Niven Ang Jia Hao - Project Portfolio

PROJECT: TravelBuddy

1. Overview

This project portfolio documents my contribution to **TravelBuddy**, a travel log application developed as a team project for the module CS2103T Software Engineering in the National University of Singapore. As part of the module objectives, a brown-field approach was adopted where we were tasked to morph the given AddressBook into the current TravelBuddy, a travel log application for travel enthusiasts to record and manage previously visited places.

My role in this project was to design and write the code for the `search` feature. The following sections showcase, in greater detail, my contributions to TravelBuddy, as well as the documentation of my contributions in the User Guide and Developer Guide.

2. Summary of Contributions

2.1. Major Enhancement

The major enhancement I added was the ability to perform search by name, rating, tags, country and year on the places in TravelBuddy.

What it does: The search feature allows the user to search TravelBuddy for places where the requested field (name, rating, tags, country or year) matches the user input.

Justification: TravelBuddy can get long and cluttered when more places are added. This feature improves TravelBuddy as it allows users to filter through the long place list to obtain the relevant data that they need.

Highlights: This feature helps users search for what they need. Multiple keywords can be included in the search to obtain a wider scope of results. The challenge lies in the parsing of the user input, which is tested against a predicate for matching keywords. As user input is taken in, defensive programming is adopted to restrict user input to a fixed regular expression depending on the parameter involved to prevent abuse. This verified input is then tested against a predicate that is tailored to suit the regular expression of each parameter.

An in-depth analysis of design alternatives in keyword matching and the data structure used for the search feature was done to obtain the most optimal solution.

Code contributed: RepoSense | Functional Code | Test Code

2.2. Minor Enhancement

The minor enhancement I did was to convert the `Phone` parameter in the previous addressbook to

the `Rating` parameter for the `Place` class to track ratings of each place in TravelBuddy.

In addition, I added the `delete multiple` feature to delete multiple entries in TravelBuddy with one command.

Code contributed to minor enhancement: Convert Phone to Rating | Delete Multiple Feature

2.3. Other Contributions

Given below is a list of other contributions that I made to the project:

- **Project management**
  - Helped to streamline issue tracking using milestone labels and assignees on Github.
  - Added user stories to help with implementation of features.
  - Managed project release v1.3.3 on GitHub.
- **Enhancements to existing features**
  - Updated and wrote tests to improve coverage: #50, #115, #179
- **Documentation**
  - Updated AboutUs with the responsibilities of team members: #12
  - Updated the User Guide with features I added, FAQ section and command cheatsheet: #53, #63, #77, #85, #101, #174
  - Updated the Developer Guide with features I added and diagrams for illustration purposes: #68, #85, #101, #162, #174
  - Fixed documentation errors found during testing: #135
- **Community**
  - Reviewed pull requests (with non-trivial review comments): #84, #188, #189
  - Reported bugs and helped to fix bugs: #82, #118, #161
  - Tested other projects and reported bugs: #223, #225, #227, #231, #238, #240, #246, #248

3. Contributions to the User Guide

We updated the User Guide of TravelBuddy with information on the use of the implemented enhancements.

> *Given below is the start of an excerpt from the User Guide, showing the additions that I made for the search feature. They showcase my ability to write documentation targeting end-users.*

3.1. Searching places by name: `search`

Description: The `search` (Shortcut: `se`) command searches for places whose names contain any of the given keywords.

Format: `search KEYWORD [MORE_KEYWORDS]`

Preconditions: Given below are preconditions that must be met for the `search` command to work:

- The search is case insensitive. e.g `national` will match `National`.
- The order of the keywords does not matter. e.g. `University National of Singapore` will match `National University of Singapore`.
- Only full words will be matched. e.g. `Nation` will not match `National`
- Places matching at least one keyword will be returned (i.e. `OR` search). e.g. `National Museum` will return `National Museum of Singapore` and `National University Hospital`.

