Brian Fung - Project Portfolio

PROJECT: MooLah

Overview

This portfolio aims to document my contributions to MooLah

MooLah is a desktop expense tracking application, which was morphed from the code of Software Engineering Education's AddressBook - Level 3 used for teaching Software Engineering principles.

MooLah is written in Java and has around 23 kLoC. It aims to provide a streamlined expense tracking solution to NUS students who are more comfortable with Command-Line Interfaces (CLI). With features such as auto-completion, and syntax highlighting, and expense analysis features like budgeting, and statistics. MooLah provides users an expense tracking solution which is both useful and easy to use.

Summary of contributions

- **Major enhancement**: added **Easier Command-Line** feature.

As this project required the use of a Command-Line interface for user interaction, user input can sometimes require a lot of typing and users may find it difficult to remember all the different commands available and the syntax required to use these commands.

This feature consists of several features which make Command-Line easier to use, and to reduce the need for users to refer to the user guide.

Component 1: Aliases

- What it does:
  This feature allows the user to save input to an alias, allowing them to create shortcuts for command which they frequently require. This saves them time by reducing the keystrokes required to enter a command, as well as reduces the need to remember what exactly needs to be entered.

Component 2: Syntax highlighting

- What it does:
  This feature provides some validation of user input. It highlights valid command words used for commands as well as the prefixes delimiting arguments within the command. It also makes the user's input more readable, as commands may require many arguments and prefixes, it may be difficult for users to read their input before entering the command.
- Justification: This feature improves the product significantly because a user does not need to

remember confusing commands and syntax. Especially due to the fact that our application supports a large number of commands. This allows users to see what valid commands there are as well as what arguments are required without having to press enter the command and see an error message.

- Highlights: This feature required in-depth analysis of the command syntax of the original code base. The implementation was challenging as it required the use of regular expressions to analyse user input so that the right sections of text were properly highlighted. This feature also required integration of RichTextFx JavaFx extensions. As RichTextFx only provided a text styling for multi-line text areas, this feature required significant usage of JavaFx's event dispatch and in order to ensure that the text area can properly simulate a single-line textfield.
- Credits: The RichTextFx demo's included some existing logic for syntax highlighting which was modified for use in MooLash as the behaviour of syntax highlighting in the demo's code did not require different highlighting behaviour in different context.

Component 3: Input suggestions ** What it does:
As with the previous components, the main reason this feature was implemented was to reduce the need for users to have to refer to user guides or remember a lot of commands and their syntax. This feature suggests completions to the user's current input to provide them with possible commands they may enter, as well as provide them information of what arguments are required by the command and what the prefixes for the argument stand for.

- Highlights: This feature required in-depth analysis of the command syntax of the original code base. It required modification to the existing Prefix class for it to provide more information to the user. The implementation was challenging as MooLah supports almost 30 commands, all of which have different arguments. As such the feature needed to be designed such that it was scalable to support new commands as well as modification to existing commands. Similar to the syntax highlighting feature, this feature uses a significant use of JavaFx as well.

Component 4: Generic commands. As mentioned previously, MooLah supports around 30 commands, several of which do similar thing (e.g. There are several commands which add information to MooLah use the word 'add'. In order to differentiate these commands, they require a much longer name to be clear as to what the command does. This feature allows multiple commands to use the same word and be parsed differently based on the context they are used.

- **Major enhancement**: Design **GUI elements for Budget List, Budgets, Expense, and Aliases** Except for the most recent change to the colour theme of Moolah, most of the code to visualise changes to Aliases, Budgets, Expenses were written by me. #106 #231
- **Minor enhancement**: **initial refactoring of AddressBook into MooLah to allow tracking of expenses**
  While expenses trackers are similar to an address book such they keep a record of items, the requirements to store expenses are much different. The person class needed to be refactored quite heavily. Additionally, as Expense should be able to share a name unlike the behaviour of the existing Person class, a new way to differentiate Expenses needed to be implemented. #40 #42
- **Code contributed**: RepoSense link
- **Other contributions**:

- Project management:
  - Managed release for v1.3.2 (the release used for Practical Exam Dry Run) on GitHub
  - In charge of ensuring code is tested.
  - Designed mock up for inital GUI design #60
- Wrote tests to significantly increase coverage:
  - #271 65% to 80%
    #247 46% to 52%
- Documentation:
  Made changes to UG diagrams to match changes to the code #285
- Tools:
  - Integrated a third party library (RichTextFX) to the project #93
  - Integrated TestFx into the project for Testing GUI components. #271
    (* much of the code was sourced from AddressBook4 GUI-Tests)

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.

Making inputs quick and easy.

Do you have trouble remembering commands and what arguments they require? MooLah provides several features which will help you remember them and make your life much easier!

Creating a shortcut: `alias`

If you find yourself entering the same thing over and over, MooLah allows you to type less by assigning this command to an `alias`. This will allow you to type this `alias` in place of the original long command.

To assign an `alias`, use the the `addalias` command with the following format:

```
addalias a/<ALIAS_NAME> i/<INPUT>
```

There are two kinds of alias you can make, aliases which act as a standalone command, or an alias which accepts arguments.

Variation 1: Standalone

You can store an entire command using an `alias`, and then use this `alias` in place of that command. For example:

```
addalias a/chicken i/ addexpense d/ chicken rice p/2.30 c/food
```

This saves the command `addexpense d/ chicken Rice p/2.30 c/food` to `chicken`. Subsequently, you may use this alias in place of using the full command.

