TravelBuddy - Developer Guide

1. Introduction . . . 1
2. Setting Up . . . 1
2.1. Prerequisites . . . 1
2.2. Setting Up the Project in Your Computer . . . 1
2.3. Verifying the Setup . . . 2
2.4. Configurations to do Before Writing Code . . . 2
3. Design . . . 3
3.1. Architecture . . . 3
3.2. UI Component . . . 5
3.3. Logic Component . . . 7
3.4. Model Component . . . 8
3.5. Storage Component . . . 10
3.6. Common Classes . . . 10
4. Implementation . . . 10
4.1. Undo/Redo Feature . . . 11
4.2. Search Feature . . . 15
4.3. Add Feature . . . 28
4.4. Photos Feature . . . 32
4.5. Chart Feature . . . 36
4.6. Logging . . . 41
4.7. Configuration . . . 42
5. Documentation . . . 42
5.1. Editing Documentation . . . 42
5.2. Publishing Documentation . . . 42
5.3. Converting Documentation to PDF Format . . . 42
5.4. Site-wide Documentation Settings . . . 43
5.5. Per-file Documentation Settings . . . 44
5.6. Site Template . . . 44
6. Testing . . . 45
6.1. Running Tests . . . 45
6.2. Types of Tests . . . 45
6.3. Troubleshooting Testing . . . 46
7. Dev Ops . . . 46
7.1. Build Automation . . . 46
7.2. Continuous Integration . . . 46
7.3. Coverage Reporting . . . 46
7.4. Documentation Previews . . . 46
7.5. Making a Release . . . 46
7.6. Managing Dependencies . . . 47
Appendix A: Product Scope . . . 47
Appendix B: User Stories . . . 47

Appendix C: Use Cases 48
Appendix D: Non-Functional Requirements 50
Appendix E: Glossary 51
Appendix F: Instructions for Manual Testing 51

By: CS2103T W11-3 Team Since: Feb 2019 Licence: MIT

1. Introduction

Welcome to **TravelBuddy**'s Developer Guide, put together by Chung Ming, Prem, Niven and Shaun. In this guide, you will find all the relevant information to get you familiarize with TravelBuddy's design and implementation.

2. Setting Up

Before we begin, here is a quick guide on how you can set up your machine to optimize the performance of TravelBuddy.

2.1. Prerequisites

There are two installations that you need to take note of. They are:

- **Install JDK version 9 or later**

WARNING	For Windows developers, Java version 10 will fail to run tests in headless mode due to a JavaFX bug. Therefore, we strongly encourage all Windows developers to stick with JDK version 9.

- **Install IntelliJ IDE**

NOTE	By default, IntelliJ already has both Gradle and JavaFX plugins installed. As TravelBuddy will be making use of them, please do not disable them. If you have disabled them, go to File > Settings > Plugins to re-enable them.

2.2. Setting Up the Project in Your Computer

For this section, we will integrate JDK, IntelliJ and GitHub with TravelBuddy, before compiling it using Gradle. The steps are listed below:

1. Go to this repo, fork it first before cloning it. Fork it again to your computer.
2. Open IntelliJ (if you are not in the welcome screen, click File > Close Project to close the existing project dialog first)
3. Set up the correct JDK version for Gradle
    a. Click Configure > Project Defaults > Project Structure
    b. Click New… and find the directory of the JDK
4. Click Import Project
5. Locate the build.gradle file and select it. Click OK
6. Click Open as Project

7. Click `OK` to accept the default settings
8. Open a console and run the command `gradlew processResources` (Mac/Linux: `./gradlew processResources`). It should finish with the `BUILD SUCCESSFUL` message.
   This will generate all resources required by the application and tests.
9. Open `MainWindow.java` and check for any code errors
   a. Due to an ongoing issue with some of the newer versions of IntelliJ, code errors may be detected even if the project can be built and run successfully
   b. To resolve this, place your cursor over any of the code section highlighted in red. Press `ALT`+`ENTER`, and select `Add '--add-modules=…' to module compiler options` for each error
10. Repeat this for the test folder as well (e.g. check `HelpWindowTest.java` for code errors, and if so, resolve it the same way)