Example: `search Singapore`
Executes a search for places that contain the keyword `Singapore` in its name.
In Figure 4.6.1, running the `search` returns places that contain `Singapore` in their names:

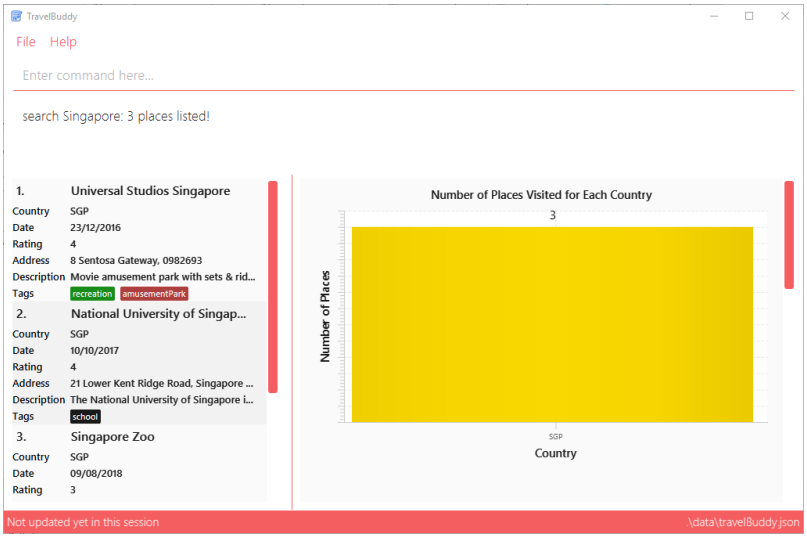

Figure 4.6.1: Search results of `search Singapore`

3.2. Searching places by ratings: `searchr`

Description: The `searchr` (Shortcut: `sr`) command searches for places whose ratings match the specified rating from 1 to 5.

Format: `searchr INDEX [MORE_INDICES]`

Preconditions: Given below are preconditions that must be met for the `searchr` command to work:

- The rating used in the search must be an integer ranging from 1 to 5. e.g `searchr 5` will return places with 5-star ratings.
- Multiple indices can be included in the query. e.g `searchr 4 5` will return places with `4` or `5` star ratings.

Example: `searchr 4`
Executes a search for places with a rating of `4`.
From Figure 4.7.1 below, using `searchr 4` will return all places in your TravelBuddy that have a rating of `4`.

Figure 4.7.1: Search results of `searchr 4`

3.3. Searching places by tags: `searcht`

Description: The `searcht` (Shortcut: `st`) command searches for places whose tags correspond to any given keywords.

Format: `searcht KEYWORD [MORE_KEYWORDS]`

Preconditions: Given below are preconditions that must be met for the `searcht` command to work:

- The search is case insensitive. e.g `Temple` will match `temple`.
- Only full words will be matched e.g. `temp` will not match `temple`.
- Places tagged with at least one matching keyword will be returned (i.e. `OR` search). e.g. `temple school` will return places tagged with `temple` or `school`.

Example: `searcht distillery`
Executes a search for places that are tagged with `distillery`.
From Figure 4.8.1 below, using `searcht distillery` will return all places in your TravelBuddy that are tagged with `distillery`.

Figure 4.8.1: Search results of `searcht distillery`

3.4. Searching places by country: `searchc`

Description: The `searchc` (Shortcut: `sc`) command searches for places whose country matches the specified ISO-3166 3-letter country code.

Format: `searchc KEYWORD [MORE_KEYWORDS]`

Preconditions: Given below are preconditions that must be met for the `searchc` command to work:

- The country code keywords for `searchc` must be valid 3-letter ISO-3166 country codes. e.g `JPN` will return places located in Japan.
- Multiple keywords can be included in the query, i.e. `searchc JPN CHN` will return places located in Japan or China.

Example: `searchc JPN`
Executes a search for places located in `JPN` (Japan).
From Figure 4.9.1 below, using `searchc JPN` will return all places in your TravelBuddy that are located in Japan.

Figure 4.9.1: Search results of `searchc JPN`

3.5. Searching places by year of visit: `searchyear`

Description: The `searchyear` (Shortcut: `sy`) command searches for places whose year of visit matches the specified year of interest.

