Stephen Chua - Project Portfolio

PROJECT: SugarMummy

Overview

SugarMummy is a desktop application used to manage a type-II diabetic lifestyle. The user interacts with it using a CLI, and it has a GUI created with JavaFx. It is written in Java and has about 30 kLOC.

Summary of contributions

- **Major enhancement**: added **the ability for the application to handle multiple record types from a small set of commands.**
    - What it does: Allows the user to leverage existing add, list and delete commands for all record types instead of specifying a new set of commands for each record type.
    - Justification: This features improves the product significantly because a user would want to keep track of multiple record types (blood sugar, BMI and more!) while reducing the cognitive load of remembering different commands.
    - Highlights: This enhancement benefits commands to be added in the future since new record types can be easily incorporated into existing commands. Polymorphism was employed to make it easier for the application to adapt to new record types.
- **Minor enhancement**: added the ability to naturally display multiple record types with differing fields without unnecessary whitespace or empty fields.
- **Code contributed**: [View RepoSense]
- **Other contributions**:
    - Documentation:
        - Added Use Cases, Product Scoping and Non-Functional Requirements in the User Guide: #23
        - Detailed Data Logging feature in the User Guide: #127, #128

Contributions to the User Guide

Given below are sections I contributed to the User Guide. They showcase my ability to write documentation targeting end-users.

DATA LOG

Add a data entry to the diabetes tracker app: `add`

Format: `add rt/RECORD_TYPE dt/DATETIME` followed by parameters that the RECORD_TYPE accepts.

There must not be any duplicate parameters. Other than "add", ordering of fields is flexible.

rt/ value is case sensitive, RECORD_TYPE must be in capital letters. dt/ value must not be in the future.

BMI must receive exactly 1 valid parameter each for height and weight: `h/VALUE w/VALUE`

Height input (in meters) will be rounded to 2 decimal places. This rounded value must be less than 3 to be recorded successfully.

Weight input (in kilograms) will be rounded to 2 decimal places. This rounded value must be less than 500 to be recorded successfully.

BLOODSUGAR must receive exactly 1 valid parameter for concentration: `con/VALUE` Concentration (mmol/L) will be rounded to 2 decimal places. This rounded value must be less than 33 to be recorded successfully.

`add` does not allow entries that have the same datetime and RECORD_TYPE.

Show a list of all data entries in the diabetes tracker app: `list`

Format: `list`

This command lists all types of recent data entries.

Deletes a specified data entry from the diabetes tracker app: `delete`

Format: `delete INDEX`

Index must be a positive integer.

The index refers to the index number shown in the displayed entries list

Edits an existing data entry from the diabetes tracker app: `edit [coming in v2.0]`

Format: `edit INDEX`

Allows users to modify existing record fields.

Contributions to the Developer Guide

> *Given below are sections I contributed to the Developer Guide. They showcase my ability to write technical documentation and the technical depth of my contributions to the project.*