2.3. Verifying the Setup

1. Run the `seedu.travel.MainApp` and try a few commands
2. Run the tests to ensure they all pass.

2.4. Configurations to do Before Writing Code

2.4.1. Configuring the Coding Style

This project follows the OSS-Generic Java Coding Standard. IntelliJ's default style is mostly compliant with ours but it uses a different import order from ours. To rectify,

1. Go to `File` > `Settings…` (Windows/Linux), or `IntelliJ IDEA` > `Preferences…` (macOS)
2. Select `Editor` > `Code Style` > `Java`
3. Click on the `Imports` tab to set the order
   - For `Class count to use import with '*'` and `Names count to use static import with '*'`: Set to `999` to prevent IntelliJ from contracting the import statements
   - For `Import Layout`: The order is `import static all other imports`, `import java.*`, `import javax.*`, `import org.*`, `import com.*`, `import all other imports`. Add a `<blank line>` between each `import`

Optionally, you can follow the UsingCheckstyle.adoc document to configure Intellij to check style-compliance as you write code.

2.4.2. Setting Up CI

Set up Travis to perform Continuous Integration (CI) for your fork. See UsingTravis.adoc to learn how to set it up.

After setting up Travis, you can optionally set up coverage reporting for your team fork (see UsingCoveralls.adoc).

NOTE	Coverage reporting could be useful for a team repository that hosts the final version but it is not that useful for your personal fork.

Optionally, you can set up AppVeyor as a second CI (see UsingAppVeyor.adoc).

NOTE	Having both Travis and AppVeyor ensures your App works on both Unix-based platforms and Windows-based platforms (Travis is Unix-based and AppVeyor is Windows-based)

2.4.3. Getting started with coding

When you are ready to start coding, get some sense of the overall design by reading Section 3.1, "Architecture".

3. Design

This section provides a high-level overview of overall architecture of TravelBuddy, including the design and structure of components and their constituent classes.

3.1. Architecture

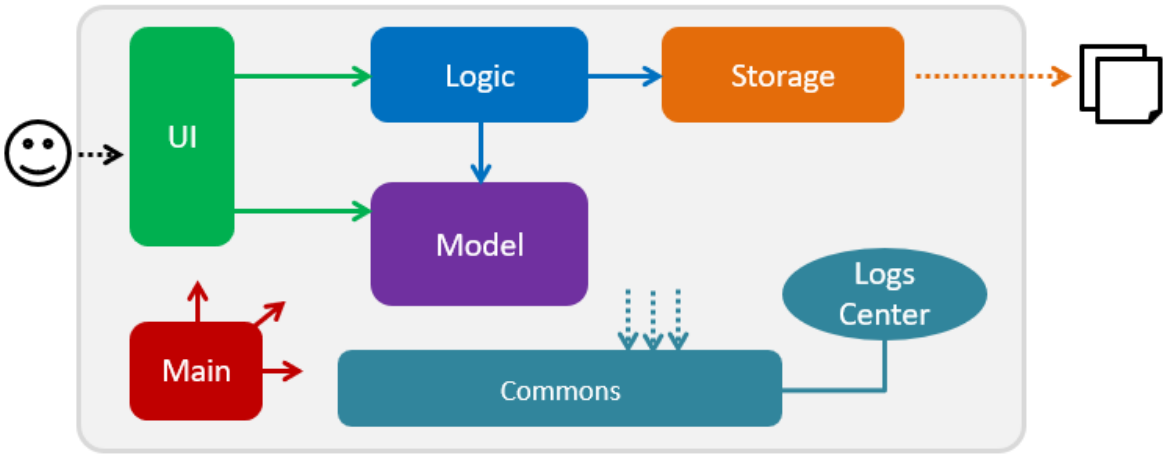


Figure 3.1.1: Architecture Diagram

Figure 3.1.1 above explains the high-level design of TravelBuddy. Given below is a quick overview of each component.

TIP	The `.pptx` files used to create diagrams in this document can be found in the diagrams folder. To update a diagram, modify the diagram in the pptx file, select the objects of the diagram, and choose `Save as picture`.

`Main` has only one class called `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 TravelBuddy's log file.

The rest of TravelBuddy consists of four components.

- `UI`: The User Interface (UI) of the App.
- `Logic`: The command executor.
- `Model`: Holds the data of TravelBuddy in-memory.
- `Storage`: Reads data from, and writes data to, the hard disk.