Variation 2: with arguments

You may also save an incomplete input to an `alias`. For example:

```
addalias a/ addfood i/ addexpense c/Food
```

Subsequently, entering the following:

```
addfood d/chickenrice p/2.30
```

is equivalent to entering:

```
addexpense c/Food d/chickenrice p/2.30
```

- Alias names can only contain alphanumeric characters.
- Alias names cannot be a command word used by a built-in command, e.g. you may not save a command to an alias named "addexpense".
  For the list of built-in command words, see: [Command Summary].
- Only one input may be saved to each alias name. Saving an input to an alias name which already exists will overwrite the existing input if it exists.
- Repeated prefixes are not allowed! 'a/' and 'i/' may only be used once.

Listing the shortcuts you have saved: `listalias`

To list all of the aliases you have saved, you can the the `listalias` command. Alternatively, you may use the `view` command by typing `view Aliases`. Either of these will bring you to the *User Defined Aliases* panel where you can see the list of aliases you have created.

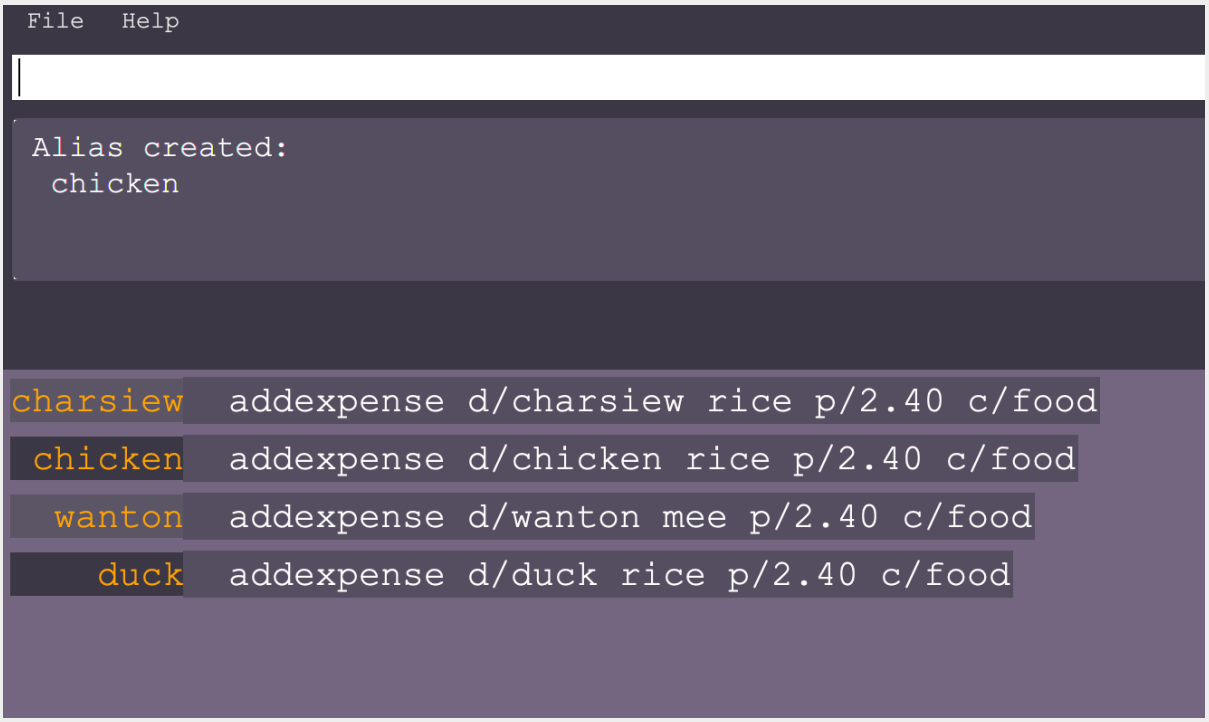

Figure 1. User Defined Aliases panel

The *user defined aliases* showing some valid aliases.

Deleting the shortcuts you no longer want to use: `deletealias`

To delete an `alias` you no longer wish to use, you may use the `deletealias` command.

This is the format:

```
deletealias <ALIAS_NAME>
```

This will delete the `Alias` with the name you specified. For example:

```
addalias a/hello i/helloworld
deletealias hello
```

This will delete the `hello` alias from your saved aliases.

Autocomplete and Suggestions

If you find yourself forgetting the syntax for some commands, enable the suggestion feature. This will enable MooLah to suggest valid commands, as well as prefixes when you already have a command word entered.

Enable
To enable the suggestion feature, press the `TAB` key. This will enable the suggestion feature! You will see that the command box has a green border. This indicates that the feature is enabled.

Disable
To disable the suggestion feature, press the TAB key again. This will disable the suggestion feature! You will see that the command box no longer has a green border. This indicates that the feature is no longer enabled.

The suggestion menu will show you the command words which match your current input.

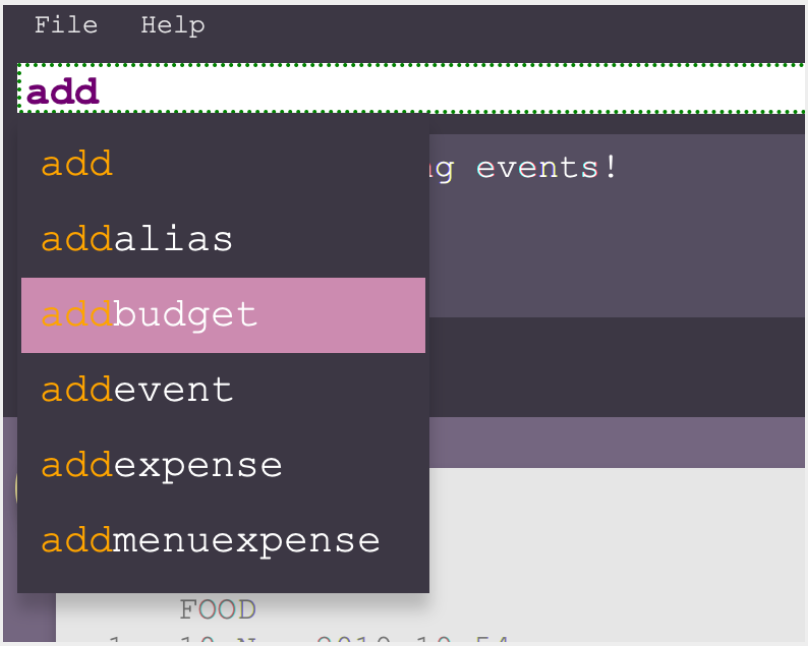

Figure 2. suggestions for command words beginning with "add"

The suggestion menu showing suggestions for built-in commands which begin with 'add' when the feature has been enabled

You can cycle through the choices using the UP and DOWN keys. To confirm your choice, push the ENTER or SPACE key! LEFT, RIGHT, and ESC can be used to close the menu without disabling suggestions.