Format: `searchyear KEYWORD [MORE_KEYWORDS]` OR `searchyear KEYWORD-KEYWORD`

Preconditions: Given below are preconditions that must be met for the `searchyear` command to work:

- The search year is bounded from 1900 to the current year.
- The year keywords for `searchyear` can be entered as a range. e.g `2010-2017` will return all the places visited from `2010` to `2017`.

Example: `searchyear 2016`

Executes a search for places visited in the year `2016`.
From Figure 4.10.1 below, using `searchyear 2016` will return all places in your TravelBuddy that you visited in the year `2016`.

Figure 4.10.1: Search results of `searchyear 2016`

This marks the end of the excerpt from the User Guide.

4. Contributions to the Developer Guide

We updated the Developer Guide of TravelBuddy with the logic behind the implementation of the enhancements.

Given below is the start of an excerpt from the Developer Guide, showing the additions that I made for the search feature. They showcase my ability to write technical documentation and the technical depth of my contributions to the project.

4.1. Search Feature

4.1.1. Current Implementation

The `search` command provides functionality for users to search for places in TravelBuddy that contain the specified input. The user's input is split into separate keywords and matched by a `Predicate` to the list of places in TravelBuddy. Places with matching keywords will be displayed as a list on the GUI.

The search feature comprises of the following search commands:

- Search by Name: `search`
- Search by Rating: `searchr`
- Search by Tags: `searcht`
- Search by Country: `searchc`
- Search by Year: `searchyear`

NOTE The various `search` commands are in lower-case. Mixed-case or upper-case commands are not recognised by the application.

Figure 4.2.1.1 below shows the class diagram of the `search` mechanism and its associations to the other classes in the Logic component.

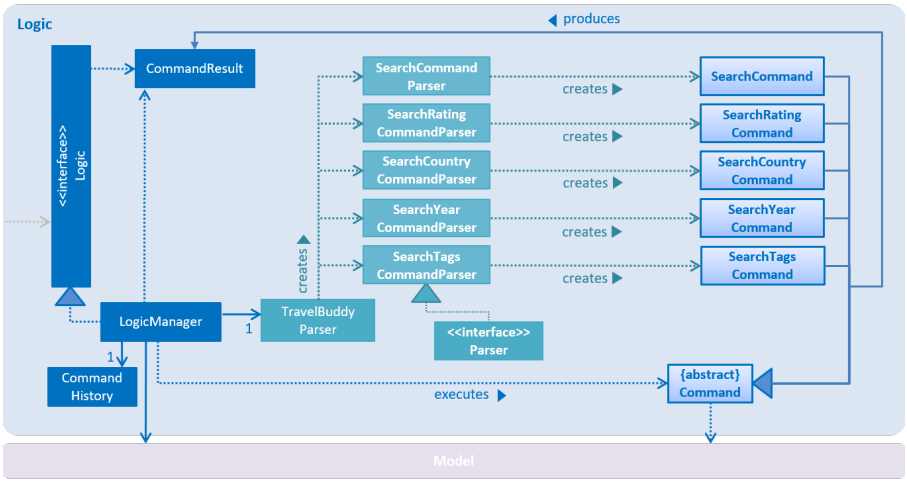

Figure 4.2.1.1: Class Diagram for `search` command.

NOTE The various search features (i.e. search name, country code, rating, tags, year) function on a similar concept, differing only in the `Parser` which is called using different command words and the `Predicate` to filter arguments.

Given below is a usage scenario for the search feature and is based on the search name feature.