Data log feature

Implementation

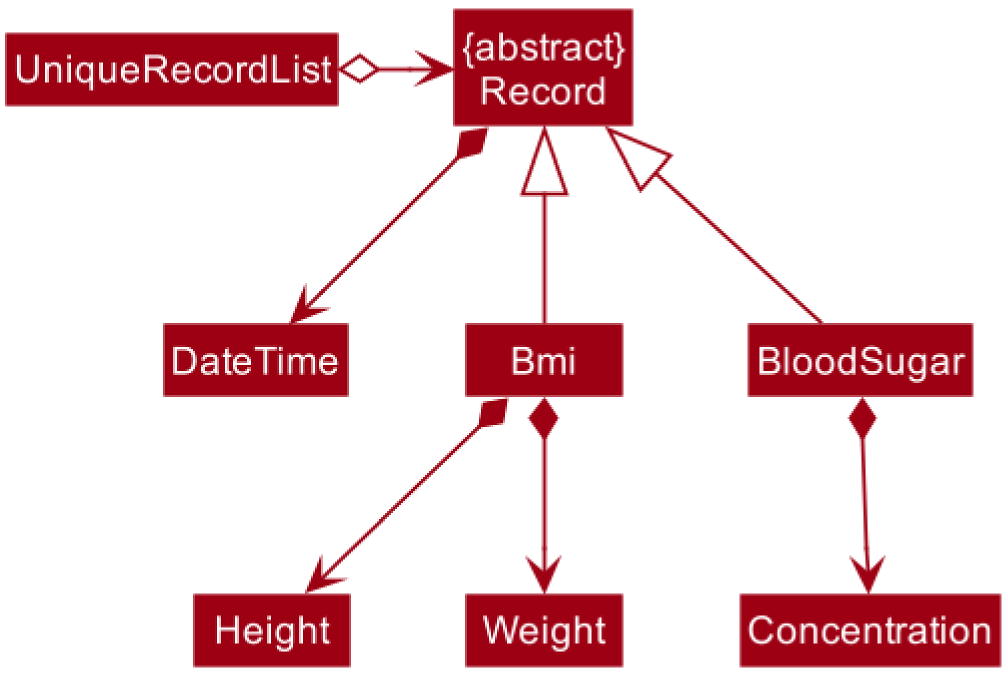
The multi-record data logging mechanism is facilitated by a new Record package containing BloodSugar and Bmi classes that extend an abstract Record class. Add, list and delete command classes and parsers are modified to accommodate multiple record types. Multi-record data is stored internally as a recordList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

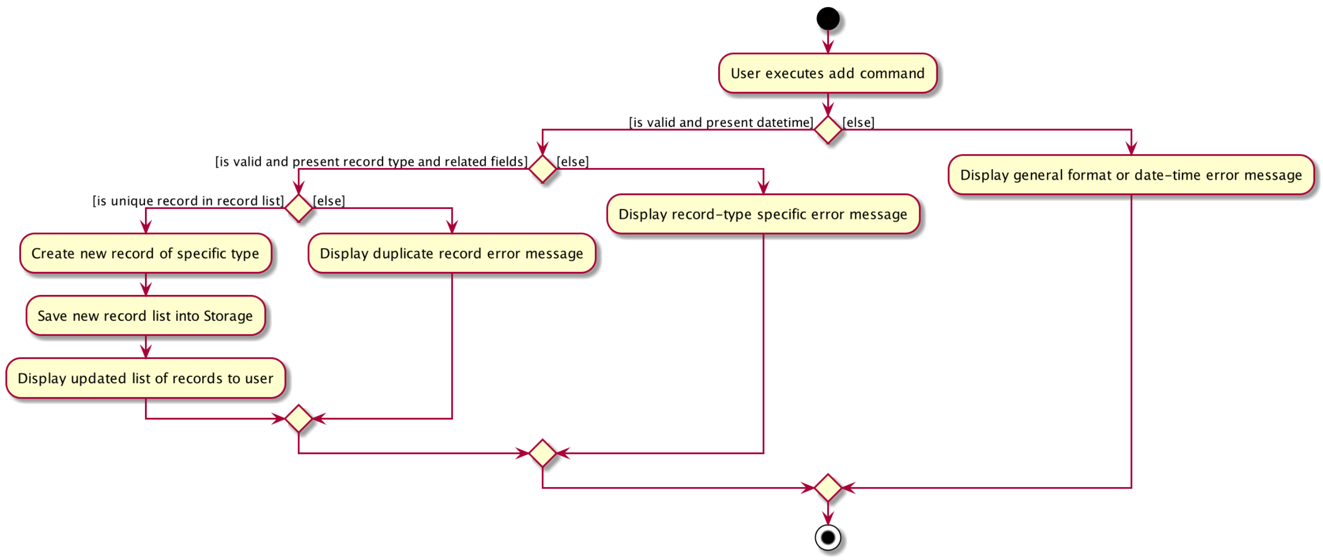


Below is an example usage scenario of how the data log feature behaves at each step.