When you have entered a valid command, MooLah will show you a list of prefixes that are supported by this command and that you have not yet entered.

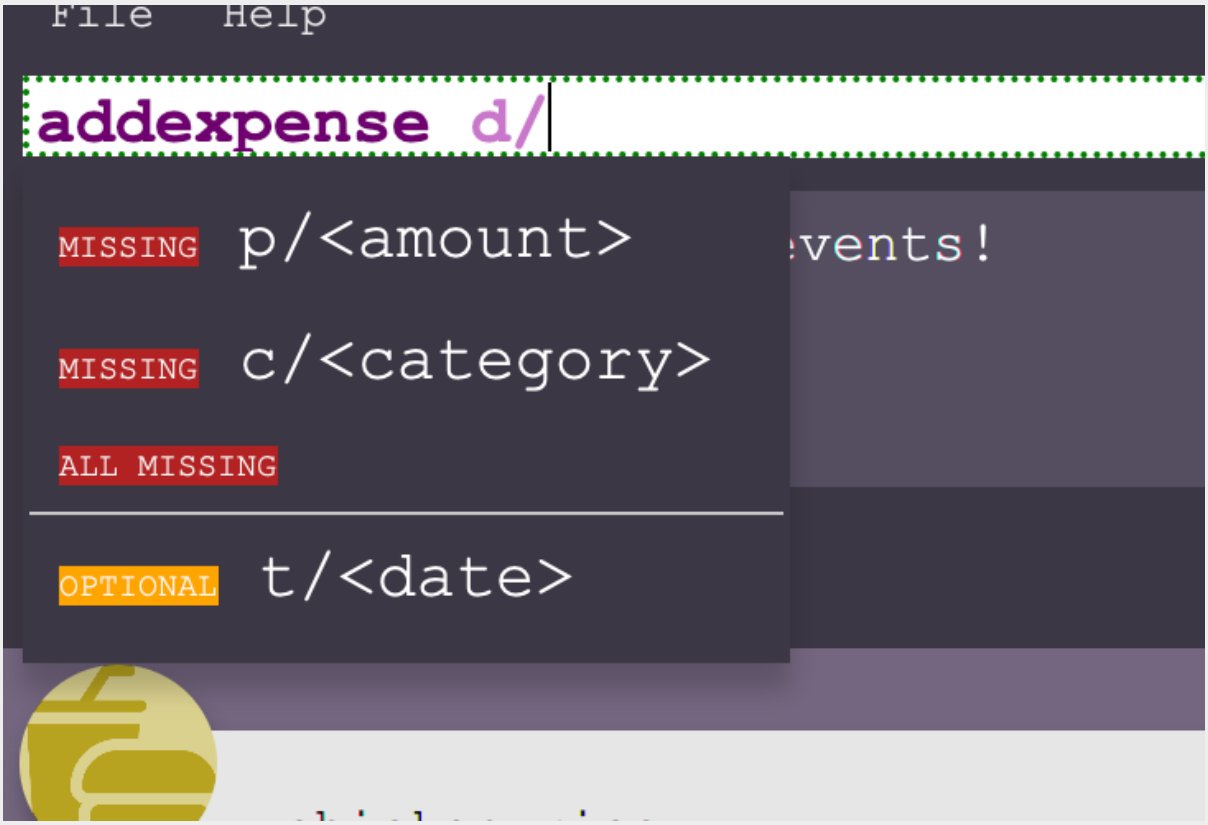

Figure 3. suggestions for "addexpense" command

The suggestion menu showing the missing prefixes in the current input, "addexpense d/", along with what the prefix represents.

When the suggestion menu shows you the prefixes that you are missing, you may see these 3 tags:

- MISSING indicates that this prefix is mandatory and you have yet to include it.
- ALL MISSING represents all the missing mandatory prefixes.
- OPTIONAL indicates that this prefix is missing, but is not compulsory to enter.

Syntax Highlighting

MooLah will highlight valid command words and prefixes, as well as the arguments that will be used with that argument.

When you are entering a command, MooLah will highlight command words which are supported built-in commands.

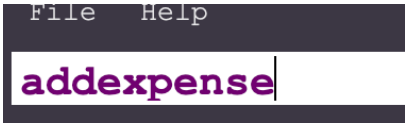

Figure 4. addexpense is valid and highlighted

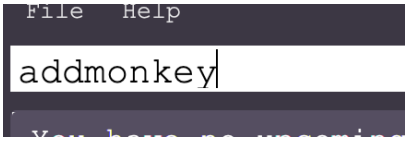

Figure 5. addmonkey is not built-in and not highlighted

After entering a valid command word, MooLah will also highlight the prefixes that are supported by this command. However, it will not validate them for you!

Figure 6. Highlighting of prefixes and arguments for the add.

Figure 7. Wrong arguments are also highlighted.

Input history

MooLah remembers what commands you have previously entered, and allows you to access them to use them again.

If you need to enter the same input multiple times but do not want to save it as an `alias` you may cycle through the history of successfully executed inputs within the current session.