Each of the four components

- Defines its Application Programming Interface (*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, as seen in the class diagram in Figure 3.1.2, defines it's API in the `Logic.java` interface and exposes its functionality using the `LogicManager.java` class.

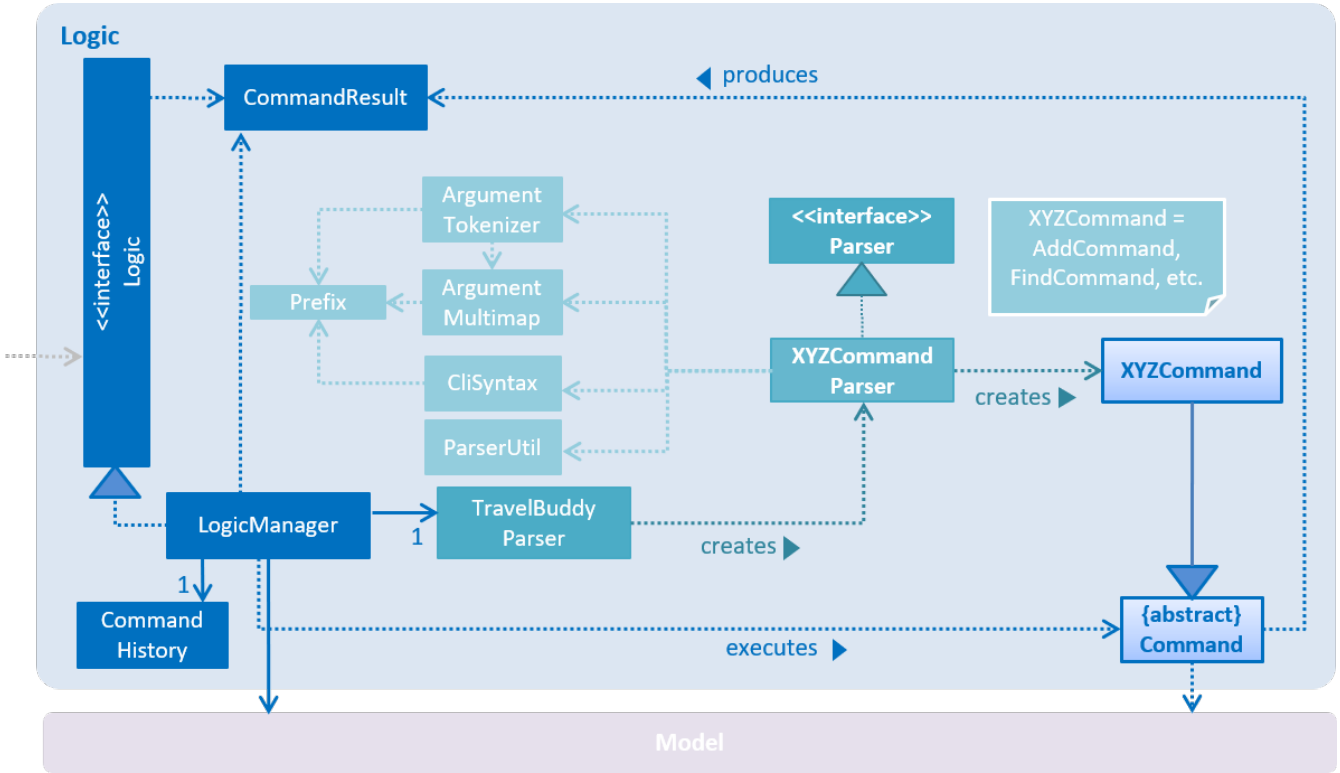


Figure 3.1.2: Class Diagram of the Logic component

How the architecture components interact with each other

The sequence diagram, as seen in Figure 3.1.3, below shows how the components interact with each other for the scenario where the user issues the command `delete 1`.

