Sugar Mummy - Developer Guide

By: `AY1920S1-CS2103-T16-1` Since: `Oct 2019` Licence: `MIT`

1. Setting up

Refer to the guide here.

2. Design

2.1. Architecture

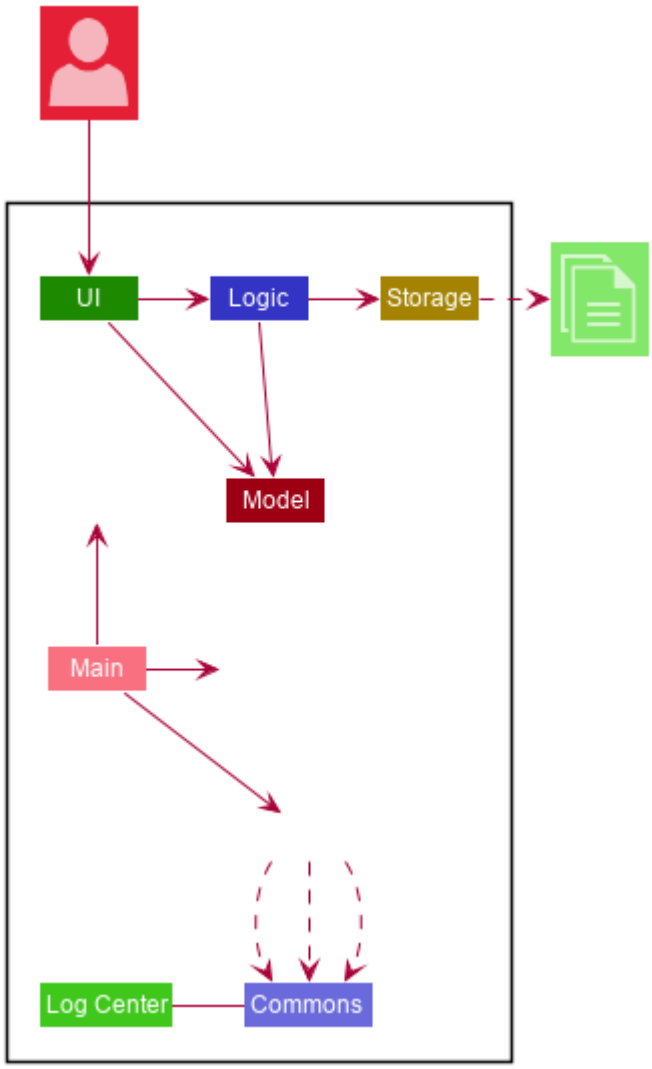


Figure 1. Architecture Diagram

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.

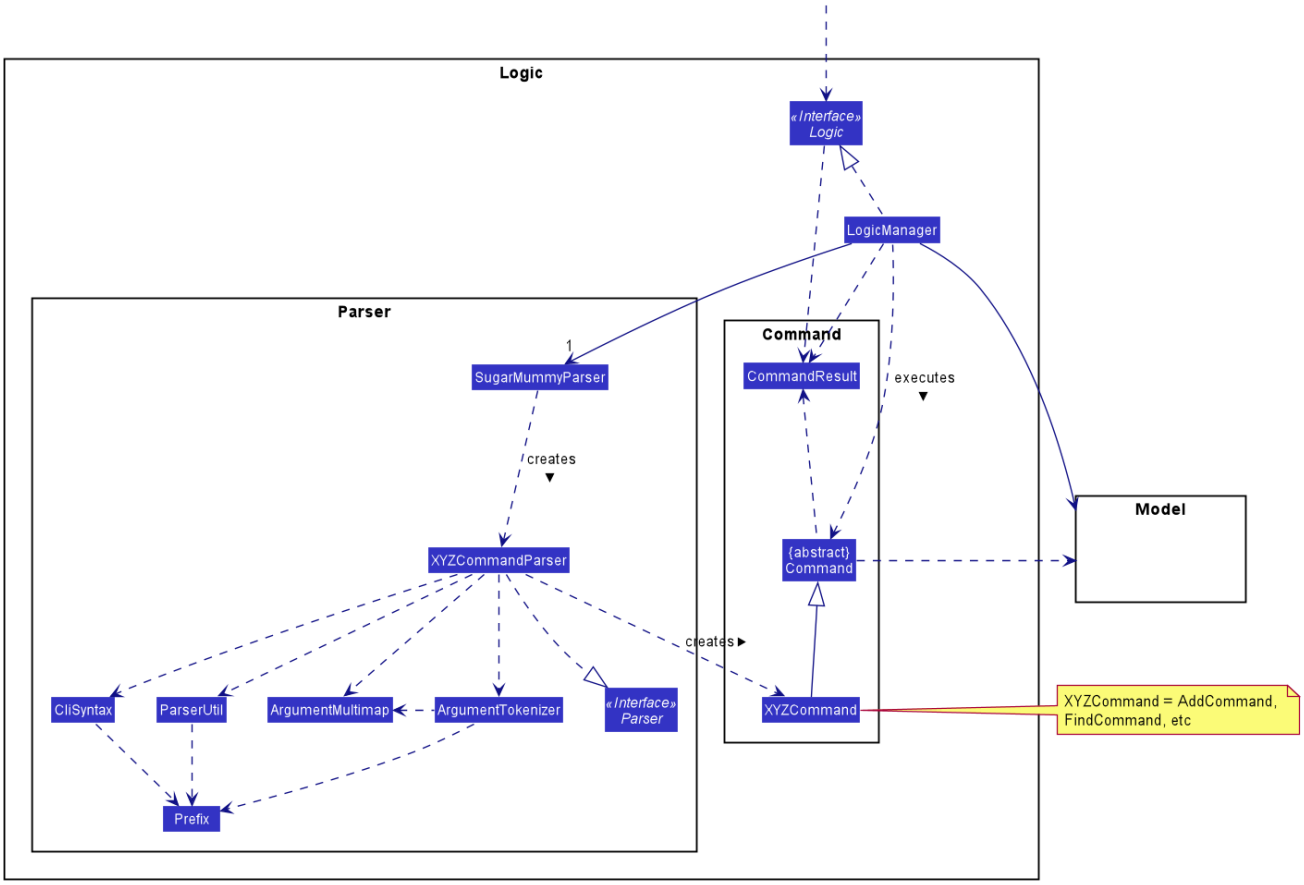


Figure 2. Class Diagram of the Logic Component

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 1`.

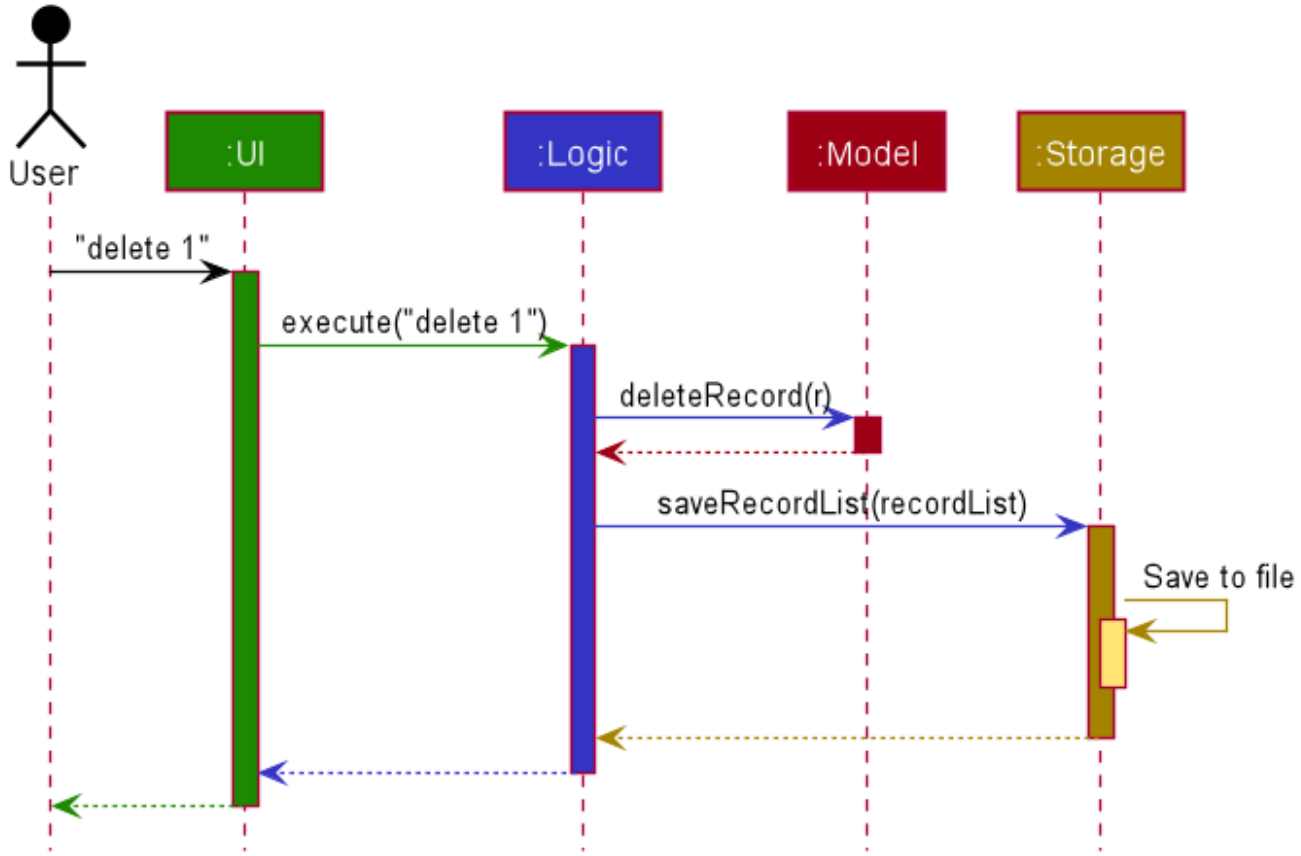


Figure 3. Component interactions for `delete 1` command

The sections below give more details of each component.

2.2. UI component

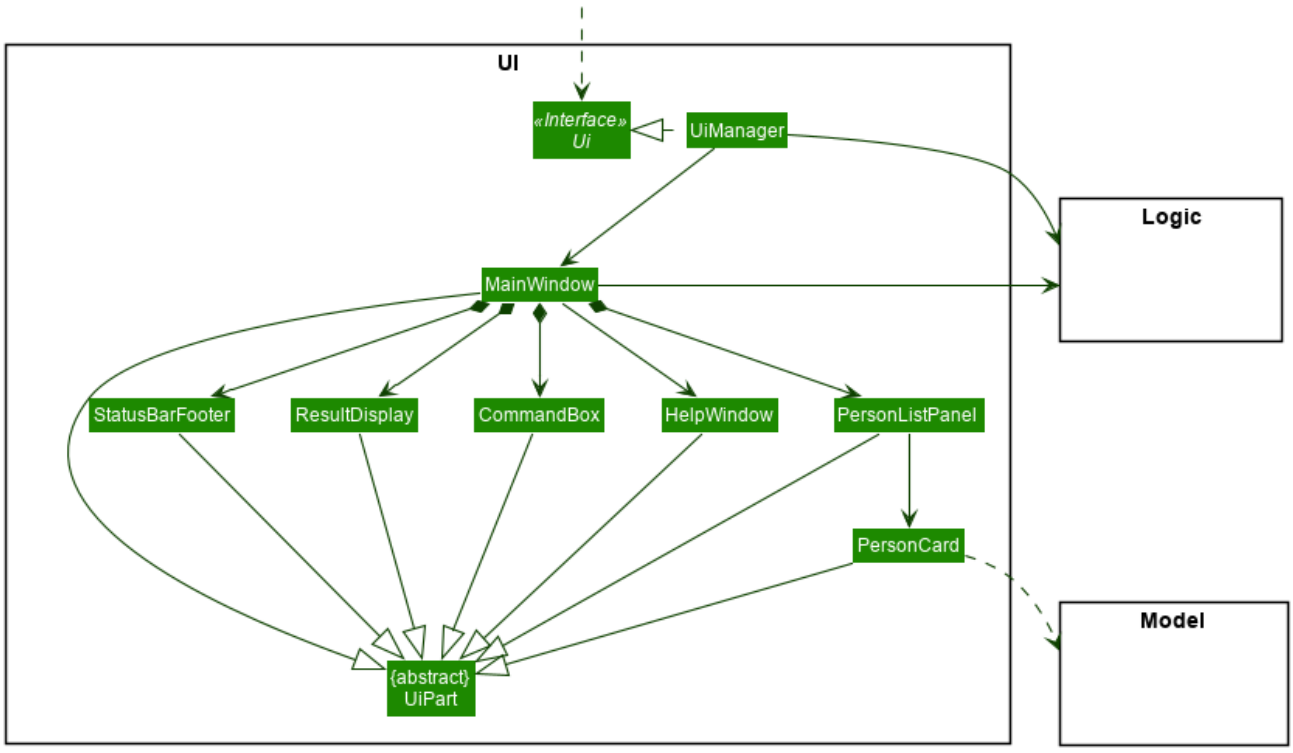


Figure 4. Structure of the UI Component

API : `Ui.java`

The UI consists of a `MainWindow` that is made up of parts e.g.`CommandBox`, `ResultDisplay`, `PersonListPanel`, `StatusBarFooter` 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

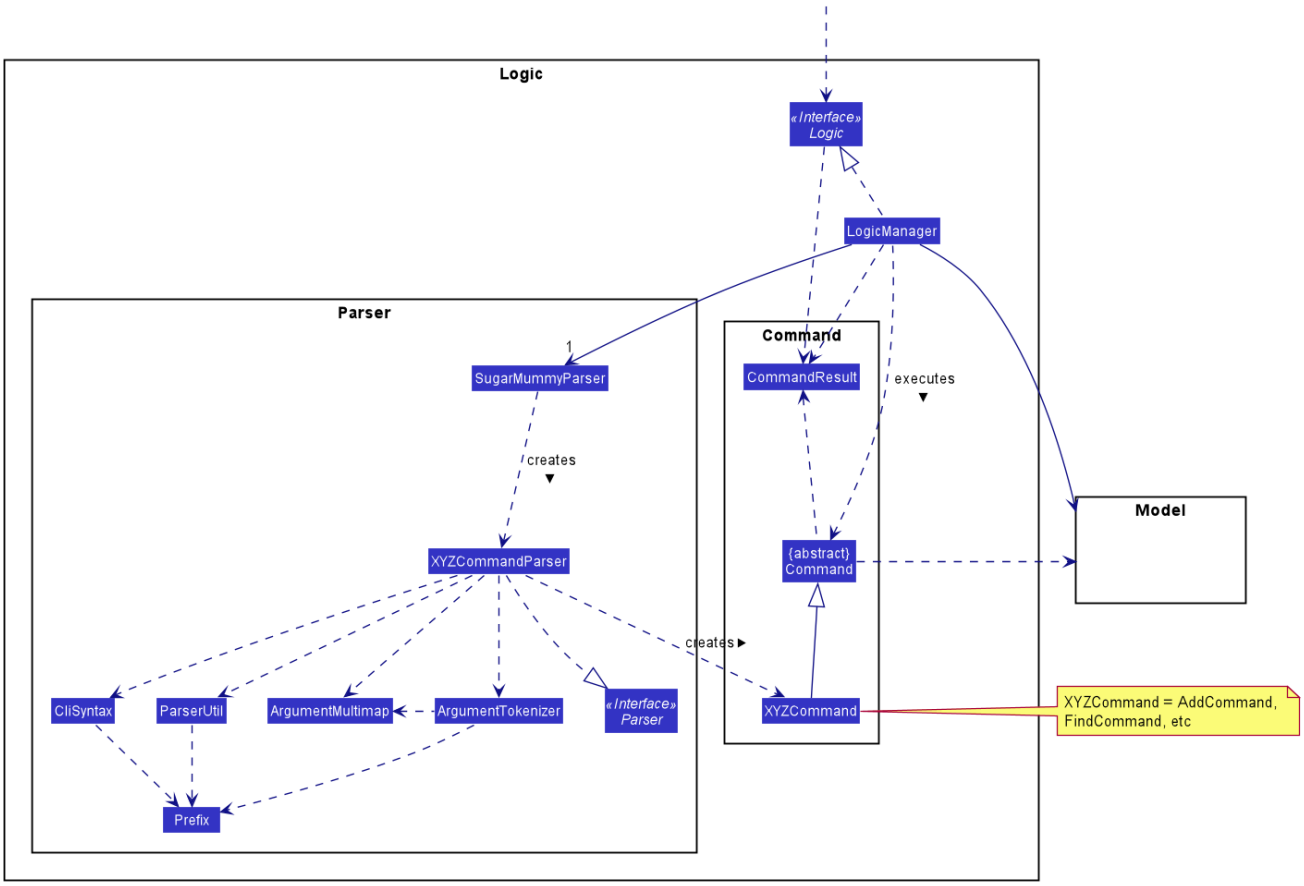


Figure 5. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `AddressBookParser` 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 person).
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.