Press `UP` to scroll through previously executed commands.
Press `DOWN` to go back the more recent commands.

For example, if you entered these commands previously.

```
> addexpense d/chickenrice p/12.3 c/food
> addexpense d/chickenrice p/12.3 c/food
> add d/chicken rice p/2.30 c/food
```

Pressing the `UP` key to quickly enter the previous input.

Figure 8. the last input which was successfully executed will be entered into the command box.

NOTE	When the suggestion menu is open, the `UP` and `DOWN` keys will cycle through the suggestions instead!

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.

Alias feature

The Alias feature allows users to assign inputs they may use very often to a shortcut, and execute the input by entering the shortcut, (a.k.a `Alias`), in place of the full or partial command.

Implementation

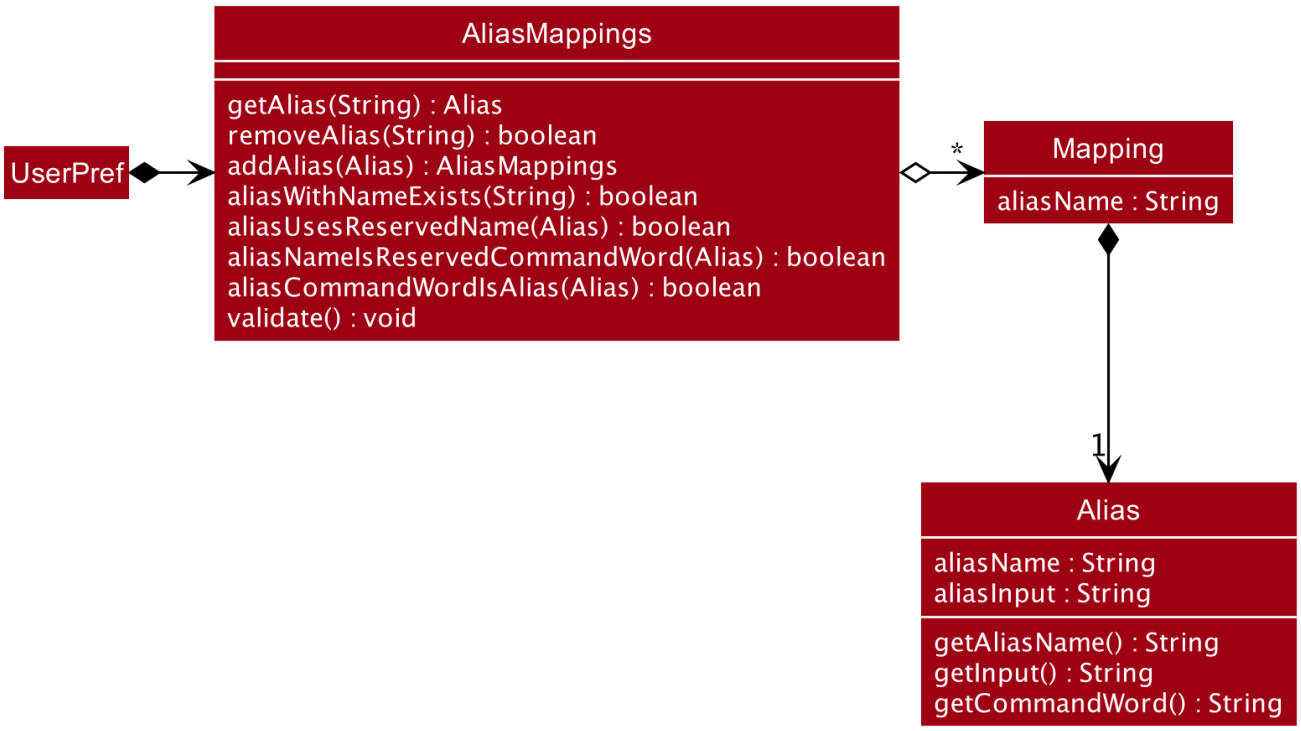

Figure 9. AliasMappings Class Diagram

These user defined `Alias`es are saved in an `AliasMappings` object within `UserPref` as seen in the above diagram. Internally, the `AliasMappings` object stores an `Alias` in a Map<Strings, Alias> object. With the addition of `AliasMappings` object to `UserPref`, `UserPref` supports these additional operations:

- `UserPref#addUserAlias(Alias)` — Saves a specified `Alias` to the user preferences for future use.
- `UserPref#hasAlias(String)` — Query if the there is an `Alias` with this name already defined.
- `UserPref#getAlias(String)` — To get an `Alias` with this name if it exists.
- `UserPref#aliasNameIsReservedCommandWord(Alias)` — To query if this `Alias` is uses a name which clashes with existing built-in commands.
- `UserPref#aliasCommandWordIsAlias(Alias)` — To query if this `Alias input` begins with another `Alias`, this is used to validate that an `Alias` will not cause an infinite loop by chaining multiple aliases in a loop.
- `UserPref#getAliasMappings()` — To access the `Alias` saved by the user.

- `UserPref#setAliasMappings(AliasMappings mappings)` — To overwrite all the `Alias` saved by the user.