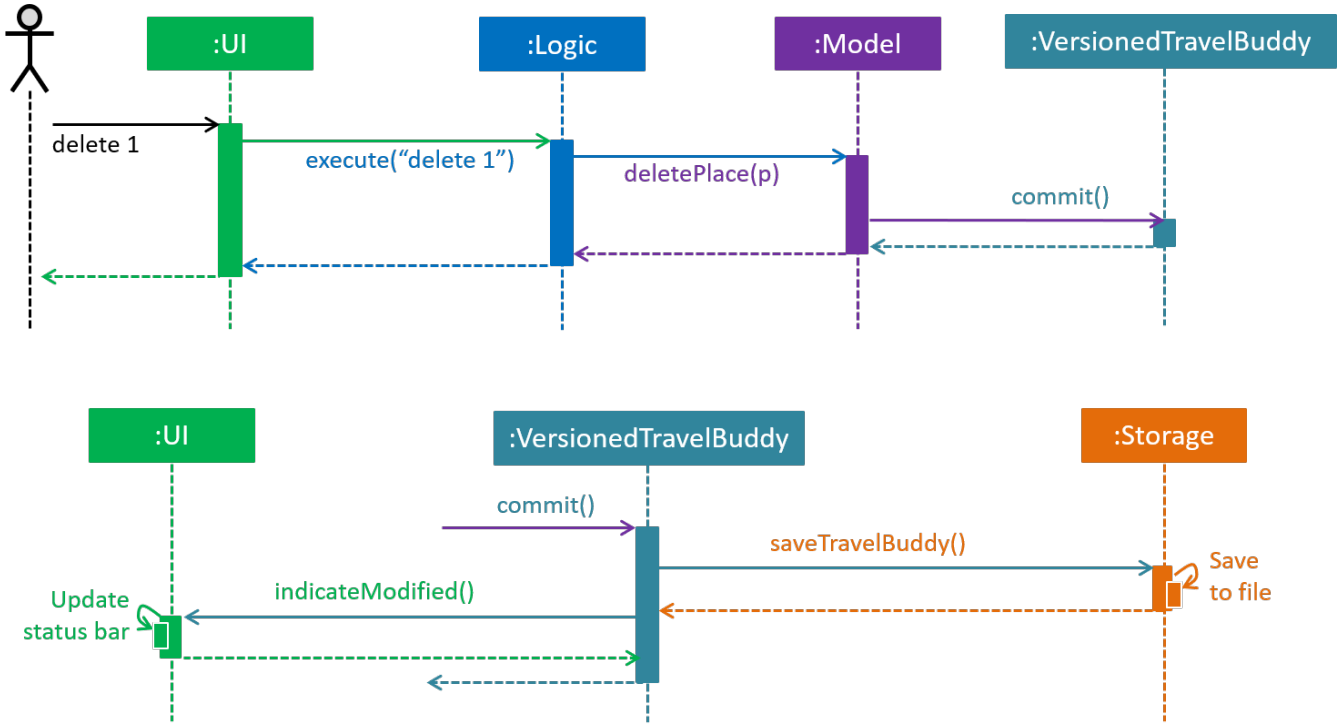


Figure 3.1.3: component interactions for `delete 1` command

The sections below give more details of each component.

