Expected: Similar to previous.

iii. Test case: `suggest s/possible o/or m/Legs`
Expected: Similar to previous.

b. Prerequisites: A `CustomProperty` is created and there are exercises being tracked with `CustomProperty`.
For example,

1. `custom s/r f/Rating p/Number`
2. `add t/exercise n/Run d/03/11/2019 c/200 q/10 u/km m/Legs r/8`
3. `add t/exercise n/Bench Press d/05/11/2019 c/150 q/40 u/kg m/Chest r/8`

i. Test case: `suggest s/possible o/and m/Chest r/8`
Expected: The previously added exercise `Bench Press` is displayed.

ii. Test case: `suggest s/possible o/or m/Chest r/8`
Expected: The previously added exercises `Bench Press` and `Run` are displayed. In addition, exercises from database that are tagged `Chest` are also displayed.

c. Test case: `suggest s/possible m/Chest m/Legs`
Expected: Error details shown in the status message.

d. Test case: `suggest s/possible o/or`
Expected: Similar to previous.

7.8. Statistic

1. Display charts and statistic for completed exercises

a. Test case: `stats t/calories h/barchart s/01/11/2019 e/30/11/2019`
Expected: The chart in the right panel will be updated to a bar chart. Total and average will be shown below the chart.

b. Test case: `stats t/calories h/piechart s/01/11/2019 e/30/11/2019`
Expected: The chart in the right panel will be updated to a pie chart. Total and average will be shown below the chart.

c. Test case: `stats t/calories h/linechart s/01/11/2019 e/30/11/2019`
Expected: The chart in the right panel will be updated to a line chart. Total and average will be shown below the chart.

d. Test case: `stats t/calories h/barchart s/01/01/2019 e/02/02/2019`
Expected: The chart in the right panel is not updated. Error message shown in the status message.

7.9. Adding a custom property

1. Adding a custom property when app is first launched.

    a. Prerequisites: The custom property `Remark` with the prefix name `re` must not be created yet. The following test cases should be tried in order.

    b. Test case: `custom s/re f/Remark p/Text`
    Expected: The custom property `Remark` is created for all exercises. The prefix name and full name of the property will be displayed in the status message.

    c. Test case: `custom s/re f/AnotherRemark p/Text`
    Expected: No custom property is created. An error message will be shown, informing the user that the prefix has been used for an existing parameter in add/edit command.

    d. Test case: `custom s/tt f/Remark p/Text`
    Expected: No custom property is created. An error message will be shown, informing the user that the name has been used by an existing property.

7.10. Deleting a custom property

1. Deleting a custom property.

    a. Prerequisites: The custom property `Remark` with the prefix name `re` should have been created. There should be at least 3 exercises with the custom property `Remark`, preferably the exercises at indices 1 to 3. The following test cases should be tried in order.

    b. Test case: `custom rm/Remark i/1`
    Expected: `Remark` is removed from exercise 1. A message informing the user that `Remark` has been removed from exercise 1 will be shown. The `Remark` property for exercises 1 and 2 will still be present.

    c. Test case: `custom rm/Remark` Expected: `Remark` is removed from the app. A message informing the user that `Remark` has been removed will be shown. The `Remark` property is removed from all exercises.

    d. Test case: `custom rm/Date` Expected: No custom property is removed. An error message informing the user that `Date` is not used by an custom property will be shown.

7.11. Saving data

1. Dealing with missing/corrupted data files

    a. Prerequisites: Must have ran `ExerHealth` at least once and have `exercisebook.json`.

    b. Open up `exercisebook.json` with any text editor and change one of the dates to `//`, representing an invalid date.
    Expected: `ExerHealth` will start with an empty exercise book due to data corruption. Exercise Panel will be empty.