Alias creation

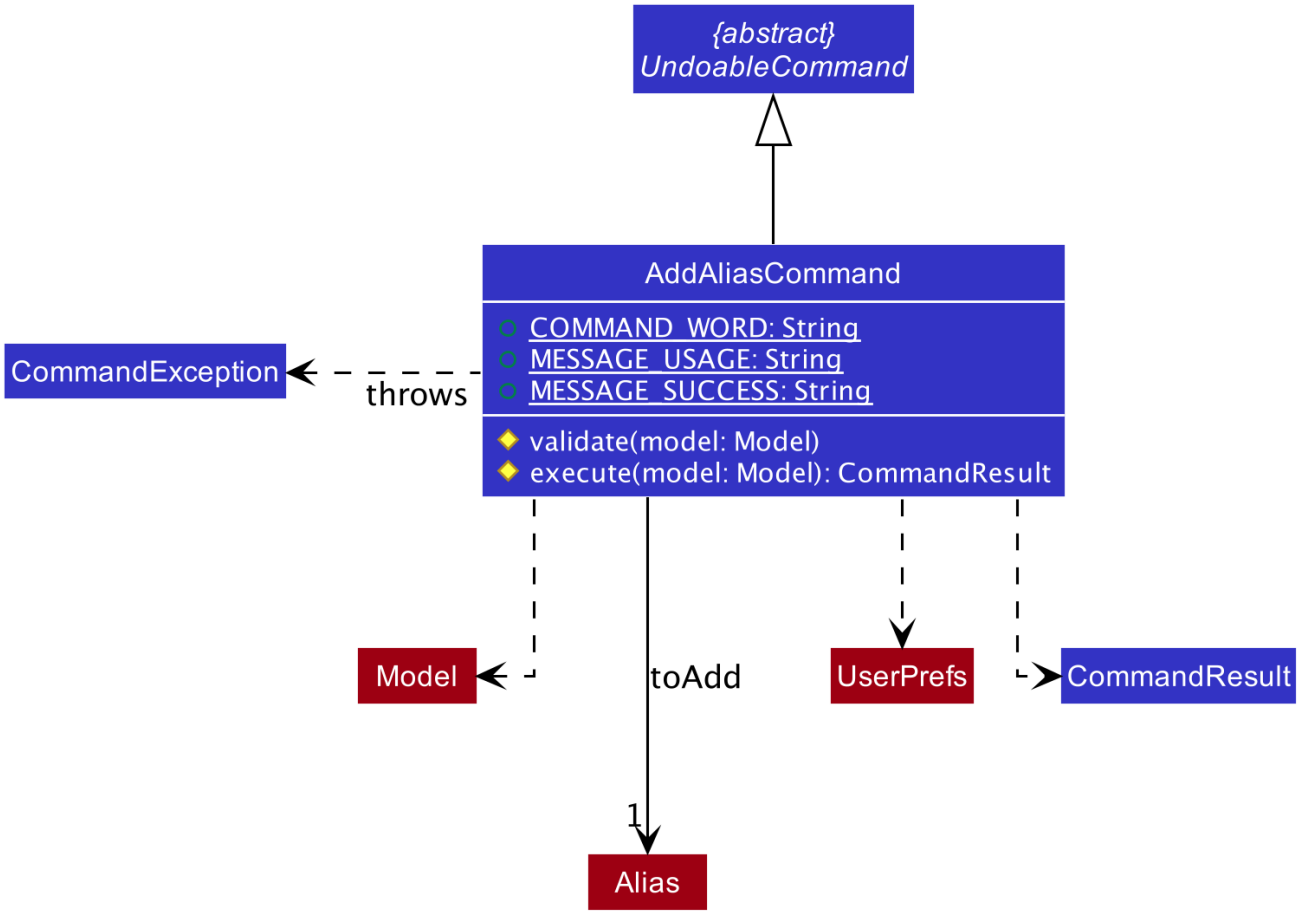

Figure 10. AddAliasCommand Class Diagram

In order for the user to save an `Alias`, they first define it using the `AliasCommand`. The `AliasCommand` command extends `UndoableCommand` to allow users to undo defining an `Alias`. The following sequence diagram describe in more detail how an `Alias` is added.

> Note:
> Alias and AliasMapping are in Model and not Core. This change has yet to be reflected in the following sequence diagrams.