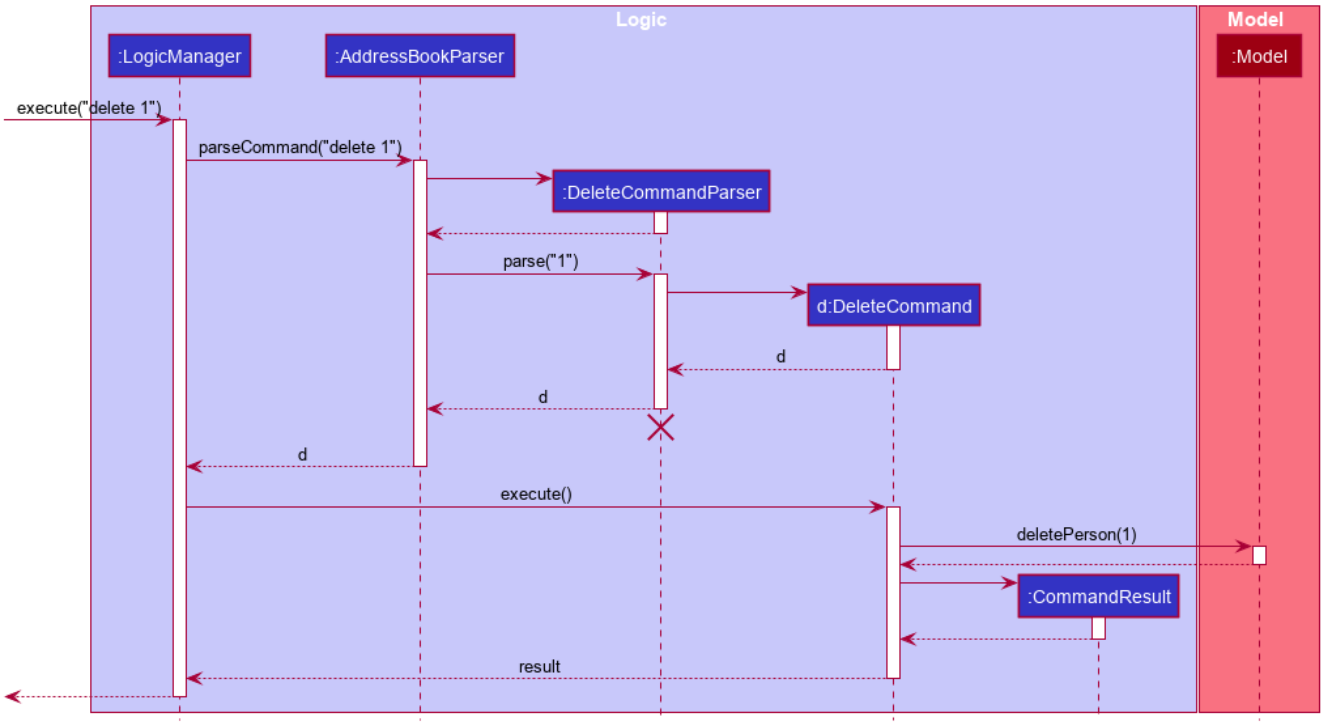


Figure 6. Interactions Inside the Logic Component for the `delete 1` Command

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

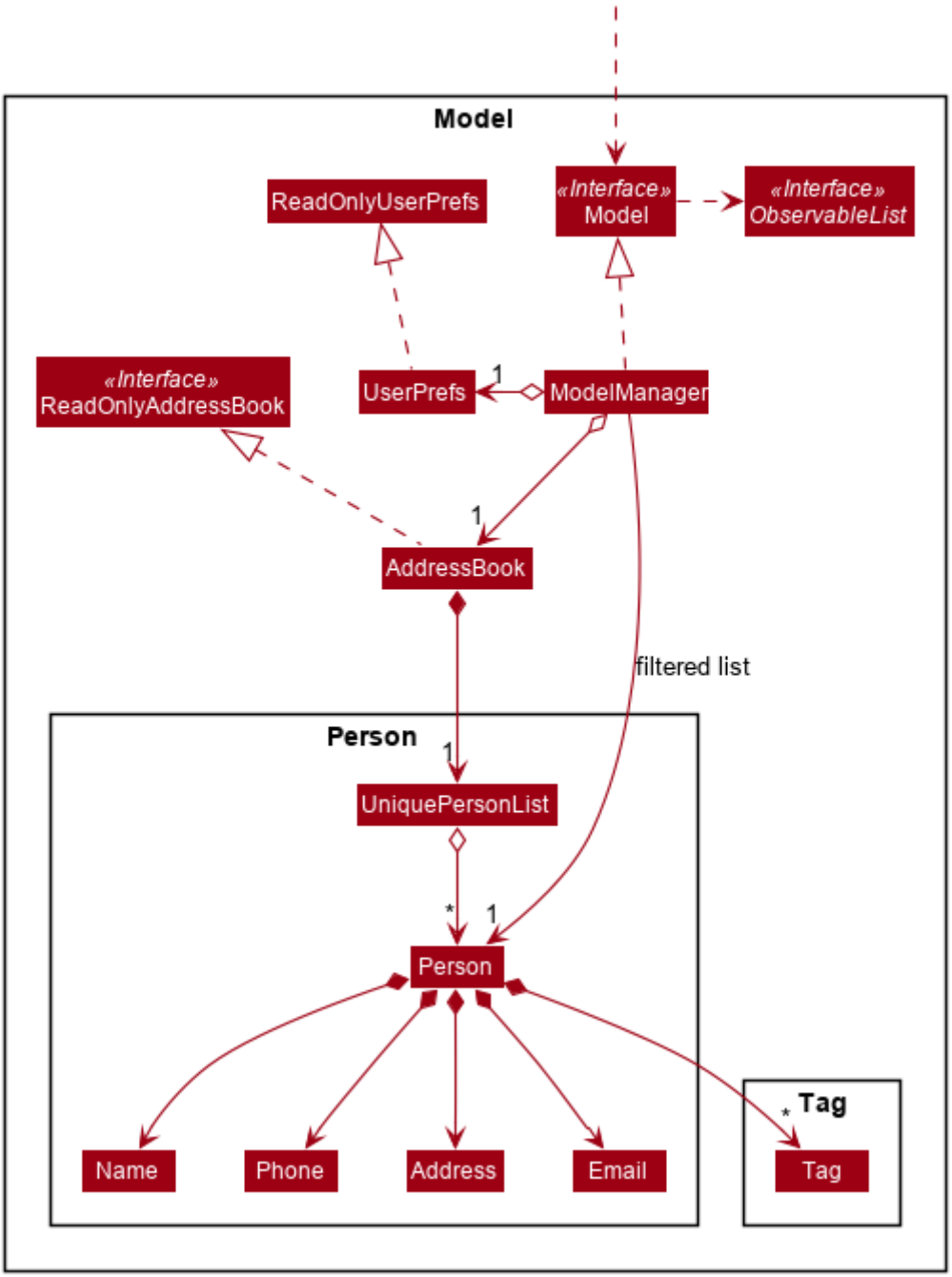


Figure 7. Structure of the Model Component

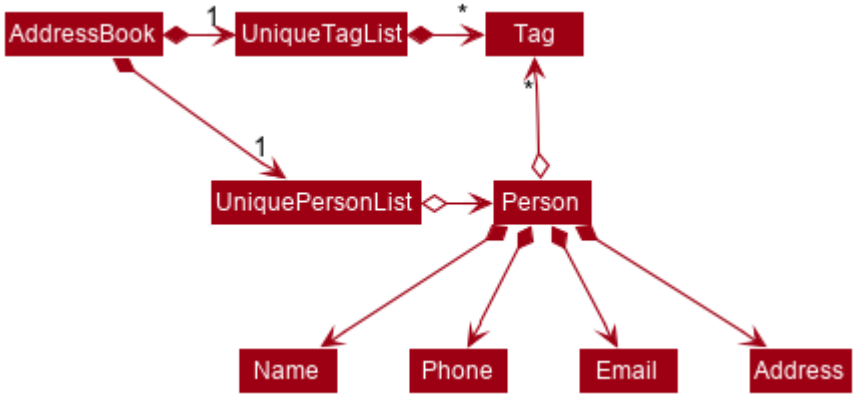
API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user's preferences.
- stores the Address Book data.
- exposes an unmodifiable `ObservableList<Person>` 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.

NOTE

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



2.5. Storage component

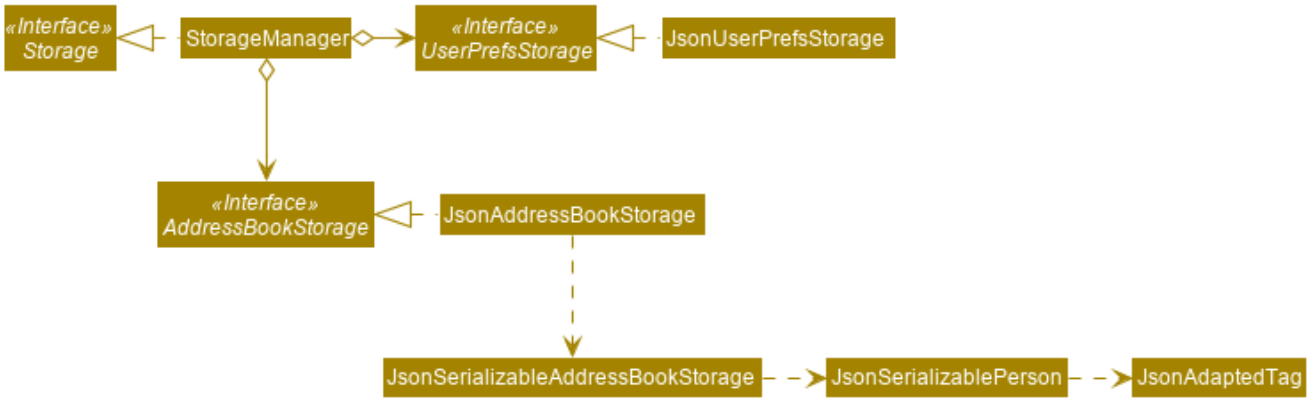


Figure 8. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

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

2.6. Common classes

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

3. Implementation

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

3.1. Data Sumamry/Analysis feature

3.1.1. Average graph feature: Displays the daily/weekly/monthly average of records in a line graph: `average`

The average graph shows how the average blood sugar level or BMI of users change over time. Daily, weekly, monthly average are supported.

Implementation

User input to get average graph is parsed by `SugarMummyParser` which creates a new `AverageCommandParser`. `AverageCommandParser` then parses user input and creates a new `AverageCommand`. Next, `AverageCommand` performs operations on `AverageMap` in `Model` with the help from `RecordContainsRecordTypePredicate` to filter `UniqueRecordList` in `Model`. The result of the execution is returned to `Ui` as a `CommandResult` object and is displayed to the user. In addition, `Ui` calls and displays average graph related `.fxml` file to the user.

The average graph data points generation is implemented by `AverageMap` and the average values are stored internally as `internalMap`. Additionally, it implements the following method:

- `AverageMap#calculateAverage()` - calculates and stores the average values needed by `AverageCommand`.
- `AverageMap#asUnmodifiableObservableMap()` - returns a read only version of `internalMap`.

These operations are exposed in the `Model` interface as `Model#calculateAverageMap()` and `Model#getAverageMap()` respectively.

Example Usage Scenario

Below is an example usage scenario and how average graph is created.

Step 1. User launches the application for the first time. The `AverageMap` will be initialized and `internalMap` will be empty.

Step 2. User enters `average a/daily rt/bloodsugar n/4` in SugarMummy to get daily average blood sugar. Input is parsed and send to `AverageCommand`. `AverageCommand` then calls `Model#updateFilteredRecordList` to filter record list with `RecordContainsRecordTypePredicate`. This results in a list of records containing only blood sugar records. Subsequently, `AverageCommand` calls `Model#calculateAverageMap()` to update the `internalMap` to store 4 most recent daily average values by using the filtered record list.

The following sequence diagram shows how the average operation works:

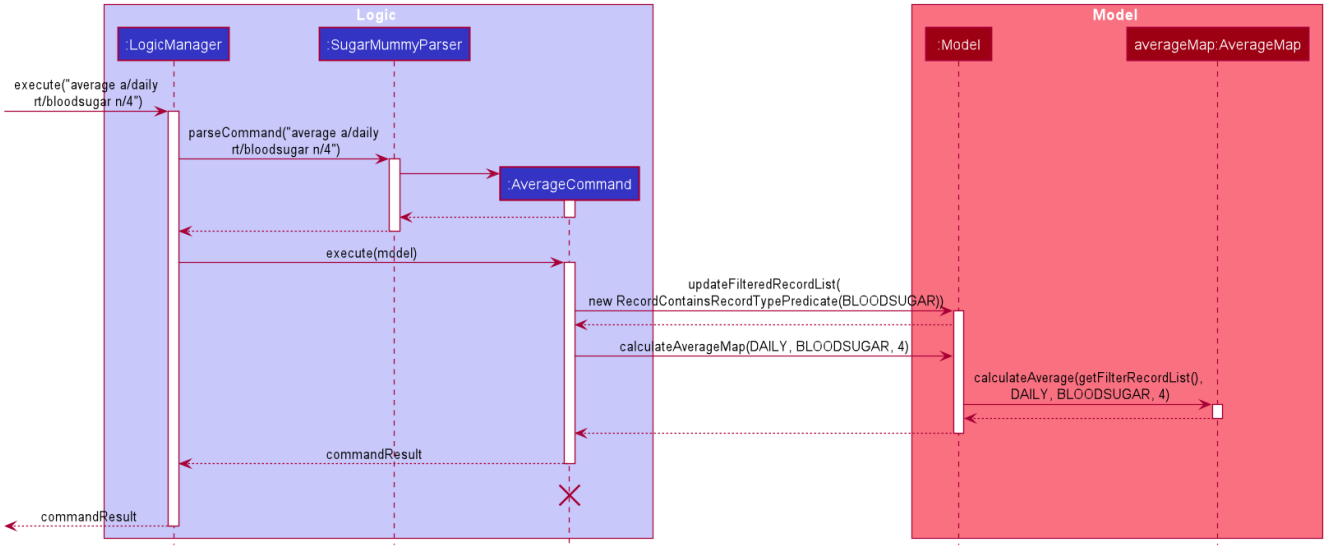


Figure 9. Sequence diagram of how average command calculates average values.

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

Step 2a. If the user enters `average a/daily rt/bloodsugar n/4` and there is no data available, then the command will fail to execute and throw a `CommandException`. Alternatively, if user enters an invalid command, a `ParseException` will be thrown. This is illustrated in the activity diagram below.

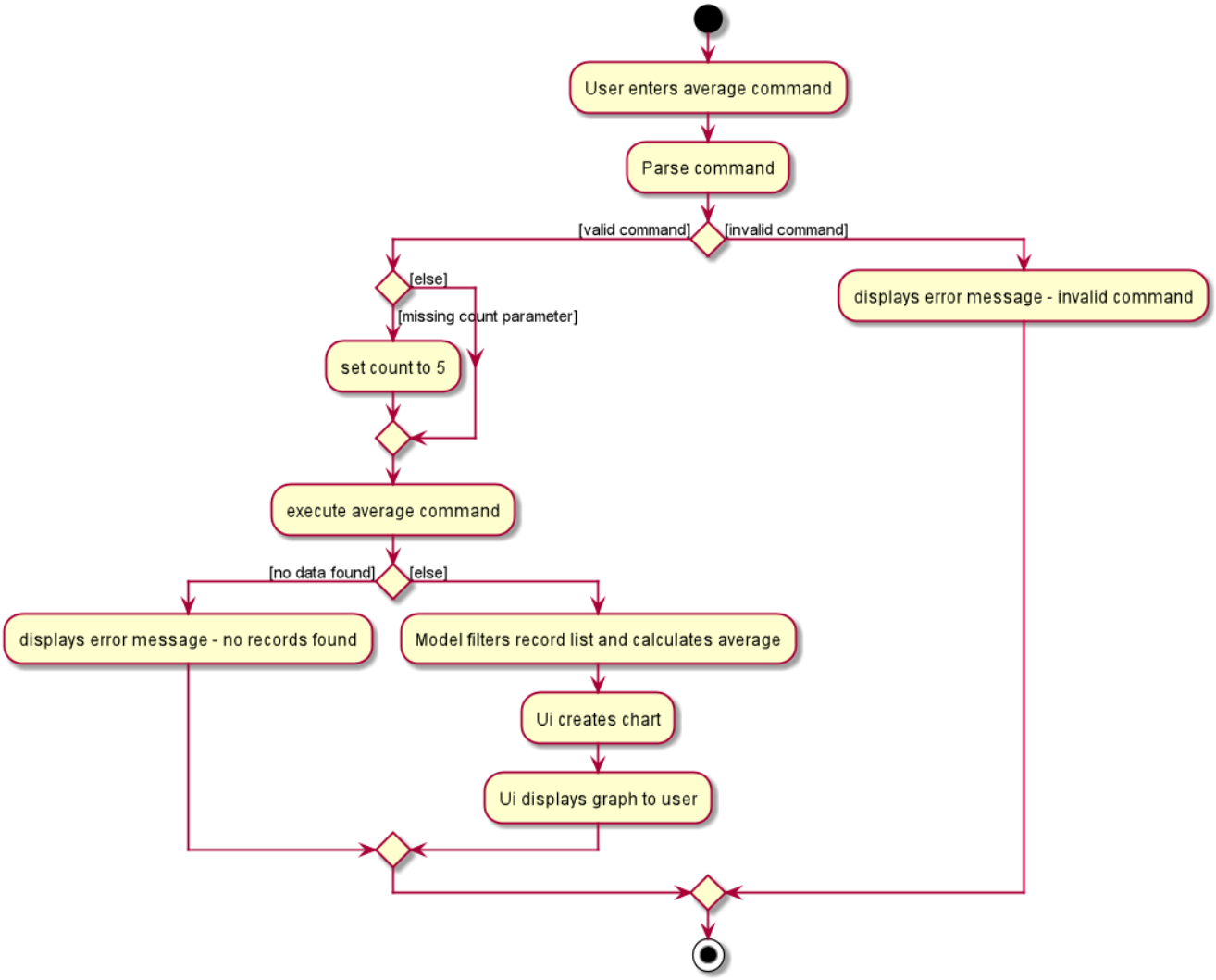


Figure 10. Activity diagram of user enter an average command.