The following sequence diagram, Figure 4.2.1.2, shows how the search feature works:

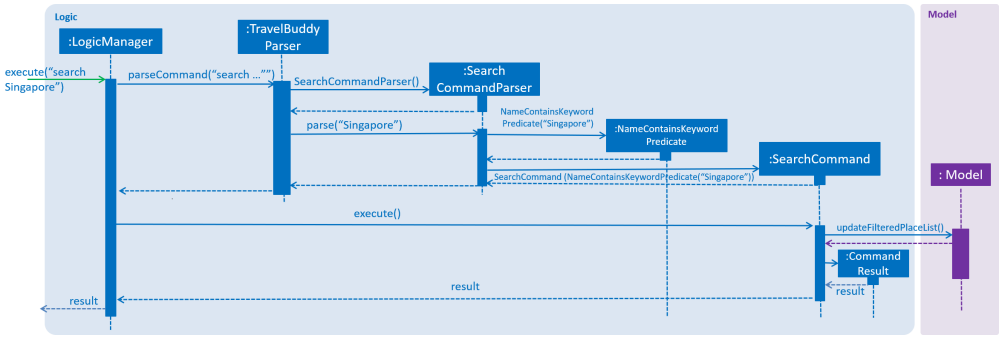

Figure 4.2.1.2: Sequence Diagram for `search` command.

The control flow of the sequence diagram above is as follows:

1. Initially, a user enters a command with the command word `search` followed by argument(s).
2. `LogicManager` receives the `execute` command and calls the `parseCommand` method in `TravelBuddyParser`.
3. `TravelBuddyParser` parses `search` as the command and a `SearchCommandParser` will be instantiated to further parse the command.
4. `SearchCommandParser` receives the arguments if the command word input matches the command word of any search command.

- The argument string is split into an array of keywords shown in the code snippet below.

```
String[] nameKeywords = trimmedArgs.split("\\s+");
return new SearchCommand(new
    NameContainsKeywordsPredicate(Arrays.asList(nameKeywords)));
```

- A NameContainsKeywordPredicate will be instantiated with the array of arguments as the predicate, which will be used to check if any of the places in TravelBuddy matches the user's input.

```
public boolean test(Place place) {
    return keywords.stream().anyMatch(keyword -> StringUtil
        .containsWordIgnoreCase(place.getName().fullName, keyword));
}
```

5. Subsequently, `SearchCommandParser` creates a `SearchCommand` object with the predicate and returns it to `LogicManager`.
6. Following that, `LogicManager` calls the `execute` method of `SearchCommand`, shown in the code snippet below.

```
public CommandResult execute(Model model, CommandHistory history) {
    requireNonNull(model);
    model.updateFilteredPlaceList(predicate);
    return new CommandResult(constructFeedbackToUser(model));
}
```

7. `SearchCommand` updates the list in `Model`, which will be displayed in the GUI.
8. `SearchCommand` instantiates a `CommandResult` object and passes it to `LogicManager` .

The activity diagram, Figure 4.2.1.3, below summarises what happens when a user inputs a search command:

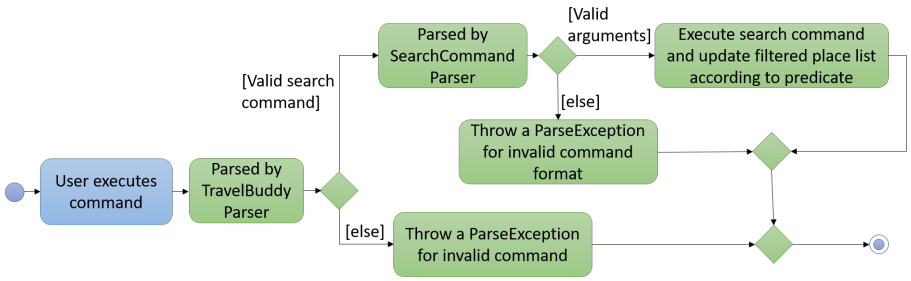

Figure 4.2.1.3: Activity Diagram showing the process flow when a search command is issued.

Search Name Feature

The command word for search name is `search` and is parsed by `TravelBuddyParser`. The name arguments are then passed into `SearchCommandParser` and are stored in a list of keywords before being passed into `NameContainsKeywordsPredicate`. The list is converted into a stream and

individually matched to the names of each entry in TravelBuddy.