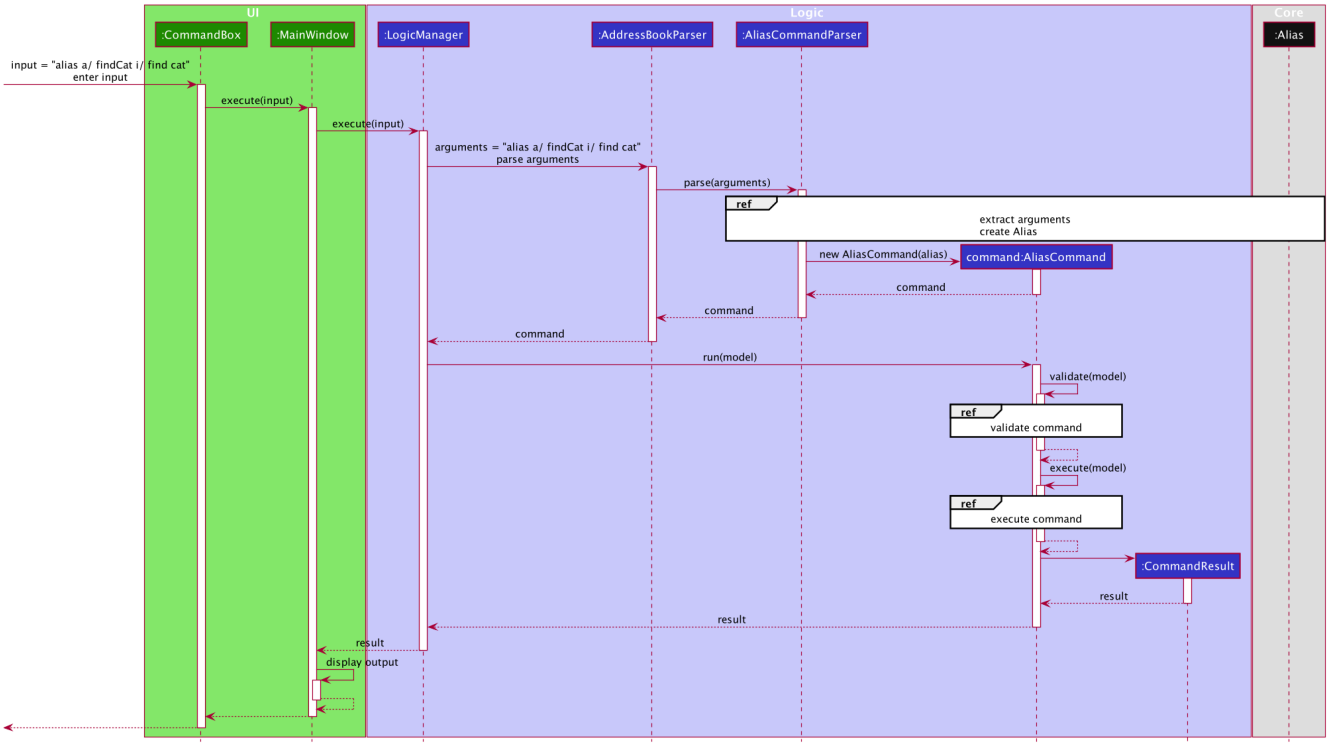

Figure 11. AddAliasCommand execution Sequence Diagram

1. 1. The user enters a command with the following syntax `alias a/ <name> i/ <input>`.
2. 2. The `UI` passes this command string to the `LogicManager` which passes it onto the `MooLahParser`.
3. 3. The parser extracts the argument string and passes it to an `AliasCommandParser`.

ref : extract arguments create alias

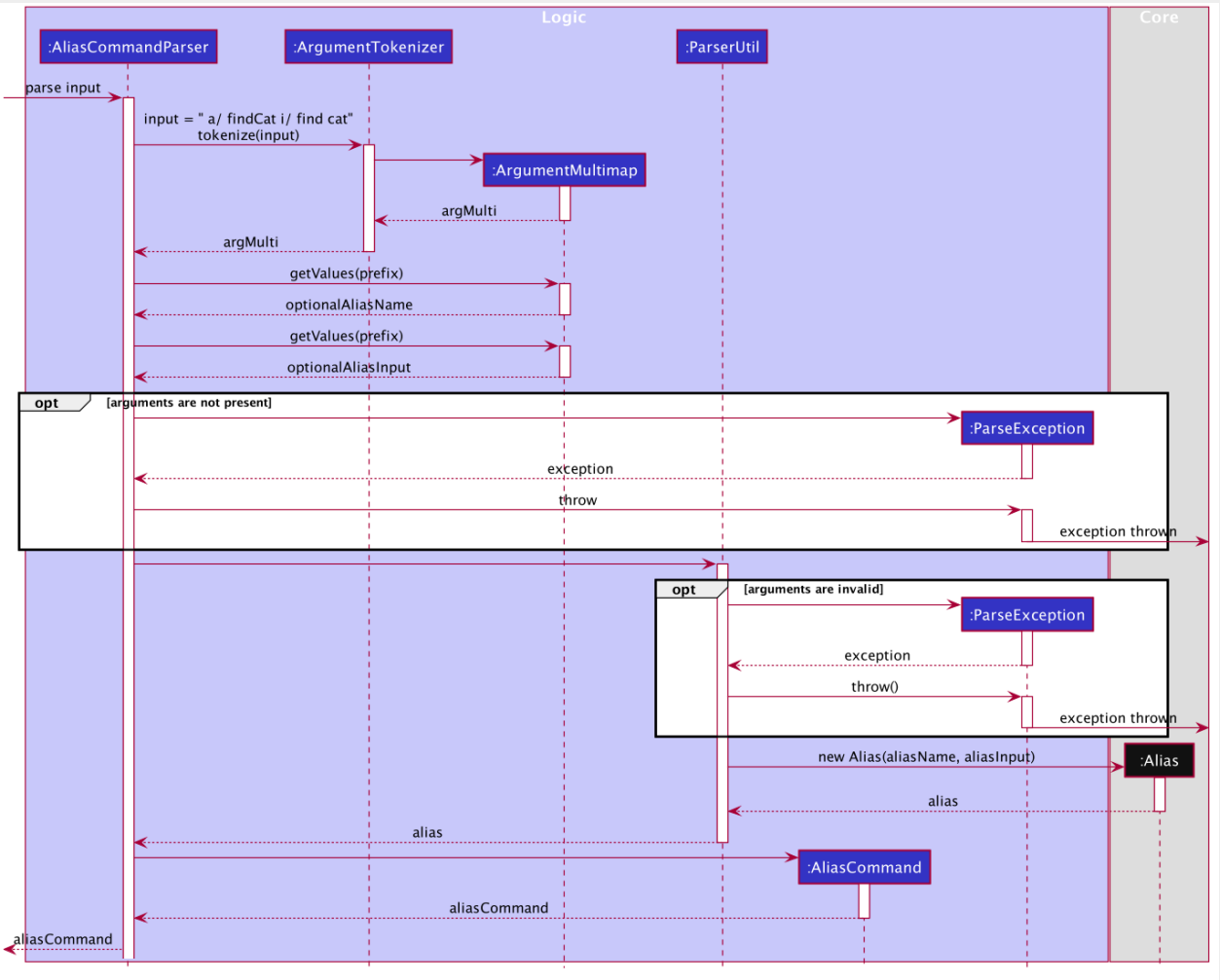

Figure 12. AliasCommandParser Sequence Diagram

1. 4. The `AliasCommandParser` uses the `ArgumentTokenizer` to tokenize the argument string and extract the `alias name` and `input` fields into an `ArgumentMultimap`.
2. 5. The arguments are obtained and to create a new `Alias` using the the `Alias` parser in `ParserUtil`.
3. 6. An `AliasCommand` is created containing this new `Alias` to add to the `UserPref`.
4. 7. This is passed back to the `LogicManager` to call `AliasCommand#run()`.