Step 3. `Ui` receives average `CommandResult` from `LogicManager` and creates a new `AverageGraphPane` as well as all other necessary components (see below). `Ui` then displays the `AverageGraphPane` to user.

Average graph Ui consists of several parts:

- `AverageGraphPane`: Placeholder for `AverageGraph` and `LegendPane`.
- `AverageGraph`: Contains the average graph. Data points are generated by `internalMap`.
- `CustomLineChart`: The implementation for average graph which extends and override JavaFx `LineChart`.
- `LegendPane`: Placeholder for `LegendRow`. This is the legend box for average graph.
- `LegendRow`: Consists of a colored legend symbol and its description.

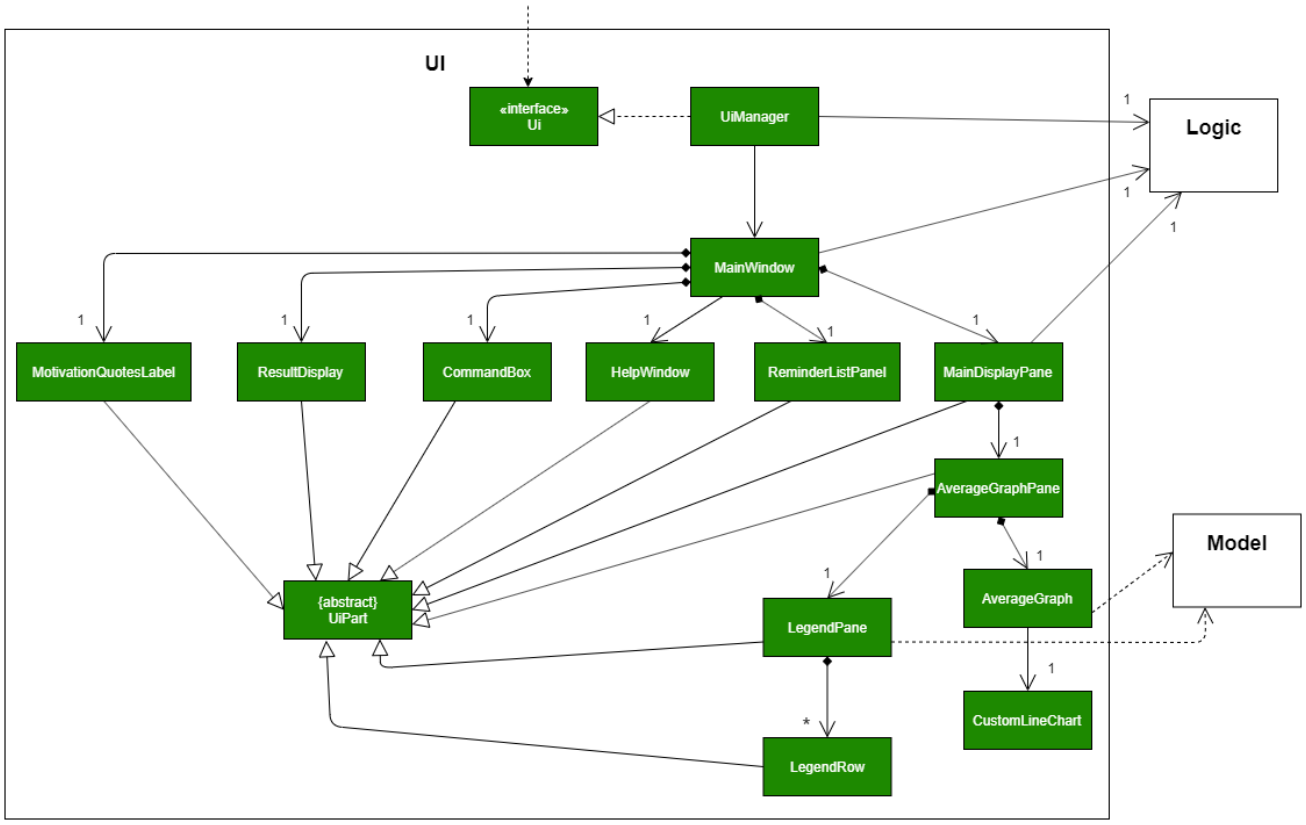


Figure 11. Class diagram of average graph ui related classes.

Design Considerations

Aspect: How to display average graph to user.

The dilemma arises because users, especially recently diagnosed type 2 diabetics, do not know the normal range of BMI and blood sugar level. An intuitive and aesthetically pleasing method is needed to convey this information to user.

- **Alternative 1:** Use JavaFx `LineChart` to display the average graph and display the ranges below the graph using JavaFx `Label`.
  - Pros: Do not need to implement anything.
  - Cons: User need to trace data points to the y axis to find it's value and compare it with the ranges given below the graph. This can be annoying and tedious for the user.
- **Alternative 2 (current choice):** Override JavaFx `LineChart` by adding horizontal range markers to the graph and color the area between the markers.
  - Pros: User is able to tell which range a particular data point falls in immediately.
  - Cons: Need to implement horizontal range markers and lay it out on the graph. In addition, a custom legend box is needed to label the horizontal range markers.

3.1.2. Data Summary/Analysis Feature `coming in v2.0`

[Proposed] Summary statistics of a particular record type `[coming in v2.0]`

The implementation will be similar to average graph feature. The `UniqueRecordList` can be filtered

the same way as average graph feature to get a list containing only the specified record type. If only records from a certain time period is needed, a new date predicate class needs to be created to further filter the `UniqueRecordList` by starting and ending date. Using the filtered record list, count the number of low, normal and high values based on some threshold set by the developer. These counts will then be displayed using JavaFX `PieChart`. Also calculate the minimum, maximum and average of the filtered record list. These 3 statistics will be displayed right under the pie chart as plain text.

[Proposed] Shows relationship between record types: `[coming in v2.0]`

The implementation will be similar to average graph feature. But now, `UniqueRecordList` needs to be filtered so that it only contains the two record types needed. To do this, future developer need to tweak the current `RecordContainsRecordTypePredicate` to be able to filter two record types.

NOTE	Since SugarMummy only supports two record types now, filtering `UniqueRecordList` is redundant. However, this implementation consider the situation that more record types may be added in the future.

Using the filtered record list, pair two different record types from the same day together and this pair represents a data point. Discard records that cannot be paired. Once the pairing process finishes, display the points in JavaFX `ScatterChart`.

[Proposed] Exports summary of all medical records into pdf `[coming in v2.0]`

This feature can be implemented using `PDFBOX` libraries or any other existing libraries.

3.2. Food Recommendation Feature

The food recommendation mechanism is based on the manipulation on `UniqueFoodList`, with the implementation of operations:

- **Showing filtered food cards** — Shows recommended foods to the users, which are filtered by `Flags` and / or `FoodNames`.
- **Sorting the food list** — Sorts the recommendation order based on comparing food fields specified in `SortOrderType`.
- **Showing concise recommendations** — Recommends one food from each food type with an additional *summary* food.
- **Adding new a food** — Adds a new food to the food database and for future recommendations.
- **Resetting food database** — Clears user-added foods.

These operations are respectively exposed in the `Model` interface as `Model#updateFilteredFoodList()`, `Model#sortFoodList()`,`Model#getMixedFoodList`, `Model#addFood()`, `Model#setFoods()`.

3.2.1. Overview of Data Structures

The main data structures used to support food recommendation feature are **Food Model**, **UniqueFoodList**, and **Predicates**

1. Food Model

- **Food Model**: holds data for a certain food
- Predicates: indicates the filters that the user wants to apply on partial presentation of the foods
- Unique Food List: holds the collection of all foods

API: `Food.java`

---

2. UniqueFoodList

The `UniqueFoodList` is the main model that contains all the foods and interacts with logic and UI. It exposes an unmodifiable `ObservableList<Food>` that associates the UI display of food recommendations.

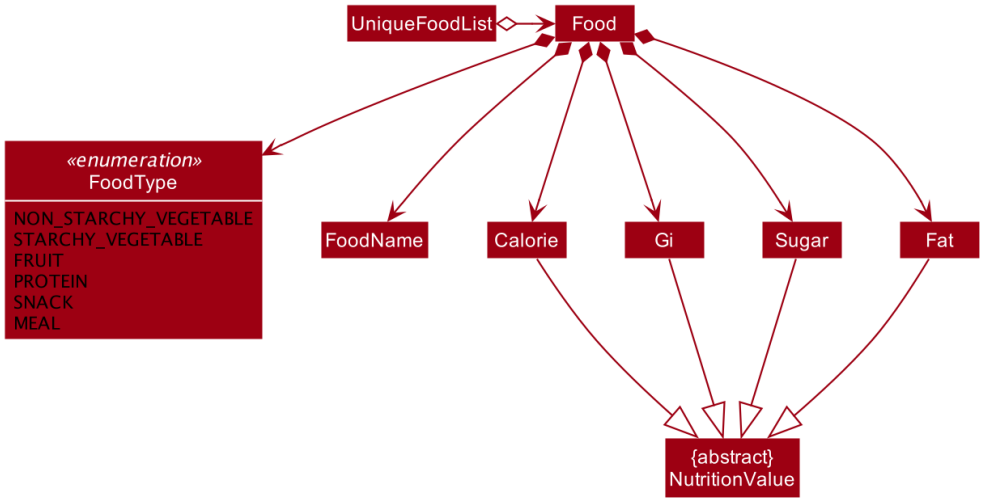
API: `UniqueFoodList.java`

3. Predicates

API: `FoodNameContainsKeywordsPredicate.java` `FoodTypeIsWantedPredicate.java`

The following class diagram shows the main association and interactions among the main components. Other essential parts are summaries as follows:

- UI: `FoodFlowPanel` holds an `ObservableList` of `Food`, each visulized in the form of `FoodCard`.
- Storage: Reading data from and writing data to is handled by `JsonFoodListStorage`, inherited methods from `JsonGeneralStorage`.



3.2.2. Implementation of *recmf* and *recmfmix* command

***recmf* command**

The `recmf` command visualizes medically suggested foods as food cards for diabetics that are contained in `UniqueFoodList`.

The customised presentation of food recommendations is implemented via the following three ways:

1. Specifying flags

Flags are similar to the usage of flags / options in Unix commands. In SugarMummy, they are used to specify food types that are intended to be shown. The existence of a certain flag depends on available food types in `FoodType`. The flags in the user input will eventually be translated to `FoodTypeIsWantedPredicate` and applied on `UniqueFoodList`.

Data Structure

A `HashSet` is used to hold specified `FoodTypes` translated from user-input flags.

NOTE	If no flag is specified, `RecmFoodCommandParser#getWantedFoodTypes(flagsStr)` will return an empty `HashSet`.

API: Flag.java

---

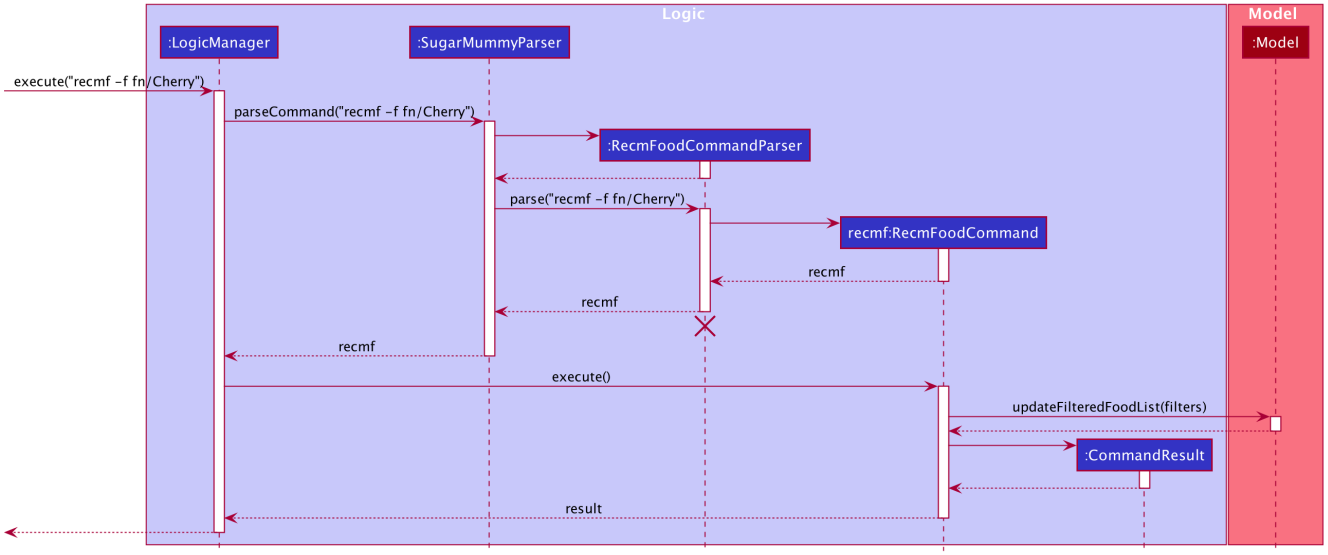
2. Filtering food names

A `List` of strings as food names will be supplied to `FoodNameContainsKeywordsPredicate`. Similar to the situation of no specified flags, an empty `ArrayList` will be returned by `RecmFoodCommandParser#getWantedFoodKeywords(namesStr)`.

The implementation is simple, details about `FoodNameContainsKeywordsPredicate` and `RecmFoodCommandParser` can be referred at Overview section.

Diagram

The following sequence diagrams shows the how recmf command with food name as the filter works.



---

3. Sorting

Sorting the food recommendations is via supplying a `FoodComparator` to `UniqueFoodList#sortFoods`. To maintain or change the ordering of food recommendations, `model#sortFoodList` method must be added to `execute` method of relevant commands.

NOTE	The default sort order is currently set to be by food type, specified in public variable `FoodComparator#DEFAULT_SORT_ORDER_STRING`. It is used when there is no specified sort type from the user and for the `addfood` command.

Data Structure

- A `Comparator` is wrapped by `FoodComparator` to handle the main logic, such as reversing the `FoodComparator` via `Comparator#reversed()`.
- A inner enum class SortOrderType` is used to specify all the available food fields for comparison and sorting. (See Food model for its field details.)

NOTE	Instantiating `FoodComparator` by other classes is done by supplying `String` that matches one of the enum value of its inner enum class `SortOrderType`. Instantiate FoodComparator directly from `Comparator` is for internal usage of getting reversed `FoodComparator`.

API: `FoodComparator.java`

Diagram

The following object diagram summaries the components in food recommendation mechanism.

recmfmix **command**

Compared to `recmf` command with customized options, `recmfmix` *is a simpler command that concisely recommends one food from each type with a summary food card at the end.*

General: Randomly selecting foods is implemented by `UniqueFoodList#getMixedFoodList()` that generates a separate and temporary `ObservableList` from the existing food data of `UniqueFoodList`. This list of mixed foods will be accessible by the `Model` and will be further used by the `Logic` to fill the content of `FoodFlowPanel`.

Food Summary Card: It is essentially treated as `Food` with *Summary* as food name and *meal* as food type. The total / average nutrition values are calculated by `FoodCalculator`.

NOTE	This command has to override the `Command#isToCreateNewPane()` to return a `true` value, since it should refresh the display pane each time by randomly getting new foods, rather than getting the existing display pane from `typeToPaneMap`.

Diagram

The following sequence diagram shows how recmfmix operation work.

API: `FoodCalculator.java`

3.2.3. Implementation of other supplementary commands

The following two commands are designed to help expand and clean up database of foods.

addfood **command**

`addfood` _command adds a new food with all specified fields into the food list.

It is implemented by using `AddFoodCommandParser`, which relies on `RecmFoodParserUtil` to check the validation of input values.

API: `RecmFoodParserUtil.java`

resetf **command**

`resetf` _command clears(deletes) all newly added foods from the user.

It is implemented by setting the internal list of `UniqueFoodList` to be the pre-loaded food data in `SampleFoodDataUtil`.

3.2.4. Example Usage Scenario and Summary

Given below is an example usage scenario and how the food recommendation mechanism behaves at each step.

1. The user launches the application and enter `recmf` command for the first time.
    a. The `foodlist.json` storage file will be created and written with the pre-loaded food data from `SampleFoodDataUtil`.

b. The `UniqueFoodList` will indirectly (via `Model`) supply a list of foods to `FoodFlowPane` to display.

2. The user executes `recmf -f -sv fn/Chicken` command.

   a. `RecmFoodParser` will parse the flag string as *fruit* and *starchy vegetable* for `FoodTypeIsWantedPredicate` and the food name string as *Chicken* for `FoodNameContainsKeywordPredicate`. Meanwhile, `FoodComparator` will be of default sorting order

   b. They will be supplied to instantiate a `RecmFoodCommand` to update the display of food recommendations.

3. The user executes `recmfmix` command.

   a. The `UniqueFoodList#getMixedFoodList()` method will return a list of randomly selected foods from each food type.

   b. A summary food with calculated value by `FoodCalculator` will be appended at the end.

4. The user feels the database is insufficient and wants to add a new food by executing `addfood fn/Cucumber ft/nsv ca/15 gi/15 fa/0 su/1.7`.

   a. The display will switch to show the full list containing the newly added food in the default sorting order.