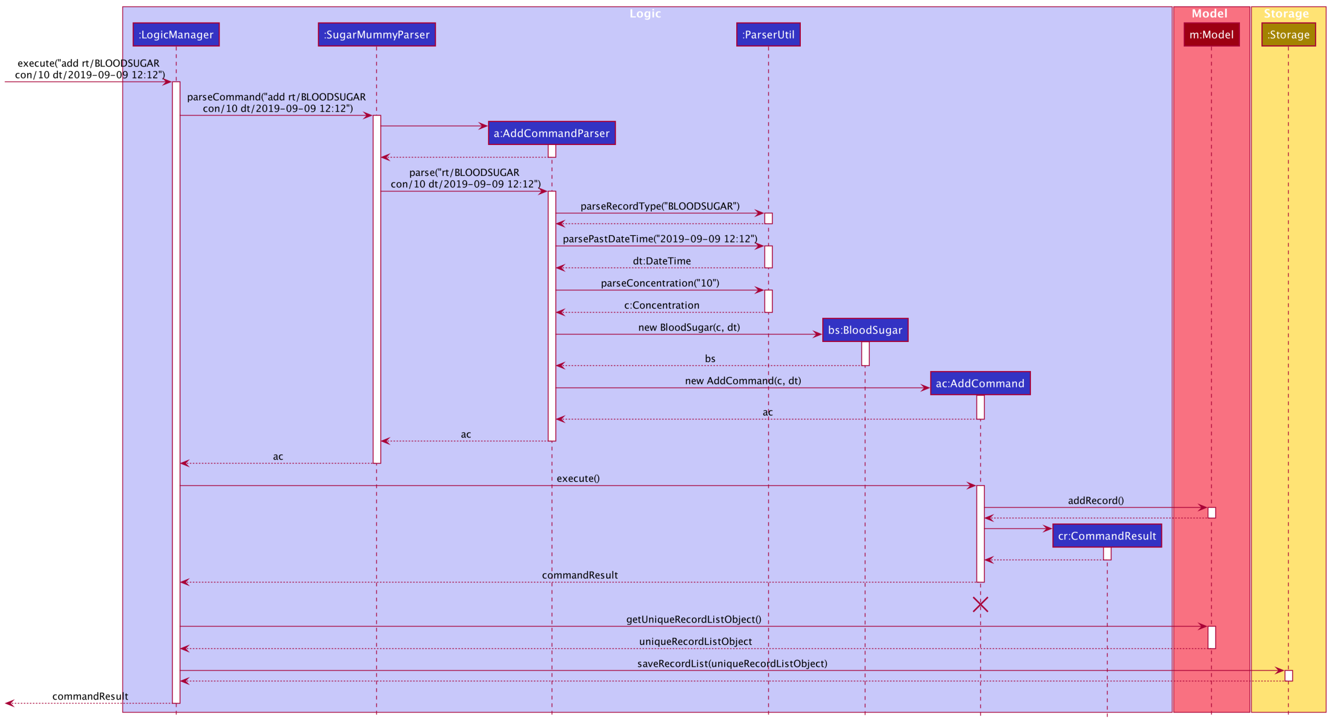
Step 1. The user launches the application for the first time. If `/data/recordList.json` is not found, it will be produced from `SampleRecordDataUtil#getSampleRecords()`. If `/data/recordList.json` is found, the recordList will be loaded from there using `UniqueRecordList#setRecord()` and checked for 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 and uniqueness. 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)


User executes add command
[is valid and present datetime]
[else]
Display general format or date-time error message
[is valid and present record type and related fields]
[else]
Display record-type specific error message
[is unique record in record list]
[else]
Create new record of specific type
Display duplicate record error message
Save new record list into Storage
Display updated list of records to user

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()`.

Design Considerations

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.
- **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.

Aspect: Data Structure for managing multiple record types

- **Alternative 1 (current choice):** Use a single list to store multiple record types.
  - Pros: Easy to understand and 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.

Appendix A: Product Scope

Target user profile:

- is 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
- finds it difficult to get quick and customized suggestions about diets
- 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: Use Cases

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

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 C: 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