ref: validate command

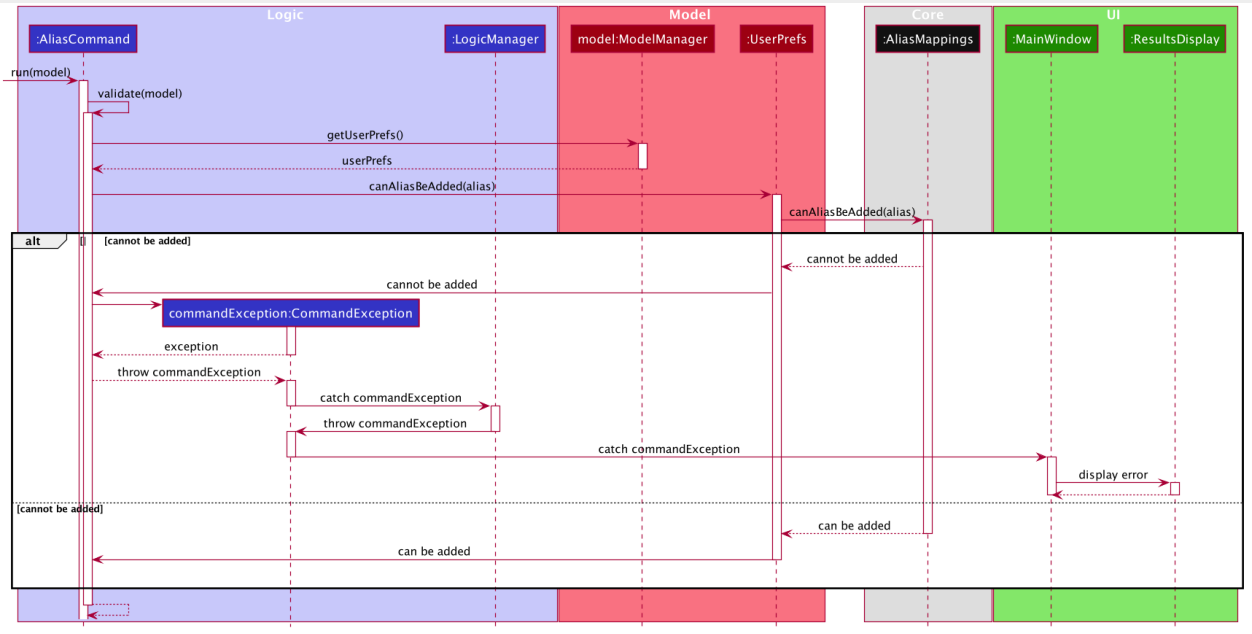

Figure 13. AliasCommand #Validate Sequence Diagram

1. 8. The `AliasCommand` is validated using `AliasCommand#validate()`. The `Alias` is checked to ensure it does not
    - a. Have a clashing name used by an existing `Command` as a CommandWord.
    - b. Have an input beginning with a supported `Alias`.
2. If is not valid, handled exception is thrown.

ref: execute command

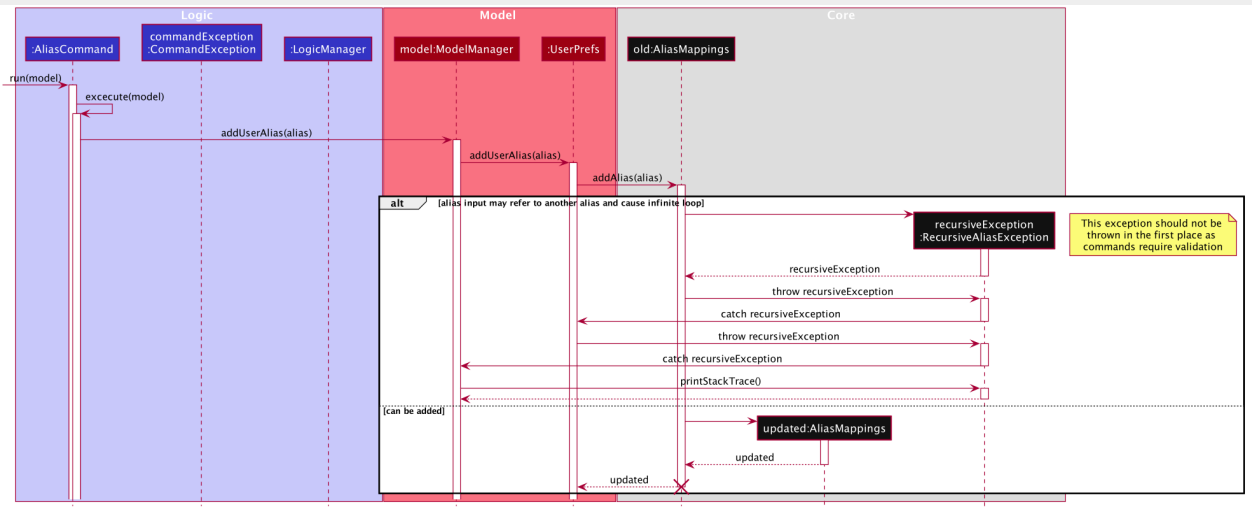

Figure 14. AliasCommand #execute Sequence Diagram

1. 9. If it was validated that the `Alias` can be added.
2. 10. The `Alias` is then added to the `AliasMappings` object within `UserPref`.

3. 11. The `Alias` is now usable by the user.