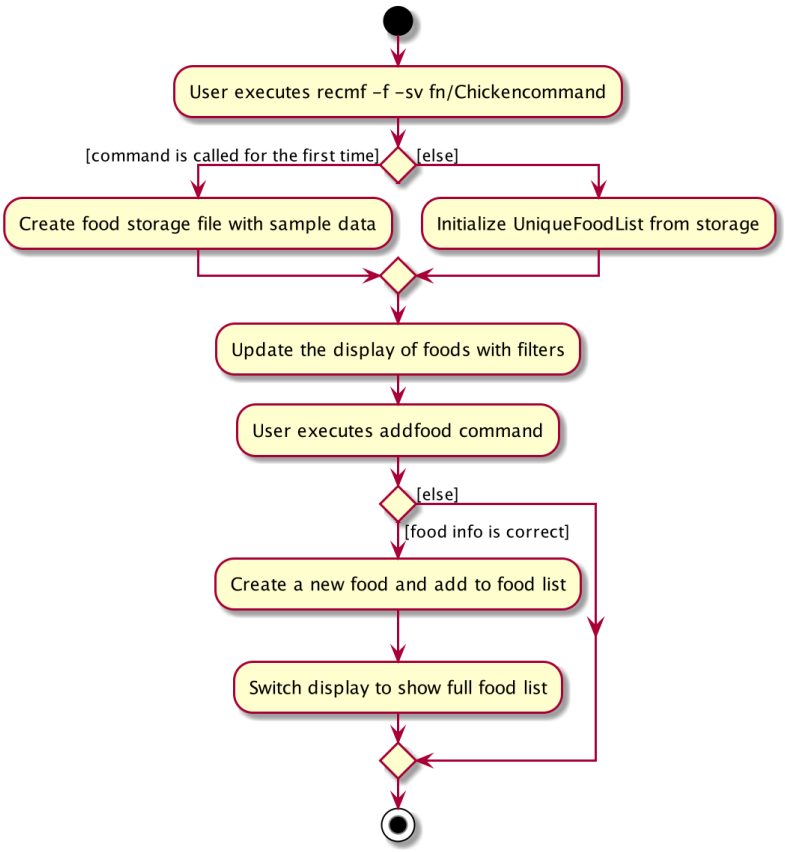
   b. The `foodlist.json` storage file will be immediately updated with this new food.

5. The user executes `resetf` command.

   a. `UniqueFoodList` will reset its internal list to hold the sample data from from `SampleFoodDataUtil`.

Diagram

The following activity diagram summarizes the above steps.



3.2.5. Design Considerations

Aspect: Data Structure of the Food Collection

- **Alternative 1 (current choice):** Use a `List` to store all the foods
  - Pros: The logic can be easily understood.
  - Cons: Filtering, sorting, and adding new foods need to enumerating through the whole list.
- **Alternative 2:** Use a `Map` that categorizes foods based on their food types
  - Pros: For the `Flag` filtering, it can simply get the wanted types from the `Map`. Besides, maintaining the order after adding a new food only requires to sort foods of the same type. It can improve efficiency especially the database is large.
  - Cons: There is no `FilteredMap` class supported by JavaFX. Thus, additionally structures needs to be defined to accept `Predicate` as filters.

Aspect: The presentation of food recommendations (UI)

- **Alternative 1 (current choice):** Show the user a pane of cards. Different types are indicated by the different background colors of the food names.
  - Pros: Easy to implement. The usage of cheerful colors may make reading recommendations more pleasant.
  - Cons: The size of food cards cannot be customized. If the window size is relatively small, the user may need to repeatedly scroll up and down to locate some foods.
- **Alternative 2:** Use several horizontal `ListViews` to hold different food type.

- Pros: The content is more organized and the user does not need to specify food types in the filter. Besides, the food card for different food types can be more targeted. For example, for most proteins, the sugar and gi of value 0 can be omitted while protein values can be added.
- Cons: The operations targeting at the whole list, such as filtering based on food names, need to be applied repeatedly for each food list.

Aspect: Inputting New Food Data

- **Alternative 1 (current choice):** Require inputs for all fields (e.g. calorie, gi…).
  - Pros: It is easy for data manipulation. Specifically, this prevents some foods from permanently having empty fields. Additionally, this may further hinder the data usage for data analysis.
  - Cons: Some data may not be currently available while the user still to want to record a new food by simply inputting the food name.
- **Alternative 2:** Allow temporarily empty fields and use a separate list to hold such incomplete inputs.
  - Pros: This provides the user with more freedom and flexibility of entering data.
  - Cons: Every change or manipulation on food data needs to be applied on two lists. Transferring data from one list to the other may also be error-prone.

3.2.6. Future Development Suggestion

- Recovering data after resetting This would be useful if the user wrongly enter `resetf` command, or another user want to temporarily use the same jar file on the same PC.
- Editing Foods This would provide more flexibility to the user to manipulate food data, instead of resetting all the food data.
- Disliking Foods This would prevent the user from repeatedly seeing the foods they dislike, cannot eat (due to religion reason), or are allergic to.
- Expanding the food database This would relieve the extra work required from the user to input unavailable food data. Ideally, the recommendation data can be connected to online database for dynamic updates while can be stored locally for offline operations.
- Recording and Analyzing diets This would allow the user to have an overview of his food consumption statistics. Bases on such statistics, more specific suggestions can be proposed to to balance the user's nutrition intake.

3.3. Data log feature

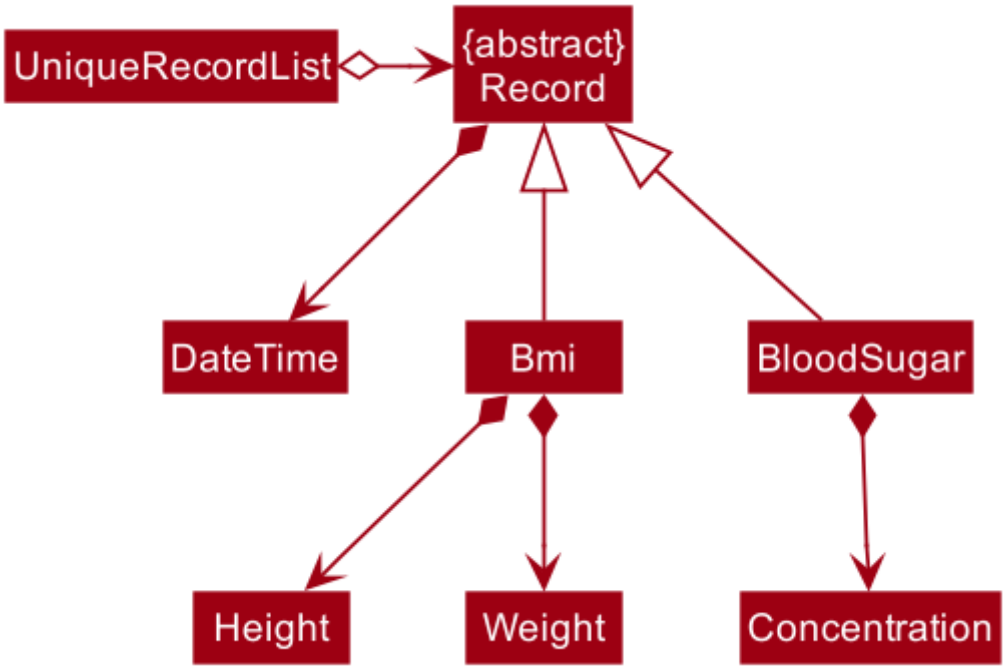
3.3.1. Implementation

The multi-record data logging mechanism is facilitated by a new Record package containing BloodSugar and Bmi classes that extend an abstract Record class. Add, list and delete command classes and parsers are modified to accommodate multiple record types. Multi-record data is stored internally as a recordList where members are Records.

It modifies the following operations: * `SugarMummy#add()` — Adds a record to the record list. * `SugarMummy#delete()` — Deletes a record from the record list. * `SugarMummy#list()` — Retrieves all records in record list.

These operations are exposed in the `Model` interface as `Model#addRecord()`, `Model#deleteRecord()` and `Model#getUniqueRecordListObject()` respectively.

The internal data structure contains an ObservableList<Record> that the UI can observe to display the record list.

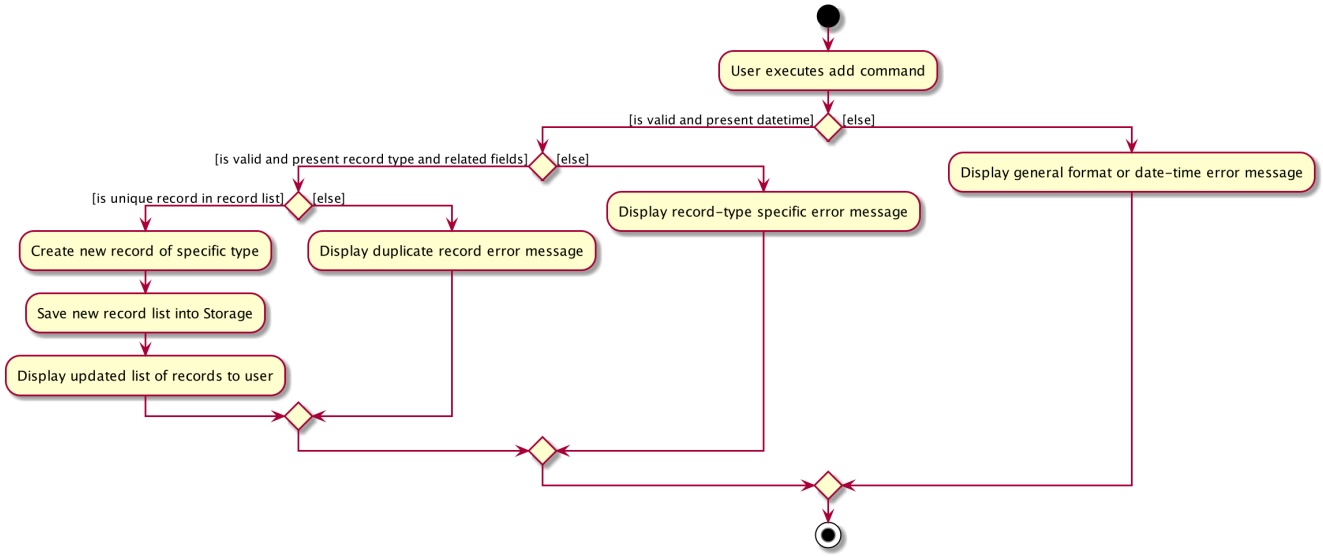


Below is an example usage scenario of how the data log feature behaves at each step.

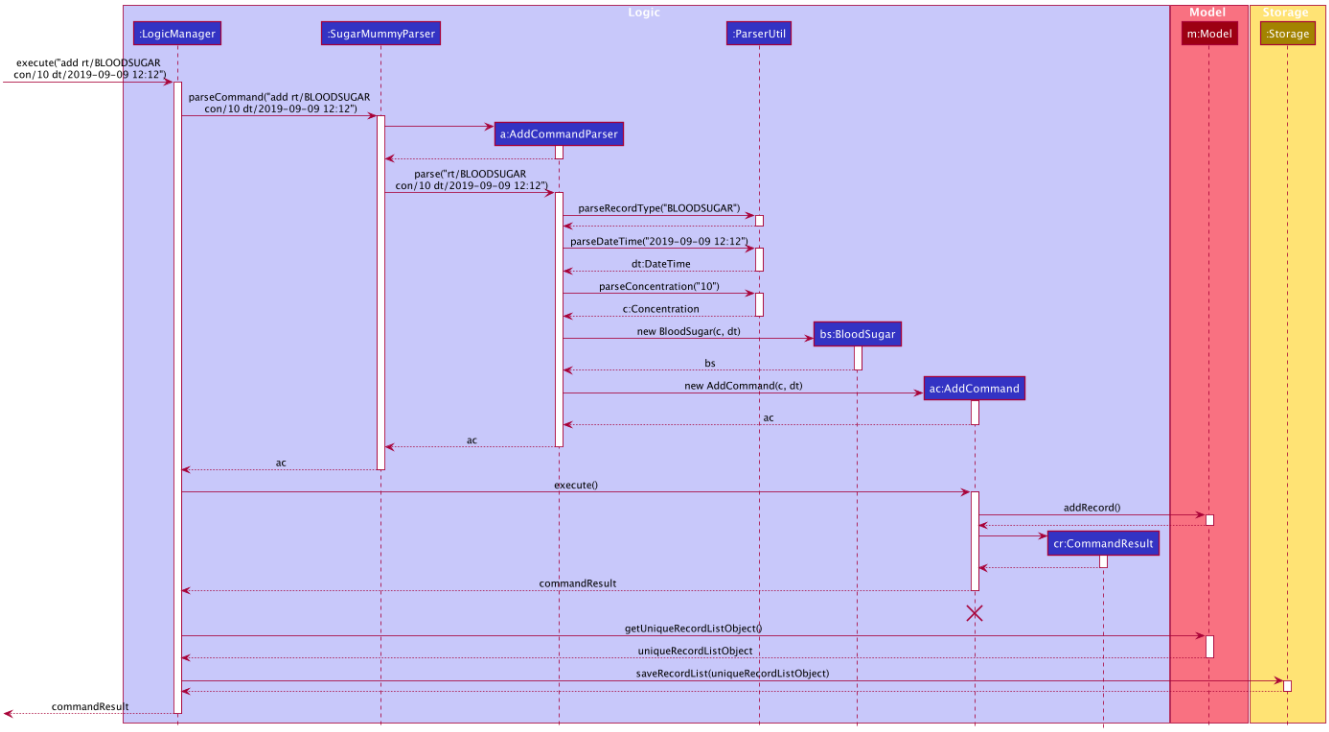
Step 1. The user launches the application for the first time. If `/data/recordList.json` is not found, it will be produced from `SampleRecordDataUtil#getSampleRecords()`. If `/data/recordList.json` is found, the recordList will be loaded from there using `UniqueRecordList#setRecord()` and checked for inconsistencies e.g. missing fields, invalid fields. If inconsistencies are detected, an Exception is thrown and existing `recordList.json` is erased.

Step 2. The user lists all records. A new `RecordListPanel` is created and populates each `ListViewCell` with `BloodSugarRecordCard` and `BmiRecordCard`. `ObservableList<Record>` is used to populate the `ListViewCell`.

Step 3. The user executes `add rt/BMI h/1 w/1 dt/2019-09-09 12:12` command. The add command parameters are parsed for validity. This job is delegated to the following classes: `SugarMummyParser` ,`AddCommandParser` and `ParserUtil. This is illustrated in SequenceDiagram below. After parsing is completed, either a complete `BloodSugar` or `Bmi` Object is returned otherwise a `ParseException` will be thrown. The `Record` is checked against the model for uniqueness. If it is unique, it is added to the Model via `Model#addRecord()` (illustrated by the red portion of the sequence diagram below)



The above activity diagram illustrates step 3.



The above sequence diagram provides a in-depth look at how parsing is delegated to various classes within the blue Logic component. The calls to the red model component illustrates Step 3 adding records to the model. The final call to the yellow storage component illustrates step 5.

Step 4. The user decides to delete a record. The delete command is parsed for validity by `SugarMummyParser`, `DeleteCommandParser` and `ParserUtil`. `ParseUtil` checks whether the index is a positive number, otherwise a `ParseException` will be thrown. `DeleteCommand` checks whether the positive index points to a valid record. `DeleteCommand` will call `Model#deleteRecord()` to remove the record from the list.

Step 5. After add or delete commands have been executed in `LogicManager`, the Model's recordList is written to `recordList.json` using `Storage#saveRecordList()`.

3.3.2. Design Considerations

3.3.3. Aspect: Commands and parsers implementation

- **Alternative 1 (current choice):** Parse for new record type X within existing add, list, delete commands and their parsers
  - Pros: Easy to implement as long as record type X inherits from Record. AddCommand, ListCommand and DeleteCommand classes remain very similar to their original implementations.
  - Cons: Harder to debug when parsing fails because XCommandParser classes are responsible for checking for presence of multiple fields of multiple record types. Parsing may become complicated if the order of parsing fields becomes important.
- **Alternative 2:** Create separate AddX, ListX, DeleteX, AddXParser, ListXParser, DeleteXParser for each new record type X introduced
  - Pros: Each parser is responsible for parsing only record type X's fields. This narrows down the scope of debugging should parsing fail.
  - Cons: Accommodating a new record type involves creating at least 6 additional classes just for operations on data classes. Data classes required to represent the data include: Bmi class with Height and Weight class.

3.3.4. Aspect: Data Structure for managing multiple record types

- **Alternative 1 (current choice):** Use a single list to store multiple record types.
  - Pros: Easy for new Computer Science student undergraduates to understand, who are likely to be the new incoming developers of our project. Simpler to implement.
  - Cons: Must conduct type checks when retrieving from list. When a new record type is added, all type checks in different places must be updated.
- **Alternative 2:** Use separate lists to store different record types.
  - Pros: Do not need to perform type checks when retrieving from list.
  - Cons: Listing all records together becomes difficult, must build a new list from all separate lists. Each class must reference a different kind of list.

3.4. 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 3.5, "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

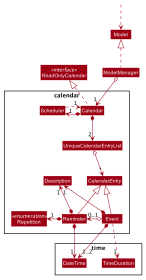
3.5. Configuration

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

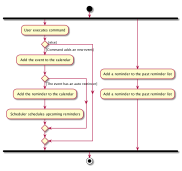
3.6. Calendar feature

3.6.1. Implementation

The calendar feature works along with a scheduler. Calendar stores internally a `calendarEntries` list and a `pastReminders` list. Calendar entries consists of reminders and events. Past reminders list is dynamically determined by time and all reminders in this list is shown to the user. The scheduler acts as a wrapper for the Java ScheduledExecutorService, which updates past reminder list at specific time. See the class diagram below for calendar related classes.