The search name mechanism is facilitated by `SearchCommand`, which extends `Command` with a predicate that specifies the conditions of the name of the place to be chosen from TravelBuddy.

Search Rating Feature

The command word for search rating is `searchr` and is parsed by `TravelBuddyParser`. The rating arguments are then passed into `SearchRatingCommandParser` and are checked for validity before being stored in a list of keywords. The list is then passed into `RatingContainsKeywordsPredicate`, where it is converted into a stream and individually matched to the rating of each entry in TravelBuddy.

The search rating mechanism is facilitated by `SearchRatingCommand`, which extends `Command` with a predicate that specifies the conditions of the rating of the place to be chosen from TravelBuddy.

Search Tags Feature

The command word for search tags is `searcht` and is parsed by `TravelBuddyParser`. The tag arguments are then passed into `SearchTagsCommandParser` and are stored in a list of keywords before being passed into `TagContainsKeywordsPredicate`. The list is converted into a stream and individually matched to the tags of each entry in TravelBuddy.

The search tags mechanism is facilitated by `SearchTagsCommand`, which extends `Command` with a predicate that specifies the conditions of the tags of the place to be chosen from TravelBuddy.

Search Country Feature

The command word for search country is `searchc` and is parsed by `TravelBuddyParser`. The country code arguments are then passed into `SearchCountryCommandParser` and are checked for validity before being stored in a list of keywords. The list is then passed into `CountryCodeContainsKeywordsPredicate`, where it is converted into a stream and individually matched to the country code of each entry in TravelBuddy.

The search country mechanism is facilitated by `SearchCountryCommand`, which extends `Command` with a predicate that specifies the conditions of the country code of the place to be chosen from TravelBuddy.

Search Year Feature

The command word for search year is `searchyear` and is parsed by `TravelBuddyParser`. The year arguments are then passed into `SearchYearCommandParser` and are checked for validity before being stored in a list of keywords. The list is then passed into `YearContainsKeywordsPredicate`, where it is converted into a stream and individually matched to the year of visit of each entry in TravelBuddy.

The search year mechanism is facilitated by `SearchYearCommand`, which extends `Command` with a predicate that specifies the conditions of the year of visit of the place to be chosen from TravelBuddy.

4.1.2. Design Considerations

Aspect: Designing how search executes

Given below is a comparison between the alternatives of the `search` mechanism design.

	Alternative 1 (current choice)	Alternative 2
Description	Matches entire keyword.	Check if the place contains the argument string.
Pros	**Speed**: This approach is faster in processing speed and computationally less intensive. **Refined results**: This approach provides more refined results as it narrows down the search scope to the user's query.	**Flexible**: This approach supports a search for partial keywords, so that users do not have to type the full keyword.
Cons	**Inflexible**: This approach is unable to support a search for partial keywords and may prove to be restrictive for certain users.	**Unrefined results**: This approach returns a longer list of results, which may defeat the purpose of filtering the list through search.

Decision: Alternative 1 of matching the entire keyword is adopted as it reduces processing time during keyword matching. In addition, it narrows down the search options by only returning keywords that matches the search query, which is the main objective of the `search` feature.

Aspect: Data structure to support search commands

Given below is a comparison between the alternatives of the data structure used in `search`.

	Alternative 1 (current choice)	Alternative 2
Description	Use a list to store the user input keywords and places.	Use `HashMap` to map keywords to each place.
Pros	**Ease-of-implementation**: Easy for new Computer Science student undergraduates to understand, who are likely to be the new incoming developers of our project. **Refined results**: This approach provides more refined results as it narrows down the search scope to the user's query.	**Faster search**: Faster searching as HashMap lookup runs in O(1) time.
Cons	**Slower search**: This approach is less efficient as the entire list needs to be searched through.	**Memory-consuming**. This approach requires more memory as a separate HashMap needs to be stored.

Decision: Alternative 1 of using a list is preferred as it uses less memory compared to alternative 2. Moreover, future contributors to the project are likely to be student undergraduates, so a simple data structure would be more optimal for educational purposes.

This marks the end of the excerpt from the Developer Guide.