Usage of aliases in input

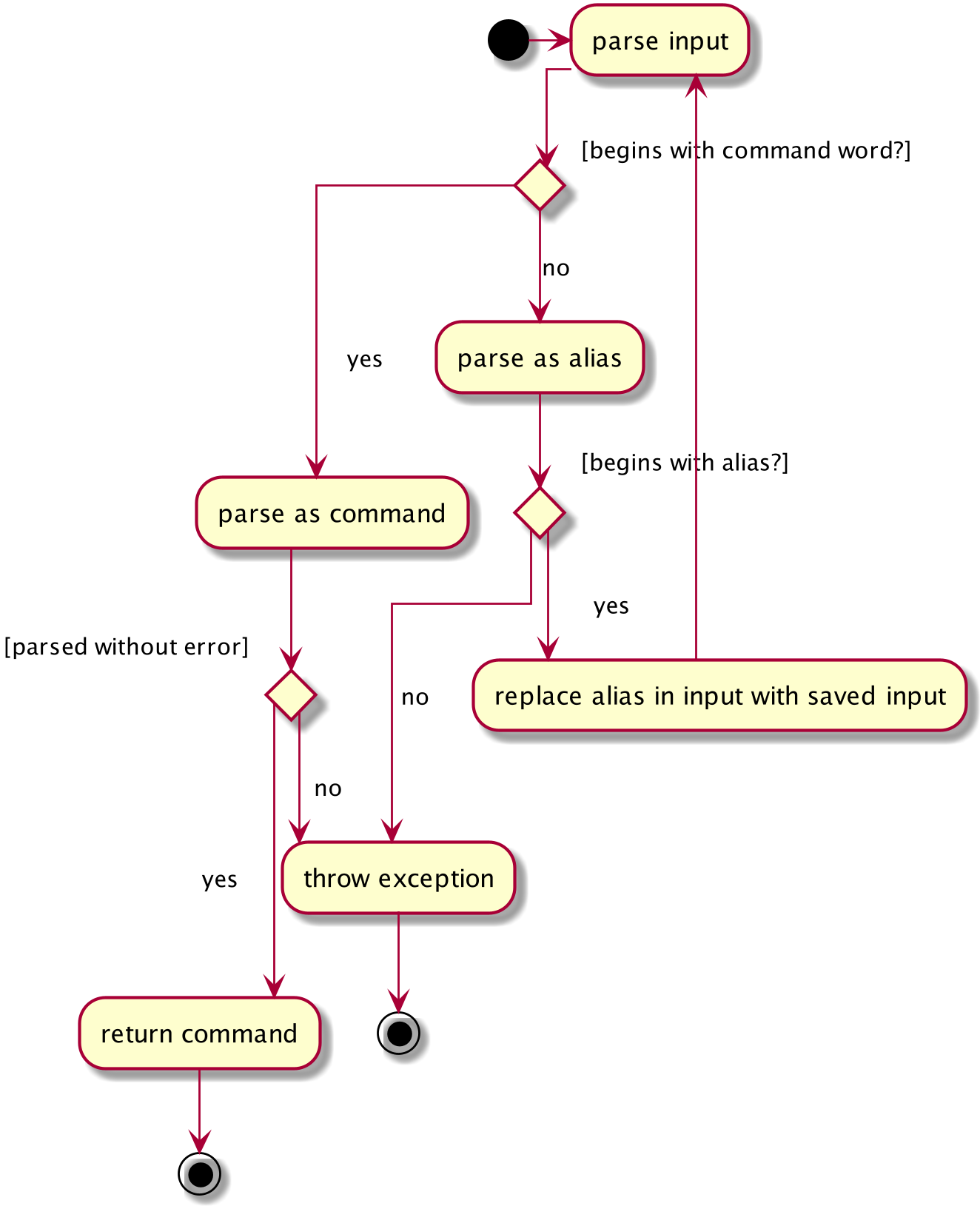

Figure 15. Activity Diagram showing the a high level view of the events that occur when parsing a command.

When a user enters an `input` to be executed, the `MooLahParser` will do the following:

1. Attemps to parse the `input` as an input which begins with a valid `CommandWord`
2. If that fails, it will try to parse it as an input which begins with an `Alias`.
    a. If it successfully does so, it replaces the alias in the original `input` with the `input` stored in the `Alias`.
    b. Finally, the `MooLahParser` re-parses the modified `input`.
3. If this too fails, an exception is thrown indicating that the command was invalid

Design Considerations

Potential Recusive aliases

As users may modify the data files of MooLah, they may modify the aliases directly. As such it was necessary to validate that the modified aliases will not cause infinite recursion. This is done by traversing between aliases which chain to each other and ensuring that none of them lead back to themselves. Otherwise, if it is detected that an alias can loop to itself, the alias data is reset.

Aspect: Why can an `Alias` only be used at the beginning of an input versus anywhere within an input.

- **Alternative 1(Chosen):** Beginning only
    - Pros: Easier to determine which word is the shortcut.
    - Pros: Easier to detect recursion due to alias chaining.
    - Cons: Less flexible in term how the shortcut can be used i.e. it can only replace or prefix an input.
- **Alternative 2:** Anywhere in input
    - Pros: A wider variety of shortcuts can be defined by the user
      e.g. `add deckChicken 2.50`, where `deckChicken` maps to `d/ chicken rice c/ food p/`.
    - Cons: Harder to detect recursion due to alias chaining.
    - Cons: Parsing becomes more complicated and alias words become unusable in other contexts. e.g. defining an `Alias` mapping `bus` to `sbs bus` prevents an input such as `smrt bus` from being parsed properly as it would be replaced with `smrt sbs bus` by the parser.
- **Solution (Current Implementation):** The biggest factor in choosing `Alternative 1` is to make it easier to prevent possible recusion due to alias chaining, which would potentially be a fatal bug in the application. Additionally, while it may be possible to determine which one the user means judging by the context in which it was used. However, that is far out of the scope of the module and would require much more processing of user input.

Another alternative would have been to have an alias name follow the following convention `'aliasName'` (i.e. some character before and after) clear what is an `Alias` within an input.

The purpose behind this feature was to make CLI input less troublesome, so users do not need to type in the full input string to carry out commands they may use frequently. In v2.0 we want to look into using data analysis track users' input habits in order to make suggestions on possible shortcuts or Aliases they may find convenient to have.