The following activity diagram shows how event command and scheduler works:



It implements the following operations:

- 'Calendar#addCalendarEntry' — Adds a new calendar entry to the calendar.
- 'Calendar#addPastReminder' — Adds a reminder to the past reminders list.
- 'Calendar#getCalendarEntryList' — Gets a list of calendar entries.
- 'Calendar#getPastReminderList' — Gets a list of past reminders.
- 'Calendar#schedule' — Schedules a series of upcoming reminders.

These operations are exposed in the `Model` interface as `Model#addCalendarEntry()`, `Model#addPastReminder()`, `Model#getFilteredCalendarEntryList()`, `Model#getPastReminderList` and `Model#schedule` respectively.

Given below is an example usage scenario and how the calendar behaves at each step.

Step 1. The user launches the application for the first time on Dec 14 2019 09:00(local time). The `Calendar` will be initialized with the initial calendar state, which includes an empty calendar entry list and an empty past reminder list.

Step 2. The user executes `reminder d/insulin injection dt/2019-12-14 17:30 r/daily` command to add a new reminder of 'insulin inject' at 17:30 every day. The `reminder` command calls `Model#addCalendarEntry()`, causing the modified state of the calendar after the reminder command executes to be saved in the `calendarEntries` list. Subsequently, it calls `Model#schedule()` which forces the scheduler to update the upcoming reminders.

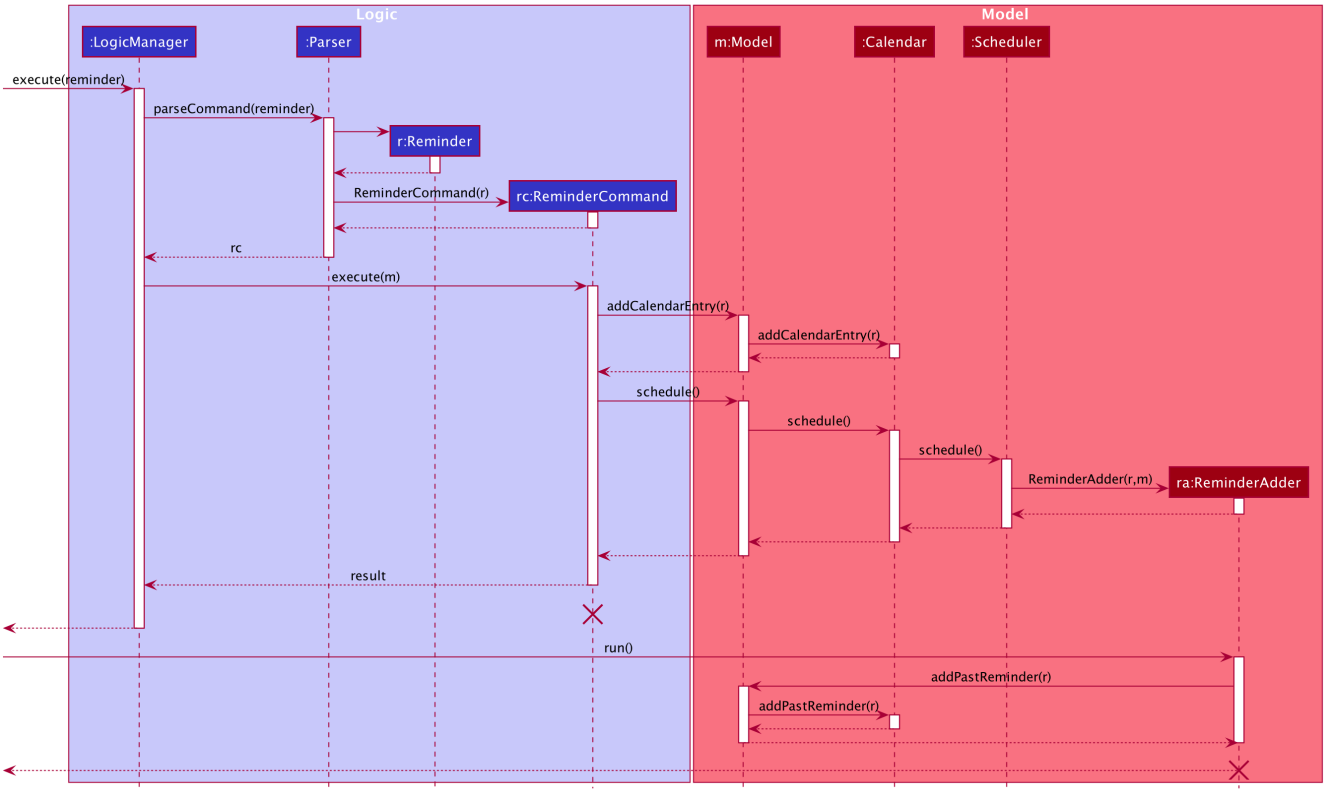
Step 3. The user executes `event d/meeting dt/2019-12-14 14:30 tp/00:30` command to add an new event with an auto reminder scheduled 30 minutes before the event. It calls `Model#addCalendarEntry()`, causing a new event as well as a new reminder saved in the `calendarEntries` list. Subsequently, it calls `Model#schedule()` which forces the scheduler to update the upcoming reminders.

NOTE	If an event or reminder command fails its execution, it will not call `Model#addCalendarEntry()`, so the calendar state will not be saved into the `calendarEntryList`.

Step 4. At 14:00, a scheduled task is executed to call `Model#addPastReminder()` and it adds the dinner event reminder to the `pastReminders` list.

Step 5. At 17:30, a scheduled task is executed to call `Model#addPastReminder()` and it adds the dinner event reminder to the `pastReminders` list.

The following sequence diagram shows how a single `reminder` command works:



3.6.2. Design Considerations

Aspect: How scheduler updates upcoming reminders

- **Alternative 1 (current choice):** Cancels all scheduled reminders and reschedule according to the updated reminder entries.
  - Pros: Easy to implement.
  - Cons: May have performance issues in terms of time.
- **Alternative 2:** Updates scheduled reminders according to the newly added reminder.
  - Pros: Will has less repeated work.
  - Cons: More work to do on deciding which tasks to cancel.

Aspect: Period of updating scheduler.

- **Alternative 1 (current choice):** Updates scheduler at 23:59(local time) every day.
  - Pros: Easy to implement.
  - Cons: May have a large number of scheduled tasks which will not be executed before the applicaion is closed.
- **Alternative 2:** Updates scheduler every hour.
  - Pros: More flexible scheduling without concerning date and less scheduled tasks.
  - Cons: May cause overhead due to frequently updating.

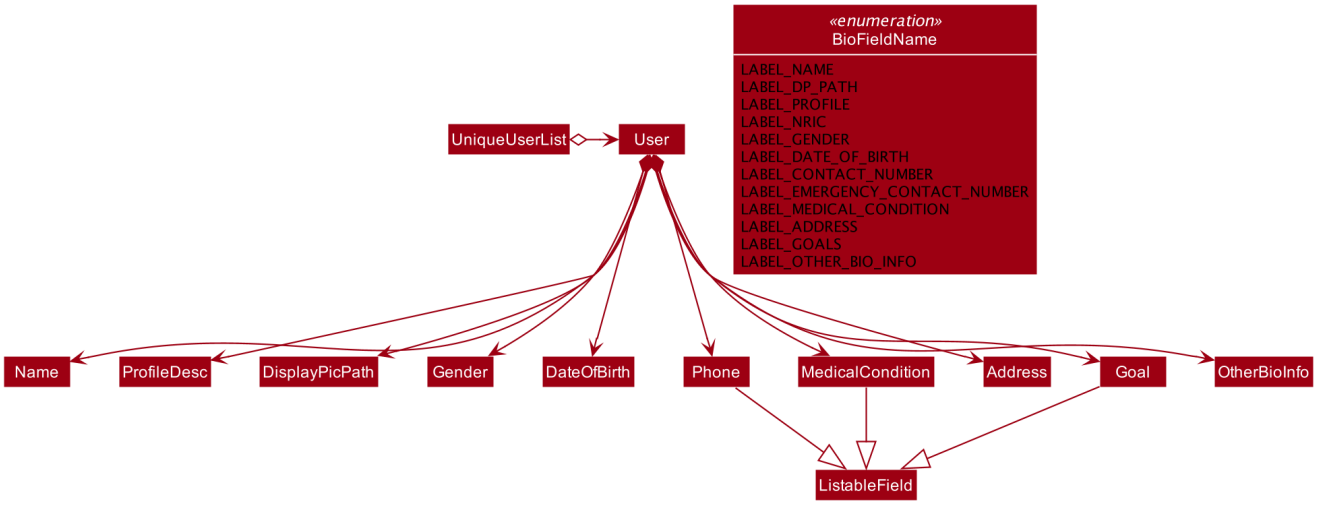
3.7. Personalised User Experience Feature

To personalise the diabetic user's the experience in using the SugarMummy app, several sub-features are used, including:

- Addition, editing and clearing of the user's biography
- Customisation of font and `background` colour, with the ability to set as `background` image for `background` as well.
- Display of motivational quotes for the user (initialisation phase; in progress)
- `Achievements` to be shown to the user upon achieving a milestone.

3.7.1. Overview

- The `User` class is used to represent a diabetic user. A diabetic user is composed of the `Name`, `ProfileDesc`, `DisplayPicPath`, `Nric`, `Gender`, `Phone`, `MedicalCondition`, `Address`, `Goal` and `OtherBioInfo` classes.
- A `User` is currently defined to be able to have more than one `Phone`, `MedicalCondition` and `Goal`. As such, these classes inherit the `ListableField` Interface.
- The structure of a `User` and its interactions are shown as follows:



- A `User` implements `ListableField` by storing them in a java `List`.
- A `User` that is created is added to a `UserList`. Although not more than one `User` can be added in current versions so as to enhance personalisation for the, future developers may decide to repurpose the app to allow more users, and their corresponding biographies represented by the `bio` fields, to the `UserList`.
- Other personalisation features such as `fontcolour`, `background` and `achievements` are currently represented by independent classes `Colour`, `Background` and `Achievement` respectively on their own, representing the model as their name describes.
- The `Colour` feature allows for either enumeration of colour names or hexadecimal colour codes to be used to set colour. `Background` is associated to `Colour` as an argument for `Background` could simply be a colour. It depends on the static method isValid`Colour`(String test) method to determine if it's argument is a `Colour`

- The AddBioParser and EditBioParser is currently used to parse command arguments given by the user and allows adding of specific biography fields, whereas the FontColour and Background parsers are used to parse arguments for other personalisation features for font colours and background respectively.
- The Ui for personalisation is separated into distinct parts. User's biography information and achievements page are components on their own in the Ui's MainDisplayPane – switched when required, whereas background and fontcolour do not have a designated Ui window, but instead changes the attributes for the entire application by modifying the CSS file used itself.
- All command words in this program, not restricted to this feature alone, are not case sensitive and implemented under SugarMummyParser.

3.7.2. Implementation

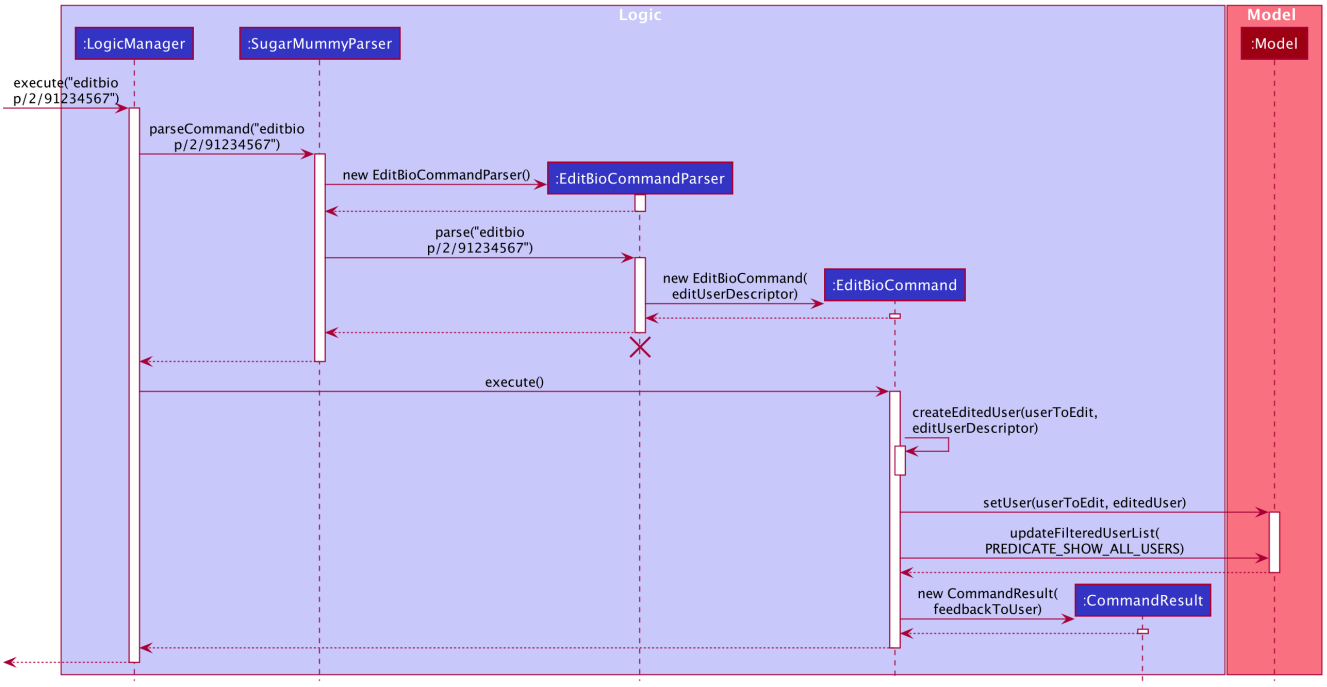
Biography

The biography feature is supported by the addbio, editbio and clrbio commands. Each command adheres to the main flow of information used by this application. In other words, when a command is received, the command is first parsed by SugarMummyParser, and to individual parsers where required, before return a Command object. The Command object is then executed by LogicManager, during which it updates ModelManager, and after which Storage is updated, before feedback from the CommandResult returned by the Command object is shown to the user back at the Ui.

- The following are possible scenarios for each of the following types of command words.
  - Scenario 1: User keys in addbio n/test minimal p/91234567 e/81234567 /test medical condition
  - Scenario 2: User keys in editbio p/2/91234567
  - Scenario 3: User keys in bio
  - Scenario 4: User keys in clrbio
- In all scenarios, SugarMummyParser responds to the command word via a series of switch cases. As mentioned above, addbio and editbio returns AddBioCommandParser and EditBioCommandParser respectively.
- A key difference between the parsers for addbio and editbio is that the former requires Name, ContactNumber, EmergencyContact, and MedicalCondition to be compulsory whereas editbio requires at least one argument denoting the User's biography field to be changed. Furthermore, EditBioCommandParser determines whether or not subarguments for fields of ListableField type contain the format INDEX/, denoting the particular number in the list to be changed.
- CommandParser then returns an AddBioCommand object that stores the User to be created. EditBioCommandParser on the other hand creates an EditBioCommand object that stores an EditedUserDescription containing information on which fields are edited to be edited.
  - A List of HashMaps that maps indices to ListableField is used in EditedUserDescription to denote changes to be made within each ListableField. When executed by Logic afterwards, the AddBioCommand creates the User to be stored in the ModelManager whereas the EditBioCommand creates a new User based on information in EditedUserDescription. A UserList is used in the ModelManager to store User instances.

    - At any point of time when a user attempts to access biography information, `LogicManager` accesses the `UserList` from `ModelManager` to display information. In order to be able to display the same information upon startup, `LogicManager` saves this `UserList` to the storage after execution of each command.

- For the `bio` and `clrbio` commands, the implementations are relatively more straightforward.
    - A `BioCommand` returned by `SugarMummyParser` simply overrides the `getDisplayPaneType()` of the `Command` object (that each `Command` object contains) so that back at `Ui`, `Ui` knows to display the `BioPane` of the `Ui` in the `MainDisplayPane` part of the window.
    - This is also done for all other biography-related commands so after each biography-related command, the `BioPane` is displayed. A `DisplayPane` is stored in the form of an enumeration as the type of display would be predefined to all it's accessors. The `ClearBioCommand` class simply clears the `UserList` stored in the `ModelManager` upon execution.
- In the cases of `bio` and `clrbio` commands, `SugarMummyParser` requires non-null arguments just as it does for other single-word commands such as `exit`.
- Each `Command` returns a `CommandResult` to logic containing feedback to be displayed to the user. Any exception that is thrown to the user is caught back at `Ui` `Ui`. Feedback is displayed to the user using the `ResultsDisplayPane`. The display of user biography is implemented using JavaFX `TableView`. If the `DisplayPicPath` of a `User` is unchanged, the `Ui` does not reload the image, so as to optimise performance of the program. If an entire pane is left unchanged, the pane is not reloaded, even upon execution of commands that are used to display the pane, unless explicitly indicated in the `getNewPaneIsToBeCreated()` method of the. `Command`. Caching is implemented using a `HashMap` that maps `DisplayPane` enumerations to the corresponding `UiPart` representing the respective pane.
- An illustration of how the information flows for the `editbio` command is shown as follows:



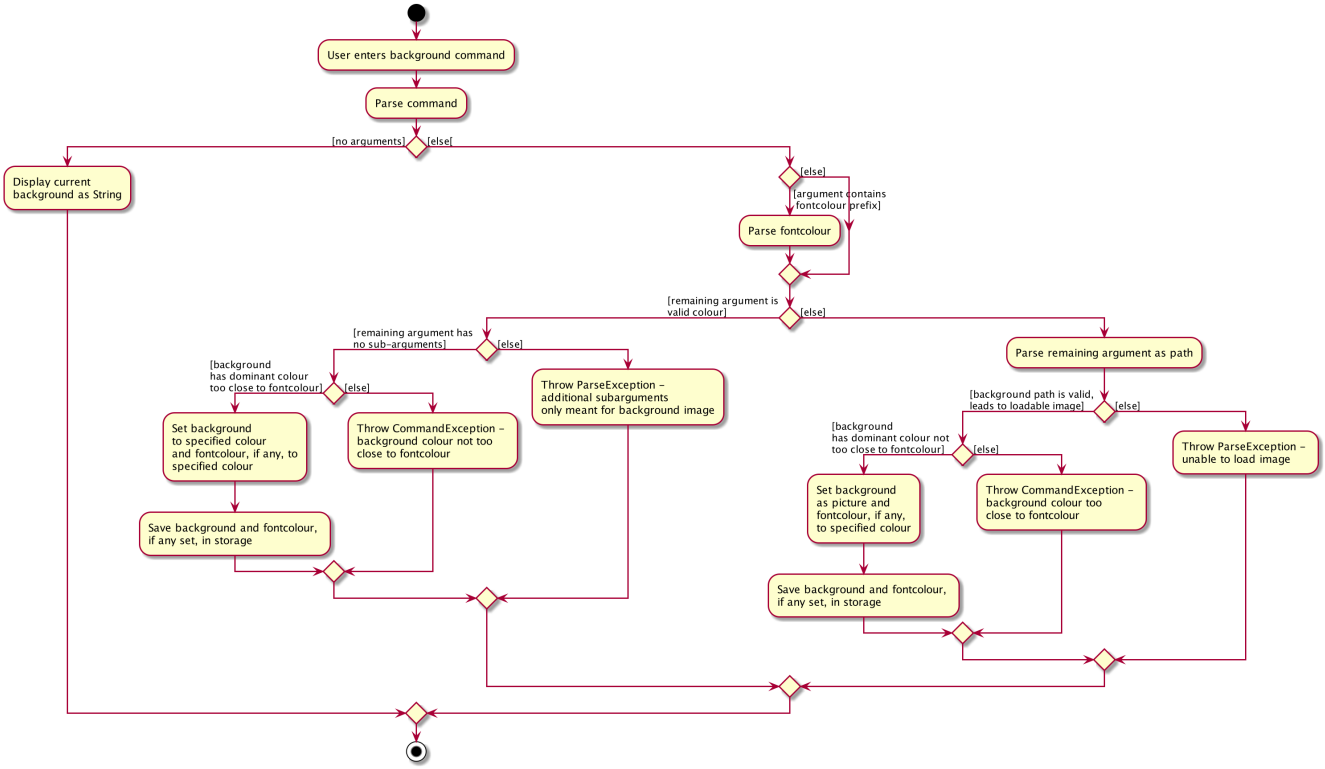
- The rest of the biography commands follow a similar logic, with key differences in the parser and command steps as described above. Validation within parsers are done via the `ParserUtil` class.

Aesthetics

The aesthetics aspects of the application help to support the feature of personalised user experience and are implemented using the command words `fontcolour` and `bg` respectively.

- Possible valid usages are as follows:
  - Scenario 1: User keys in `fontcolour`
  - Scenario 2: User keys in `fontcolour white`
  - Scenario 3: User keys in `fontcolour #FFFF00`
  - Scenario 4: User keys in `bg`
  - Scenario 5: User keys in `bg #000000`
  - Scenario 6: User keys in `bg blue`
  - Scenario 7: User keys in `bg /Users/John/displayPicture.jpg s/cover`
  - Scenario 8: User keys in `bg r/no-repeat`
- As mentioned above, `Colour` and `Background` are independent classes, and `Colour` makes use of enumerations of colour names and hexadecimal colour codes to determine validity of the colours.
- Upon receival of the command `fontcolour`, if `fontcolour` has no arguments (checked by `FontColourParser`), a new `FontColourCommand` with no arguments is returned, and upon execution return a `CommandResult` that shows the existing `fontcolour` used via access of `ModelManager` (logic is similar to the ones for biography)
  - Otherwise if arguments are received, validity of the arguments is checked against, and if the colour is a valid `Colour`, it is set in `ModelManager` and saved to Storage. `FontColourCommand` overrides the `getDisplayPane()` to return the `DisplayPane.COLOUR` enumeration. i.e. the `MainDisplayPane` is unchanged in `Ui`, and only font colours change.
- `Background` on the other hand, checks for additional possible arguments. First of all, as observed in Scenarios 6 and 7, an argument could either represent a `Colour` or a path leading to an image to be used to set the background picture (this is similar to the `DisplayPicPath` of `bio` field). Thus, `BackgroundParser` first determines if the argument received is a `Colour`. If so it returns a `BackgroundCommand` storing a `Background` that has a `backgroundColour` attribute. Otherwise, it checks, via `ParserUtil` , whether or not the argument before valid prefixes (preamble) is a valid file path. If so, a `Background` that has a `backgroundPicPath` attribute is used to create the `BackgroundCommand`.
  - Otherwise a `ParseException` is returned. Possible arguments that a `bg` command can have include the size and repeat feature, corresponding to CSS `background` attributes.
  - In current versions of the program, the program allows for fixed constants of this features to be used, that are stored in `BackgroundImageArgs` class and used by the `Background` model for validation.
  - `BackgroundCommand` overrides the `getDisplayPane()` method to return `DisplayPane.BACKGROUND` enumeration. i.e. the `MainDisplayPane` is unchanged in `Ui`, and only the `background` changes.
  - Similar to font colour, the command word on its own simply displays to the user current `background` settings.

◦ An illustration of the logic for handling a `bg` command is shown as follows:



- For both `fontcolour` and `bg` commands, the StyleManager class of `Ui` is used to set the user's intention of `fontcolour` and `background` (if parsing is successful). The way StyleManager sets the `background` is by making a copy of the existing StyleSheet used, modifying the required fields and setting it to the StyleSheets of the scene, internally.
- Perhaps an interesting area of the `Colour` and `Background` commands in more recent updates would include implementation using command composition. The driving factor that fueled this is the need to ensure the `Fontcolour` and `Background` do not have colours that are too similar (or otherwise the text could get difficult or impossible to see). This above-mentioned checking was implemented by summing the square of the differences in red, green and blue channels' values between the `Colour` of the `Fontcolour` and `Background`.
- The `Colour` for a `Background` with an image instead of a solid `Colour` is determined by extracting the `Colour` that appears the most often using the `ImageTester` class.
- An major issue with checking for colour differences would be the situation when the user intends to make changes to a `Fontcolour` that clashes with the `Background` if changed. Take for example a change in `fontcolour` intended to be changed from white to black, with a background that is curently *already* black. The system would not have allowed changes of the text from white to black because of the background's black colour and would have suggested to change the background first. The background is required to be changed to something much lighter so that the background can be set to black. However, if the background cannot be changed to something that is lighter than it's current colour but yet dark enough not to clash with the current background colour, then the user could find it hard to switch to the new colours without going through a series of specific steps that would not cause colour clash.
- Command composition allows the `bg` and `fontcolour` commands to be combined such that the user is able to set both the `background` and `fontcolour` simultaneously, and as such colour comparison is made solely between the new colours entered rather than any of the current

colours.

- `BackgroundParser` parses for `fontcolour/` and its arguments while `FontColourParser` parses for `bg/` and its arguments. Any of these prefixes observed results in the Parser generating a `FontColourCommand` and `BackgroundColourCommand` respectively. `BackgroundParser` then returns a `BackgroundColourCommand` that has a `FontColourCommand` *stored* in it and vice versa. When `LogicManager` executes `BackgroundCommand`, for instance, `BackgroundCommand` executes the `FontColourCommand` stored in it as well. The necessary adjustments are made to model accordingly and the feedback to users from both commands will be returned to the user.
- The idea of a command running another command allows commands such as `bg black fontcolour/red` to be entered by the user. Modified methods in the `ArgumentMultimap` class of the `logic` package also allows the program to ensure that the user does not enter multiple arguments of the same type at once eg. disallowing `bg black fontcolour/red fontcolor/yellow`.

Achievements

- A diabetic user's `Achievements` is supported by the `achvm` command, that displays the list of user's achievements. Similar to how `bio` is implemented, `SugarMummyParser` returns an `AchievementsCommand` that overrides the `getDisplayPane()` method to return `DisplayPane.ACHVM` – such that `Ui` of `Ui` sets the children of the `MainDisplayPane` node to be the `AchievementsPane`. Each `Achievement` is represented using an `ImageView` in JavaFX `TilePane` so that all images are of the same size.
- An `Achievement` is implemented as an abstract class in the `model` package. Each achievement contains attributes that define the `Achievement` such as its `title` and `description` which specifies the requirements needed to attain it. A significant attribute of the `Achievement` class is it's three states - `Achieved`, `Yet to Achieve` and `Previously Achieved`. Another would be the `level` of the achievement (eg. `Bronze`, `Silver`, `Gold` etc.)
- Current `Achievement` objects have `recordType Bmi` and `BloodSugar`, with corresponding interfaces that represent the `Achievement` for its `RecordType`. Specific classes inherit the `Bmi` and `BloodSugar` interfaces while extending the `Achievement` abstract class to specify defining attributes and methods.
- When the program starts, an `AchievementsMap` containing a `Map` of `RecordType` to `List` of all `Achievement` objects that the program has is created in `ModelManager`. All `Achievement` objects are initially all at the state of `Yet to Achieve`.
- The `AchievementStateProcessor` class is then called, which iterates through the list of all `Record` elements stored in `ModelManager` and updates the `State` of each `Achievement` if necessary.
- For each `RecordType` and `Level` of `Achievement`, the `AchievementStateProcessor` class checks whether the records fulfils the requirements for a predefined number of consecutive days. Requirements are in turn determined by the `MAXIMUM` and `MINIMUM` values stored in the interfaces of the `Achievement` class. State changes are made to the `Achievement` class if requirements are fulfilled (eg. if the number of requirements of a `RecordType` for `Gold` are met, then the `Achievement` of `level Gold` and of that particular `RecordType` would have it's state updated to reflect that change. This is accomplished using methods such as the `promote` and `demote` in the `AchievementStateProcessor`).
- In order to determine whether requirements are fulfilled, interaction with not only the `RecordType` is implemented, but also the methods of the `Average` feature (to obtain daily averages

of record types before comparing them).

- A notable aspect of the implementation is the reversal of `level` from high to low level. This is such that if a higher-level `Achievement` has been achieved, lower levels of achievement would also have been attained. In such cases, the program automatically sets lower levels of `Achievement` to be achieved without having to iterate through the rest of the `Record` elements in the `RecordList`.
- Thereafter, for each addition and removal of `Record` elements, the same process described above is used to update the `AchievementsMap`, that maps `RecordType` to an `AchievementsList` of `Achievement` elements with updated `State` attributes.
- When the `achvm` command is received by the program, this `AchievementsMap` is simply retrieved from `ModelManager` to `LogicManager` and the corresponding images representing the `Achievement` objects in the list, with their `State` values, and attributes are presented to the user via the `MainDisplayPane` of the `MainWindow`.
- If the `AchievementsList` happens to be unchanged since the last time the pane is loaded in the same session, the pane is not reload so as to optimise performance of the program and minimise unnecessary access and loading of images.

Motivation

- Motivational aspects of the application are supported using motivational quotes.
- Each motivational quote exists as a `String` in an unmodifiable `List` of the class `MotivationalQuotes`.
- The `List` of quotes (collated from different sources but modified to have the same formats) are initialised to be part of `ModelManager` when the program first starts up.
- Upon initialisation of the program, the `MotivationalQuotesLabel.fxml` file is referenced via its corresponding class.
- Retrieval of the `List` of motivational quotes is done via `LogicManager` which accesses the `List` of motivational quotes in `ModelManager`.
- A quote is randomly selected and then displayed to the user via the program's user interface.

3.7.3. Design Considerations

Number of Users

- It could be argued that multiple user support is not required and thus a `UserList` should not be used to store data. However, the intention is to leave it open to future developers to decide on whether to include multiple user support for the application, as the choice of a fully personalised experience for diabetic patients versus functionality for multiple users (having diabetes and using the same app), as well as the possibilities of such scenarios are debatable. Furthermore, our user stories appear to suggest the desire for a more personalised application.
- In the strict case of single-user support that leaves the app less open to such modification, the alternative would be to simply implement and store the `User` in `ModelManager`, rather than the `UserList`.

Background Sub-Argument Values

- The use of `enum` is a possibility to implement `static final background` sub-argument values (eg, `auto` of attribute `background` size). However considerations that eventually led against this idea included the possibility of values that are not in proper `String` format that may not be able to be directly enumerated (leading to the required use of additional lengthy switch cases). Additionally other `background` fields may be added by future developers and it would be more concise to have them all in a single class rather than as separate enumerations.

Command Classification

- It is possible to separate the commands for `fontcolour` and `background` into different commands (eg. `addfontcolour`, `editfontcolour`, `showfontcolour`, `clrfontcolour`). However, this is likely unnecessary as this will not only require the end user to type more words, but also introduce redundancy (eg. `clrfontcolour` could simply be `fontcolour black` and this is not hard to achieve the same effects as `clrfontcolour`, which also adds a restrictive definition on what the default colour to the user upon clearing settings should be).

Modification of Application Style Dynamically

- An alternative idea to achieving `fontcolour` and `background` throughout the entire app was to visit each `JavaFX` child `Node` recursively and set the colours and backgrounds if the nodes are of specific instances with these attributes (eg. `Label` which has `textfill` attribute). However this idea was quickly aborted as the `TableView` implemented only renders headers after the scene has been set and to include such a case in thet recursive solution adds significant complexity to the program on top of the possibility of severely breaking abstraction.

Restricting User Modification of Motivational Quotes

- The user is specifically designed to have no access in modifying the list as that would not only have taken away the element of surprise but defeat the purpose of motivating the user one step at a time.
- Additionally, no additional commands for switching quotes are implemented as the user may simply restart the application to generate a new `MotivationalQuote` out of the 600+ that are currently available.
- Future developers may decide to add more quotes, or implement the capability for users to add or modify them, but at the moment we believe modification would be unnecessary as user-defined fields may also be achieved via other existing features such as those in the biography. A user may furthermore add to quotes that may turn out to be discouraging without knowing it, or accidentally delete quotes from the list unintentionally, making the user experience of the feature much less deterministic.
- Daily motivational quotes were replaced with motivational quotes that change every time the application is restarted as not only does it increase ease of testability, but also allows the user to encounter something different each time the application is opened. Given the minimal ability intended for the user to modify the quotes, it is perhaps important that a user who may not like what he is seeing on screen, or simply wishes to see something different. does not have to wait till the end of the day in order for a change in quote to be observed.

3.7.4. Achievement Measures and Criteria

- It was difficult to define what a user needs to 'achieve' before he or she gets an achievement.
- The basic idea was to allow for different achievement levels which was eventually implemented. However, marking of the boundaries of when a user attains an `Achievement` was debatable and could still be amongst developers.
- An initial consideration was to award users achievements based on the average of the data in their health records. In other words, take the average of all data within a specific time period and award the achievement if the data within that time period matches the requirement. However a major flaw with this idea was how users would eventually be able to 'cheat' - by minimising the number of days during which records are entered, and only recording data when results are desirable. The other issue was the duration during which the average was determined. Suppose an achievement may be attained by the user upon meeting requirements based on data over a year on *average*. This means that a user could enter a record that meets the requirements in year 1, and then one year later enter another record that meets the requirements. By this definition of achievements, the user could have received the achievement even though the records may not have met requirements for the majority of year (especially for records that were not keyed in).
- Thus, user's achievements were defined by the actual duration during which they met requirements, and furthermore for *consecutive* number of days. i.e. streak
- This ensures that the user is incentivised not only to achieve good records (and in the process improve his or her health), but also acquire a good habit of keying in and storing records.

3.7.5. Future Developments

Saving of user's preferred themes: `[coming in v2.0]`

This feature has not currently been implemented, but could possibly be implemented using the existing `StyleManager` class, which processes users' `background` and `fontColour`. A `List` could be used to save an archive of users' preferred themes during that session. Adding, editing and deletion could be accomplished using `List` methods. A `HashMap` could also be used such that the user can self-define names for each of the themes. A variable would serve as a current pointer to determine the current theme the user is using. A change in theme could be achieved by updating the pointer and / or the `HashMap`, if any is implemented. If the user does not have any themes, then default aesthetics would be loaded, or if there is at least one set of saved settings (as there is in this current version of the application), the users' preferences' in those settings would be loaded. Upon termination of the program, the contents of the `HashMap` could be saved to a `JsonStorage` file.

Displaying of cartoon avatar that represents the user: `[coming in v2.0]`

This feature has yet to be implemented but could possibly be implemented using a class / method that interacts with the user's `RecordList`. A larger BMI of the user could be represented by a figure with a wider profile while a smaller BMI could lead to the avatar being represented otherwise. Users could also have the option to enable and disable this feature. This dynamically changing avatar could be achieved by combining shapes that change according to the values in `ReordList`, or by using an existing library that allows for this.

Follow up on user's goals: `[coming in v2.0]`

This feature has yet to be implemented but could possibly be implemented by first parsing inputs tat the user has entered for the `Goal` fields. If in a format that is recognised, the program would store the recognised parsed `Goal` and corresponding `LocalDate` in an `ArrayList` and `JsonStorage` file. The program would then check the user's progress over time by analysing data in the user's `RecordList`, and provide timely feedback by comparing the current date and date by which to reach the `Goal` targets set. For instance, the program may display a new alert-box like window via the `UI` indicating to user 'good job' for perhaps being 'halfway there' in attaining set goals. This feature may also implement some methods from the `Reminder` feature so the user can choose to automatically be reminded about his/her `Goal` inputs at specific time intervals desired.

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:

- diagnosed with type 2 diabetes
- consults a professional health practitioner
- has a need to manage a significant number of health-related records and tasks
- is diligent in immediately recording events but subsequently forgets events
- wants to gain a deeper understanding of his/her condition
- is struggling with obesity and lack of sleep
- is motivated by challenges
- enjoys a personalised experience
- needs to know his/her effectiveness in managing diabetes at a glance
- prefer desktop apps over other types
- can type fast
- reads and writes competently in English
- prefers typing over mouse input

- is reasonably comfortable using CLI apps

Value proposition: convenient all-in-one app for effectively managing diabetes that is faster than a typical mouse/GUI driven app

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…
* * *	diabetic patient who has different options on medical care	know exactly how much I am spending on medication and consultation	know which hospitals to seek medical care from
* * *	very busy diabetic	use a flexible calendar system that can account for updates	easily make changes to appointments that I have to change often due to other commitments
* * *	diabetic	keep track of my medical expenses	better manage my finance
* * *	person who likes numbers	see summary statistics	better track my progress
* * *	diabetic	get an overview of my dieting/exercising data regularly	save time because I am working 9-5
* * *	forgetful diabetic	be reminded to attend my medical appointments	know how well my existing measures work
* * *	patient who has recently been diagnosed of diabetes	be informed when I eat food with high sugar content	live better and reduce the chances of further health deterioration
* * *	lazy diabetic	have reminders for exercising	force myself to work out.
* * *	busy diabetic	be reminded on when to refill / stock up on insulin	
* * *	diabetic	see graphic data summary	minimise the need to read long paragraphs
* * *	diabetic patient who has just been recently diagnosed	have some motivation and reminders on my diet	reduce my struggles of cutting down on meals or even exercise that is really tough for me

Priority	As a …	I want to …	So that I can…
* * *	diabetic	automatically calculate my daily sugar/carb intake	eliminate the trouble to search for the levels of sugar content in the food I eat everyday.
* * *	diabetic who values my punctuality	adhere to my appointment timings	uphold my principles and take responsibility of my own health by not missing my appointments.
* * *	diabetic	reminded to take my insulin regularly	
* * *	diabetic	be able to track my sugar levels	
* * *	task-oriented diabetic patient	have a goal to work towards or a challenge to work on everyday	have a sense of direction in what I can do to improve my health
* *	caretaker of an elderly patient with diabetes whose family members are busy working	have a reliable app to keep track of all the patients' activities	can answer to the family members who have entrusted unto me this responsibility of care
* *	busy person	be able to easily sort and prioritize my tasks	better manage my time
* *	diabetic patient who is often being referred to new doctors at different specialist clinics every now and then	be able to be able to export all my records and activities at once	rule out the possibility of missing any information during the registration process at a new clinic/ hospital I am visiting
* *	family member of a diabetic	prioritize my tasks	be immediately contactable if my family member has an emergency situation that requires urgent medical attention
* *	diabetic	have a customisable app with avatars and different backgrounds	enjoy a personalised experience
* *	lazy and obese individual	be motivated constantly to exercise	stop procrastinating

Priority	As a ...	I want to ...	So that I can...
* *	forgetful diabetics patient	have a record of my doctors' advice for each medical appointment and prescription directions	better understand the steps that I can take to improve my condition until the next consultation
* *	achievement-oriented diabetic	view the achievements and progress I have made on food intake	remain motivated to keep my streak on good habits going
* *	paranoid diabetic who values privacy	secure/encrypt my health data and other private contact details	protect my data
* *	diabetic patient with a family	have a user-friendly app that helps me manage my medical data and appointments on my own	free the burden I have on my family
* *	diabetic patient with a family	have a user-friendly app with natural commands that helps me manage my medical data and appointments on my own	free the burden I have on my family
*	diabetic patient in a community of diabetic patients	have a standardised means of comparing our activities via a social network	learn from my peers, encourage and be encouraged through this difficult journey.
*	careless user	undo my most recent actions	easily make necessary amendments and input the correct commands
*	a diabetic patient who has many medical receipts - and is not very good at mathematics	have a simple calculator that is always easily accessible	instantly calculate all my medical costs when needed
*	an obese working adult at high risk of diabetes	start monitoring my diet	minimise my risk of having diabetes

Priority	As a …	I want to …	So that I can…
*	medical consultant	export my patient's health data	save my time

Appendix C: Use Cases

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

C.1. Use case: Add blood sugar record

MSS

1. User requests to add a blood sugar record
2. System adds the blood sugar record

   Use case ends.

Extensions

1a. The record is incomplete or passed invalid arguments.

  1a1. System shows an error message.

  Use case resumes at step 1.

Use case: Schedule a medical appointment

MSS

1. User requests to add a medical appointment
2. System adds the medical appointment
3. System notifies user of upcoming medical appointment beforehand
4. User acknowledges the notification and attends medical appointment on schedule

   Use case ends.

Extensions

1a. The appointment is incomplete or passed invalid arguments.

  1a1. System shows an error message.

  Use case resumes at step 1.

3a. User snoozes the notification.

3a1. System waits for snooze time to elapse.

Use case resumes at step 3.

Use case: Delete blood sugar record

MSS

1. User requests list of blood sugar records
2. System shows a list of blood sugar records
3. User requests to delete a specific blood sugar record in the list
4. System deletes the blood sugar record

Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index is invalid.

3a1. System shows an error message.

Use case resumes at step 2.

Use case: Recommend diabetes-friendly food

MSS

1. User requests for a diabetes-friendly food item
2. System shows a diabetes-friendly food item
3. User likes the recommendation

Use case ends.

Extensions

3a. User dislikes the recommendation.

3a1. User requests for another diabetes-friendly food item

Use case resumes at step 2.

Use case: Update blood sugar record

MSS

1. User requests list of blood sugar records
2. System shows a list of blood sugar records
3. User requests to update a specific blood sugar record in the list
4. System updates the blood sugar record

   Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index is invalid.

3a1. System shows an error message.

Use case resumes at step 2.

3b. The record is incomplete or passed invalid arguments.

3b1. System shows an error message.

Use case resumes at step 2.

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 health-related records and tasks 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. Third-party frameworks/libraries used should be free, open-source, and have permissive license terms, should not require any installation by the user of this software, and approved by teaching team.
5. Should work without requiring an installer.
6. The software should not depend on your own remote server

Appendix E: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

Appendix F: Product Survey

Product Name

Author: …

Pros:

- …
- …

Cons:

- …
- …

Appendix G: Instructions for Manual Testing

Given below are instructions to test the app manually.

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

G.1. Launch and Shutdown

1. Initial launch
   a. Download the jar file and copy into an empty folder
   b. Double-click the jar file
      Expected: Shows the GUI with a set of sample contacts. The window size may not be optimum.
2. Saving window preferences
   a. Resize the window to an optimum size. Move the window to a different location. Close the window.
   b. Re-launch the app by double-clicking the jar file.
      Expected: The most recent window size and location is retained.

{ more test cases … }

G.2. Deleting a Person

1. Deleting a person while all persons are listed
   a. Prerequisites: List all persons using the `list` command. Multiple persons in the list.
   b. Test case: `delete 1`

Expected: First contact is deleted from the list. Details of the deleted contact shown in the status message. Timestamp in the status bar is updated.

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

d. Other incorrect delete commands to try: `delete`, `delete x` (where x is larger than the list size)
{give more}
Expected: Similar to previous.

{ more test cases … }

G.3. Saving Data

1. Dealing with missing/corrupted data files

   a. *{explain how to simulate a missing/corrupted file and the expected behavior}*

{ more test cases … }

G.4. Average Command

a. Prerequisites: There are exactly 7 different days of blood sugar and exactly 7 different days of BMI records.

   i. Test case: `average a/daily rt/bloodsugar`
   Expected: Shows a graph with 5 data points. The dates of the 5 data points are the 5 most recent blood sugar records.

   ii. Test case: `average a/daily rt/bmi n/10`
   Expected: Since there are only 7 BMI records, the graph will only have 7 data points instead of 10.

   iii. Test case: `average a/yearly rt/bmi n/3`
   Expected: This is an unsupported average type. An error message is displayed saying
   `Please enter correct input for a/AVERAGE_TYPE!`
   `AVERAGE_TYPE is "daily", "weekly" or "monthly".`

   iv. Test case: `average a/weekly`
   Expected: Missing compulsory field rt/RECORD_TYPE. An error message is shown:
   `Oops! The command you've entered appears to be in an invalid format.`
   `average: Shows daily/weekly/monthly average of different record types in a line graph.`
   `Format: average a/AVERAGE_TYPE rt/RECORD_TYPE [n/COUNT]`
   `Example: average a/daily rt/bloodsugar n/5`

b. Prerequisites: There are exactly 3 distinct weeks of blood sugar records and no BMI records.

   i. Test case: `average a/weekly rt/bloodsugar`
   Expected: Since there are only 3 blood sugar records, the graph will only have 3 data points with dates of the 3 most recent blood sugar records in terms of week. There is not enough records to show 5 data points.

   ii. Test case: `average a/weekly rt/bmi`

Expected: Since there are no bmi records, an error message is displayed saying
`Sorry! You do not have any BMI record.`

c. Prerequisites: There are at least 12 distinct months of BMI records and no blood sugar records.

   i. Test case: `average a/monthly rt/bmi n/9`
   Expected: Shows a graph with 9 data points and these points are the average BMI values of the 9 most recent month.

   ii. Test case: `average a/monthly rt/expenses n/3`
   Expected: This is an unsupported record type. Following error message will be shown:
   ```
   Please enter correct input for rt/RECORD_TYPE!
   RECORD_TYPE is "BLOODSUGAR" or "BMI"
   ```

   iii. Test case: `average a/monthly rt/bmi n/13`
   Expected: COUNT field is out of the range 1 and 12 inclusive. Following error message will be shown:
   ```
   Please enter correct input for n/COUNT!
   COUNT takes integer value between 1 and 12 inclusive.
   ```

   iv. Test case: `average a/monthly rt/bmi n/five`
   Expected: COUNT field only takes integer value. Following error message will be shown:
   ```
   Please enter correct input for n/COUNT!
   COUNT takes integer value between 1 and 12 inclusive.
   ```

G.5. Achievement Commands

i. Test case: `achvm asdf`
Expected: A error message is shown to the user indicating that the command cannot have any arguments.

   a. Prerequisites: There are at least 3 days worth of bloodsugar records with a minimum of the past three days having consistent daily averages of 4.0 to 7.8 mmol/L of bloodsugar level.

ii. Test case: `achvm`+ Expected: Bronze level achievement for BloodSugar is shown to be `ACHIEVED` in the achievements pane. Coloured image representing achievement is shown.

iii. Test case: `achVm`+ Expected: Bronze level achievement for BloodSugar is shown to be `ACHIEVED` in the achievements pane. The `achvm` command is not case-sensitive.

   a. Prerequisites: There are at least 2 days worth of bloodsugar records with a minimum of the past two days having consistent daily averages of 4.0 to 7.8 mmol/L of bloodsugar level.

iv. Test case: `add rt/BLOODSUGAR dt/2019-11-06 12:12 con/4.5`+ Expected: An achievement message is appended to the message showing successful addition of records in the feedback display pane, indicated the attainment of (an) achievement(s). Bronze level achievement for BloodSugar is shown to be `ACHIEVED` in the achievements pane when `achvm` is entered.

   a. Prerequisites: There are EXACTLY 3 days of bloodsugar records (one record per day) having consistent daily bloodsugar levels of 4.0 to 7.8 mmol/L.

v. Test case: `delete 3`+ Expected: A nessage is appended to the successful records removal message indicating the loss of (an) achievement(s). Bronze level achievement for BloodSugar is no longer shown to be `ACHIEVED` in the achievements pane when `achvm` is entered. Achievement state resets to `YET TO ACHIEVE` and image representing achievement can is in silhouette form again.

a. Prerequisites: There are at least 3 days worth of bloodsugar records with a minimum of the past three days having consistent daily averages of 4.0 to 7.8 mmol/L of bloodsugar level. The last date of bloodsugar records is on 2019-11-06.

vi. Test case: `add rt/BLOODSUGAR dt/2019-11-07 12:12 con/4.5`+ Expected: Bronze level achievement for BloodSugar continues to be shown to be `ACHIEVED` in the achievements pane if `achvm` is entered.

a. Prerequisites: There are at least 3 days worth of bloodsugar records with a minimum of the past three days having consistent daily averages of 4.0 to 7.8 mmol/L of bloodsugar level. The last date of bloodsugar records is on 2019-11-06.

vii. Test case: `add rt/BLOODSUGAR dt/2019-11-08 12:12 con/4.5`+ Expected: Bronze level achievement for BloodSugar continues to be shown to be `PREVIOUSLY ACHIEVED` in the achievements pane if `achvm` is entered. Image representing achievement is gray-scaled and streak count resets to zero.

a. Prerequisites: There are at least 3 days worth of bloodsugar records with a minimum of the past three days having consistent daily averages of 4.0 to 7.8 mmol/L of bloodsugar level. The last date of bloodsugar records is on 2019-11-06.

viii. Test case: `add rt/BLOODSUGAR dt/2019-11-07 12:12 con/7.9`+ Expected: Bronze level achievement for BloodSugar continues to be shown to be `PREVIOUSLY ACHIEVED` in the achievements pane if `achvm` is entered. Image representing achievement is gray-scaled and streak count resets to zero.

G.6. Biography Commands

a. Prerequisites: NIL

i. Test case: `bio`+ Expected: Existing biography pane with profile picture, fields and data. If no biography has been set, an empty biography containing a default profile picture will be shown. Fields showing background, background size/ repeat and font colour should not be affected whether or not there is a biography. If a field has no item, it should be an empty `String`.

ii. Test case: `clrbio asdf`
Expected: A error message is shown to the user indicating that the command cannot have any arguments.

b. Prerequisites: There is no existing biography.

i. Test case: `addbio n/test minimal p/91234567 e/81234567 m/test medical condition`
Expected: A biography with updated fields name, phone, emergency contacts and medical condition is shown in the biography display pane. All other fields will remain blank. A message indicating success is displayed in the feedback display pane along with fields added.

ii. Test case: `addbio desc/hello world n/testName nric/testNric gender/testGender dob/1920-10-08 p/12343567 p/91234567 e/81234567 m/test medical condition a/example address 123 goal/testGoal o/testOtherInfo`
Expected: A biography with entered fields is shown in the biography display pane. For listable fields (i.e. of prefix p/ e/ m/ g/), if more than one field is entered, the items will be presented in a numbered list in it's cell of the biography table. A message indicating success is displayed in the feedback display pane along with fields added.

iii. Test case: `addbio n/firstTestName n/secondTestName p/91234567 e/81234567 m/test medical condition ` Expected: An error message is displayed showing there cannot be more than one prefix for n/.

iv. Test case: `addbio n/firstTestName n/secondTestName gender/Male gender/Female p/91234567 e/81234567 m/test medical condition`
Expected: An error message is displayed showing there cannot be more than one prefix for n/ and gender/ (displayed as the default `String` representation of a list to the user).

v. Test case: `addbio n/name1 p/91234567 e/81234567 m/test medical condition`
Expected: An error message is displayed showing names can only contain alphabets and spaces, and should not be blank.

vi. Test case: `addbio n/test minimal nric/@2 p/91234567 e/81234567 m/test medical condition`
Expected: An error message is displayed showing NRICs can only contain alphanumeric characters and spaces, and should not be blank.

vii. Test case: `addbio n/test minimal p/91234567hi e/81234567 m/test medical condition`
Expected: An error message is displayed showing that phone numbers should only contain numbers, and should be at least 3 digits long.

viii. Test case: `addbio n/test minimal p/91234567 e/81 m/test medical condition`
Expected: An error message is displayed showing that phone numbers should only contain numbers, and should be at least 3 digits long.

ix. Test case: `addbio n/test minimal p/91234567 e/12345 m/test medical condition m/test medical condition`
Expected: An error message is displayed showing that there are duplicate medical conditions found.

x. Test case: `addbio n/ test minimal p/ 91234567 e/12345 m/test medical condition`
Expected: Biography is added successfully with a message displayed to the user on fields added. Spaces in between fields do not affect parsing and spaces before arguments are automatically removed. Biography display pane is shown.

xi. Test case: `editbio n/test minimal`
Expected: An error message is displayed to the user indicating that a bio does not exist and to suggest creating a new biography.

xii. Test case: `aDdBio n/test minimal p/91234567 e/81234567 m/test medical condition`
Expected: A biography with updated fields name, phone, emergency contacts and medical condition is shown in the biography display pane. All other fields will remain blank. A message indicating success is displayed in the feedback display pane along with fields added. Capital letters in the command word do not affect the use of the program.

xiii. Test case: `addbio N/test minimal p/91234567 e/81234567 m/test medical condition`
Expected: An error message is displayed to the user as upper case fields are not recognised.

xiv. Test case: `addbio n/test minimal p/91234567 e/81234567 m/test medical condition GENDER/male`
Expected: Biography is added successfully but `GENDER/male` will be appended rather than added as a field.

xv. Test case: `addbio m/test medical condition p/91234567 e/81234567 n/test minimal`
Expected: Biography is added successfully with similar results as described above (for

successful addition). Order of fields do not matter so long as command word is in front.

xvi. Test case: `addbio m/test Medical Conditionp/91234567 e/81234567 n/test minimal`
Expected: An error message is shown to the user as fields must be separated by a space and in this case, the field for contact number cannot be found.

xvii. Test case: `addbio m/test Medical Conditionp/91234567 p/123 e/81234567 n/test minimal`
Expected: Biography is added successfully but `p/91234567` is appended to `test medical condition` as fields need to be separated by a space.

xviii. Test case: `addbio m/test Medical Condition p/91234567 123 e/81234567 n/test minimal`
Expected: An error message is shown to the user as phone numbers cannot contain a space.

xix. Test case: `clrbio`
Expected: A error message is shown to the user that the biography is already empty and there is no biography to be cleared.

c. Prerequisites: There is an existing biography.

i. Test case: `addbio n/test Minimal p/91234567 e/81234567 m/test Medical Condition`
Expected: An error message is displayed to the user indicating that a bio already exists and suggest clearing, editing or viewing the biography.

ii. Test case: `editbio n/Alan Wong`
Expected: Name is sucessfully changed to Alan Wong in biography. Feedback displays the successful change and modified fields and biography display pane is shown. (if not already on the biography display pane)

iii. Test case: `editBio n/Alan Wong`
Expected: Name is sucessfully changed to Alan Wong in biography. Feedback displays the successful change and modified fields and biography display pane is shown. (if not already on the biography display pane) Capital letters in the command do not affect parsing.

iv. Test case: `editBio N/Alan Wong`
Expected: An error is shown to the user as `N/` is not recognised. Field prefixes are case-sensitive.

v. Test case: `editbio n/Alan Wong p/12345678 p/234567`
Expected: Fields are edited successfully. Feedback displays the successful change and modified fields. Previous list of phone numbers will be replaced by `12345678` and `234567`.

vi. Sub-prerequisite: `Alan Wong` is already the name in the biography and phone number is `12345678`
Test case: `editbio n/Alan Wong p/12345678`
Expected: A message is indicated to the user indicating there is nothing to edit.

vii. Sub-prerequisite: `Alan Wong` is already the name in the biography and phone number is NOT `12345678`
Test case: `editbio n/Alan Wong p/12345678`
Expected: Phone number is successfully replaced with `12345678` but modified fields in the feedback display will show only the change in name.

viii. Sub-prerequisite: There contains two or more emergency contact numbers.
Test case: `editbio e/1/12345 e/2/23456`
Expected: First and second existing emergency contact numbers in the list of emergency contact numbers will be replaced by the ones specified at index 1 and 2 respectively. Note

that this should also similarly work for other listable fields such as Medical Conditions and Goals)

ix. Sub-prerequisite: There does NOT contain two or more emergency contact numbers.
Test case: `editbio e/1/12345 e/2/23456`
Expected: An error message is shown to the user that index is out of bounds.

x. Test case: `editbio e/1/12345 e/23456`
Expected: An error message is displayed to the user indicating that there is inconsistent indexing.

xi. Sub-prerequisite: There contains two or more emergency contact numbers and two or more goals.
Test case: `editbio e/1/12345 e/2/3456 goal/first goal goal/second goal`
Expected: Biography is edited successfully, with edited fields displayed in feedback display pane. Where there is more than one item edited for a field, they are displayed in the `String` representation of a list. Inconsistent indexing applies only if it is within a type of field (eg. emergency contacts in previous test case).

xii. Test case: `editbio n/Alan n/Amy`
Expected: An error message is shown to the user that there can only be one prefix for `n/` (since `Name` is not a `ListableField`)

xiii. Test case: `editbio e/1/12345 e/-2/23456`
Expected: An error message is shown to the user that index is invalid (since index cannot be negative).

xiv. Test case: `editbio e/1/12345 e/hello/23456`
Expected: An error message is shown to the user that index is invalid (since index cannot be a string).

xv. Test case: `editbio n/1/Amy`
Expected: An error message is shown to the user that names can only contain alphabets and spaces and cannot be blank since this format for editing is not recognised for fields that do not inherit `ListableField`.

xvi. Test case: `editbio o/1/Amy`
Expected: Biography is edited successfully but `1/Amy` is treated as a `String` since this format for editing is not recognised for fields that do not inherit `ListableField`.

xvii. Test case: `clrbio`
Expected: A message indicates that the biography is successfully cleared and the user is shown the biography page with a default profile picture. All fields in the biography table should be blank except for the ones showing aesthetics (i.e. `Background`, `Background Size`, `Background Repeat`, `Font Colour`)

xviii. Test action: Restart the application and enter `bio`. Expected: Last set biography is loaded upon start up and displayed.

G.7. Aesthetics Commands

a. Prerequisites: Current font colour is NOT yellow and background colour (or dominant colour of background image) is dark (eg. not white)

i. Test case: `fontcolour yellow`
Expected: Font colour is successfully changed to yellow. Colour changes instantaneously and applies to entire app. User is shown feedback that colour is changed from previous colour to "yellow" but the display pane that the user is on should not change. If the user is viewing the biography pane, the `Font Colour` field changes instantaneously.

ii. Test case: `fontcolour #FFFF00`
Expected: Font colour is successfully changed to yellow as described above. User is shown feedback that colour is changed from previous colour to "yellow" (as the colour is automatically converted)

iii. Test case: `fontcolOUr yeLlow`
Expected: Font colour is successfully changed as described above as both commands and colours are not case sensitive. User feedback should indicate that colour is changed to "yellow". (always displayed in lower case)

iv. Test case: `fontcolOUr #FfFF00`
Expected: Font colour is successfully changed as described above as both commands and colours are not case sensitive. Furthermore, there is automatic conversion of colour. User feedback should indicate that colour is changed to "yellow".

v. Test case: `fontcolor yellow`
Expected: Font colour is successfully changed as described above as the American spelling of "color" is also recognised.

vi. Test action: Restart the application Expected: Last set font colour is loaded upon start up.

b. Prerequisites: Current background colour (or dominant colour of background image) is NOT yellow and font colour is dark (eg. not white)

i. Test case: `bg yellow`
Expected: Background colour is successfully changed to yellow. Colour changes instantaneously and applies to entire app. User is shown feedback that colour is changed from previous colour to "yellow" but the display pane that the user is on should not change. If the user is viewing the biography pane, the `Background` field changes instantaneously.

ii. Test case: `bg #FFFF00`
Expected: Background colour is successfully changed to yellow as described above. User is shown feedback that colour is changed from previous colour to "yellow" (as the colour is automatically converted)

iii. Test case: `bG yeLlow`
Expected: Background colour is successfully changed as described above as both commands and colours are not case sensitive. User feedback should indicate that colour is changed to "yellow". (always displayed in lower case)

iv. Test case: `Bg #FfFF00`
Expected: Background colour is successfully changed as described above as both commands and colours are not case sensitive. Furthermore, there is automatic conversion of colour. User feedback should indicate that colour is changed to "yellow".

c. Prerequisites: Current font colour is NOT yellow and background colour (or dominant colour of background image) is close to yellow (eg. white)

i. Test case: `fontcolour yellow`

Expected: Colour is not set and an error message is shown to the user indicating font colour is too close to background's dominant colour. Feedback suggests for user to either change the background colour/ image first or simultaneously change both font colour and background together.

d. Prerequisites: Current background colour (or dominant colour of background image) is NOT yellow and font colour is close to yellow (eg. white)

   i. Test case: `fontcolour yellow`
   Expected: Colour is not set and an error message is shown to the user indicating background colour (or dominant colour of background image) is too close to font colour. Feedback suggests for user to either change the font colour first or simultaneously change both background and font colour together.

e. Prerequisites: Current font colour is NOT #FF2020 and background colour (or dominant colour of background image) is NOT close to #FF2020 (eg. red)

   i. Test case: `fontcolour FF2020`
   `Expected: Font colour is successfully changed to yellow. Colour changes instantaneously. User is shown feedback that colour is changed from previous colour to "#FF2020"` but the display pane that the user is on should not change. If the user is viewing the biography pane, the `Font Colour` field changes instantaneously. Feedback indicates colour as #FF2020 as there is no CSS colour name assigned for this colour.

   ii. Test case: `fontcolOUr #Ff2020`
   Expected: Font colour is successfully changed as described above as both commands and colours are not case sensitive. User feedback should indicate that colour is changed to "#FF2020". (always displayed in upper case)

f. Prerequisites: Background colour (or dominant colour of background image) is NOT #FF2020 and font colour is NOT close to #FF2020 (eg. red)

   i. Test case: `bg #FF2020`
   Expected: Background is successfully changed to #FF2020. Colour changes instantaneously. User is shown feedback that colour is changed from previous colour to "#FF2020" but the display pane that the user is on should not change. If the user is viewing the biography pane, the `Font Colour` field changes instantaneously. Feedback indicates colour as #FF2020 as there is no CSS colour name assigned for this colour.

   ii. Test case: `bg #Ff2020`
   Expected: Background colour is successfully changed as described above as both commands and colours are not case sensitive. User feedback should indicate that colour is changed to "#FF2020". (always displayed in upper case)

g. Prerequisites: Current font colour is yellow

   i. Test case: `fontcolour yellow`
   Expected: An error message is shown to the user indicating that the font colour is already the same as what was requested and thus there is nothing to be changed.

   ii. Test case: `fontcolour `#FFFF00`
   Expected: An error message is shown to the user indicating that the font colour is already the same as what was requested and thus there is nothing to be changed.

h. Prerequisites: Current background colour is yellow

i. Test case: `bg yellow`
   Expected: An error message is shown to the user indicating that the background is already the same as what was requested and thus there is nothing to be changed.

ii. Test case: `bg `#FFFF00`
   Expected: An error message is shown to the user indicating that the background is already the same as what was requested and thus there is nothing to be changed.

i. Prerequisites: Current font colour is NOT yellow (background can be any colour but different from what it was previously)

   i. Test case: `fontcolour yellow bg/black`
      Expected: Font colour is successfully changed to yellow as described above AND background is changed to black. Feedback message indicates both changes.

j. Prerequisites: Current background colour is NOT yellow (font colour can be any colour but different from what it was previously)

   i. Test case: `bg yellow fontcolour/black`
      Expected: Background colour is successfully changed to yellow as described above AND font colour is changed to black. Feedback message indicates both changes.

k. Prerequisites: Current font colour is yellow (background can be any colour but different from what it was previously)

   i. Test case: `fontcolour yellow bg/black`
      Expected: Font Colour is changed to black. Feedback message indicates that there is nothing to change for background and indicates the change in font colour.

l. Prerequisites: Current background colour is yellow (font colour can be any colour but different from what it was previously)

   i. Test case: `bg yellow fontcolour/black`
      Expected: Background is changed to black. Feedback message indicates that there is nothing to change for fontcolour and indicates the change in background colour.

m. Prerequisites: Current font colour is NOT yellow and background colour is black.

   i. Test case: `fontcolour yellow bg/black`
      Expected: Font colour is changed to yellow. Feedback message indicates change in font colour and that there is nothing to change for background colour.

n. Prerequisites: Current background colour is NOT yellow and font colour is black

   i. Test case: `bg yellow fontcolour/black`
      Expected: Background colour is changed to yellow. Feedback message indicates change in background colour and that there is nothing to change for font colour.

o. Prerequisites: Current font colour is yellow and background colour is black.

   i. Test case: `fontcolour yellow bg/black
      Expected: Feedback message indicates that there is nothing to change.

p. Prerequisites: Current background colour is yellow and font colour is black.

   i. Test case: `bg yellow fontcolour/black
      Expected: Feedback message indicates that there is nothing to change.

G.8. Motivational Quotes Feature

a. Prerequisites: NIL

    i. Test action: Restart the application
    Expected: A new motivation quote is selected at random and shown in the pane showing motivational quotes at the bottom of the window.

Appendix H: References

H.1. Code References

- http://tutorials.jenkov.com/javafx/tableview.html
- https://stackoverflow.com/questions/11184117/transparent-css-background-color
- https://stackoverflow.com/questions/12933918/tableview-has-more-columns-than-specified
- https://stackoverflow.com/questions/37027298/set-constrained-resize-policy-for-columns-without-first-for-number-of-row-in-t
- https://stackoverflow.com/questions/43776047/javafx-tablecolumn-setpreferredwidth-on-a-resizable-column
- https://stackoverflow.com/questions/49882605/javafx-italic-font-w-css
- https://stackoverflow.com/questions/22952531/scrollpanes-in-javafx-8-always-have-gray-background
- https://amyfowlersblog.wordpress.com/2010/05/26/javafx-1-3-growing-shrinking-and-filling/
- https://stackoverflow.com/questions/22202782/how-to-prevent-tableview-from-doing-tablecolumn-re-order-in-javafx-8
- https://stackoverflow.com/questions/14116792/how-to-disable-the-reordering-of-table-columns-in-tableview
- https://medium.com/@keeptoo/adding-data-to-javafx-tableview-stepwise-df582acbae4f
- https://self-learning-java-tutorial.blogspot.com/2018/06/javafx-tableview-adding-new-rows-to.html
- https://stackoverflow.com/questions/39366828/add-a-simple-row-to-javafx-tableview
- https://docs.oracle.com/javafx/2/layout/builtin_layouts.htm
- https://stackoverflow.com/questions/3342298/what-is-the-pattern-for-empty-string
- https://www.geeksforgeeks.org/supplier-interface-in-java-with-examples/
- https://howtodoinjava.com/regex/java-regex-date-format-validation/
- https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html
- https://stackoverflow.com/questions/4343202/difference-between-super-t-and-extends-t-in-java
- https://stackoverflow.com/questions/29004893/transparent-node-background
- https://stackoverflow.com/questions/9851200/setting-background-image-by-javafx-code-not-css

- https://jutge.org/doc/java/docs/api/javafx/scene/doc-files/cssref.html
- https://www.w3.org/TR/css-backgrounds-3/#the-background-repeat
- https://stackoverflow.com/questions/6998551/setting-font-color-of-javafx-tableview-cells
- https://stackoverflow.com/questions/228477/how-do-i-programmatically-determine-operating-system-in-java
- https://htmlcolorcodes.com/color-names/
- https://stackoverflow.com/questions/1636350/how-to-identify-a-given-string-is-hex-color-format
- https://stackoverflow.com/questions/4871051/getting-the-current-working-directory-in-java
- https://stackoverflow.com/questions/7830951/how-can-i-load-computer-directory-images-in-javafx
- https://stackoverflow.com/questions/48814467/how-do-i-bind-the-tablecell-style-classes-to-the-tablerow-style-classes-javafx
- https://www.inf.unibz.it/~calvanese/teaching/06-07-ip/lecture-notes/uni09/node12.html
- https://stackoverflow.com/questions/924394/how-to-get-the-filename-without-the-extension-in-java
- https://stackoverflow.com/questions/32639882/conditionally-color-background-javafx-linechart
- http://java-buddy.blogspot.com/2012/05/create-borderpane-using-fxml.html
- https://stackoverflow.com/questions/19512850/java-putting-hashmap-into-treemap
- https://stackoverflow.com/questions/46170807/gridpane-change-grid-line-color
- https://stackoverflow.com/questions/25168445/how-to-determine-if-color-is-close-to-other-color
- https://stackoverflow.com/questions/4129666/how-to-convert-hex-to-rgb-using-java
- https://stackoverflow.com/questions/3607858/convert-a-rgb-color-value-to-a-hexadecimal-string
- https://stackoverflow.com/questions/10530426/how-can-i-find-dominant-color-of-an-image

H.2. Reference on Healthy Blood Sugar Level Ranges for Diabetic Patients

https://www.diabetes.co.uk/diabetes_care/blood-sugar-level-ranges.html

H.3. Sources of Motivational Quotes

- http://www.wiseoldsayings.com/healthy-eating-quotes/
- https://www.centralofsuccess.com/diabetes-slogans-quotes-funny-inspiring/
- https://shortstatusquotes.com/inspirational-diabetes-status-quotes/
- https://inspiringtips.com/healthy-diet-inspirational-quotes-weight-loss/
- https://www.lifefitness.com.au/20-fitness-motivation-quotes/
- https://www.inc.com/jayson-demers/51-quotes-to-inspire-success-in-your-life-and-business.html

- https://www.stylecraze.com/articles/awesome-motivational-quotes-on-weight-loss/#gref
- https://themighty.com/2016/12/chronic-illness-uplifting-quotes/
- http://www.caringvoice.org/15-encouraging-quotes-chronic-illness-journey/
- https://everydaypower.com/chronic-illness-quotes/
- https://www.mindovermenieres.com/quotes-inspire-chronic-illness/
- https://wisdomquotes.com/health-quotes/

H.4. Credits for Images

- Default User Profile Display Picture: Icon made by Smashicons (https://www.flaticon.com/authors/smashicons) from https://www.flaticon.com/free-icon/user_149068
- Doge Profile Picture in User Guide Screenshot: https://pdxmonthly.com/articles/2015/1/5/who-let-the-doge-out-january-2015
- Mountains Background Image in User Guide Screenshot: https://www.pexels.com/photo/landscape-photography-of-mountains-covered-in-snow-691668/
- Space Background Image in User Guide Screenshot: Andy Holmes via https://unsplash.com/photos/LUpDjlJv4_c (with minor edits)