3.2. UI Component

API : `Ui.java`

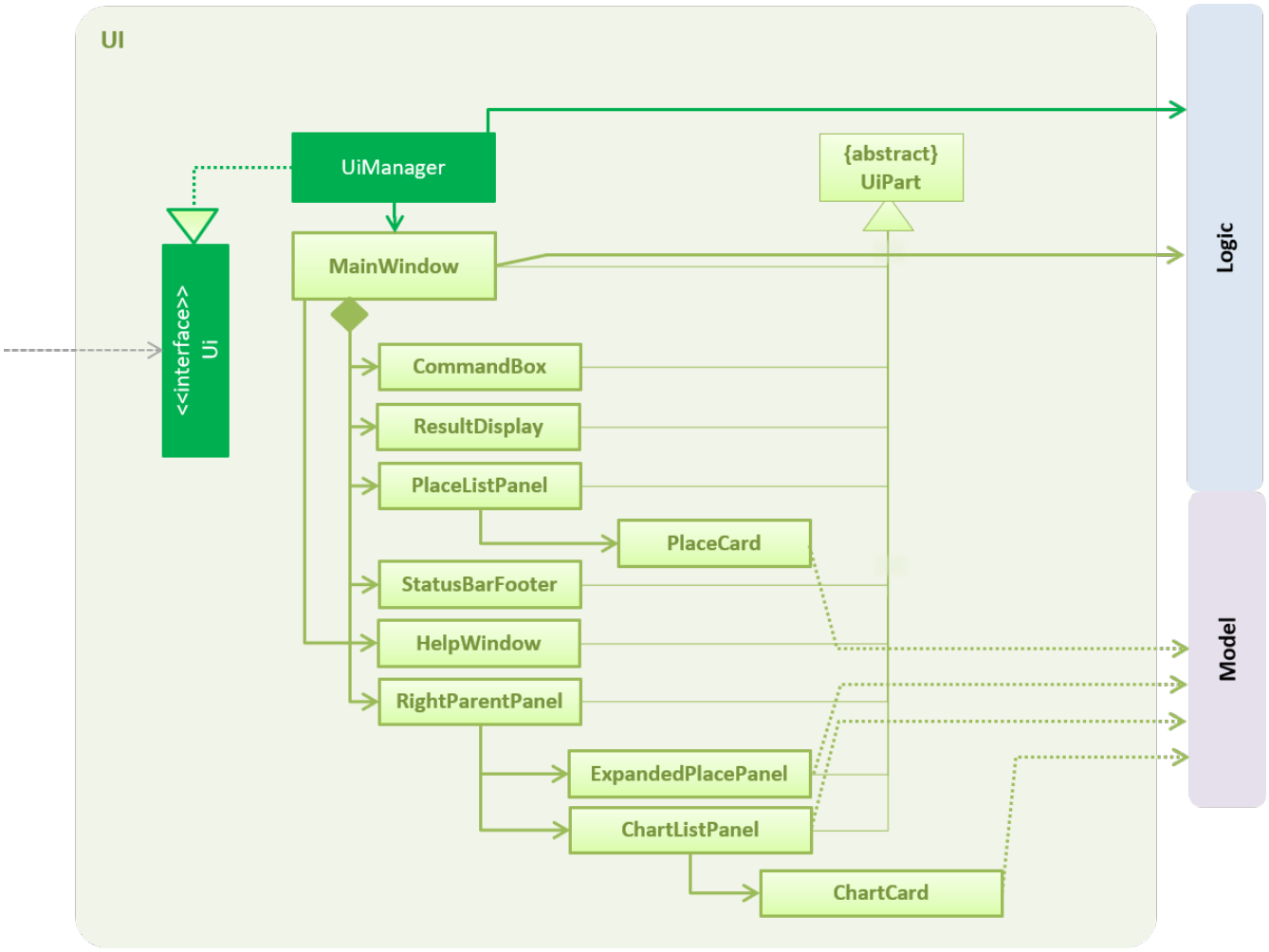


Figure 3.2.1: Structure of the UI component

The `UI` component, as seen in Figure 3.2.1, uses the 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 **controllers** are found in the `src/main/java/seedu/travel/ui` folder. Each class corresponds to a specific `.fxml` file.

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.

The UI consists of a `MainWindow` that is made up of various parts specified below. All parts inherit from the abstract `UiPart` class.

Brief overview of each UI component:

`MainWindow`: Consists of the main menu with `File` and `Help` dropdowns and the remaining UI parts below.

- `HelpWindow`: Webview to display UserGuide.html
- `CommandBox`: Directly below the main menu, CLI displayed here

- `ResultDisplay`: Displays CLI feedback to user, (eg. `Unknown command`, `Invalid command format!`)
- `StatusBarFooter`: At the bottom of MainWindow, reports the status of the TravelBudy application
- `PlaceListPanel`: On the left of the application, displays Places as a VBox of `PlaceListCards`
    - `PlaceListCard`: Labels for the Display index and Place data fields (`CountryCode`, `Rating` etc.) in a VBox the left, Labels that display Name and actual Place data (`CountryCode`, `Rating` etc) in a VBox on the right
- RightParentPanel: Parent panel for both `ChartListPanel` and `ExpandedPlacePanel`, contains helper methods for switching functionality, interacts with `Model` and `Logic`
    - `ChartListPanel`: Parent panel for analytics of all Places, the `generate` command displays this panel
        - `ChartCard`: Displays `Chart` data in a bar graph
    - `ExpandedPlacePanel`: Displays an expanded view of all Place data, including the `Photo`
- `BrowserPanel`: ***Deprecated.*** Opens on clicking the PlaceListCard, on the right of the PlaceListPanel

*All CSS code can be found in `LightTheme.css` and `Extensions.css`

3.3. Logic Component

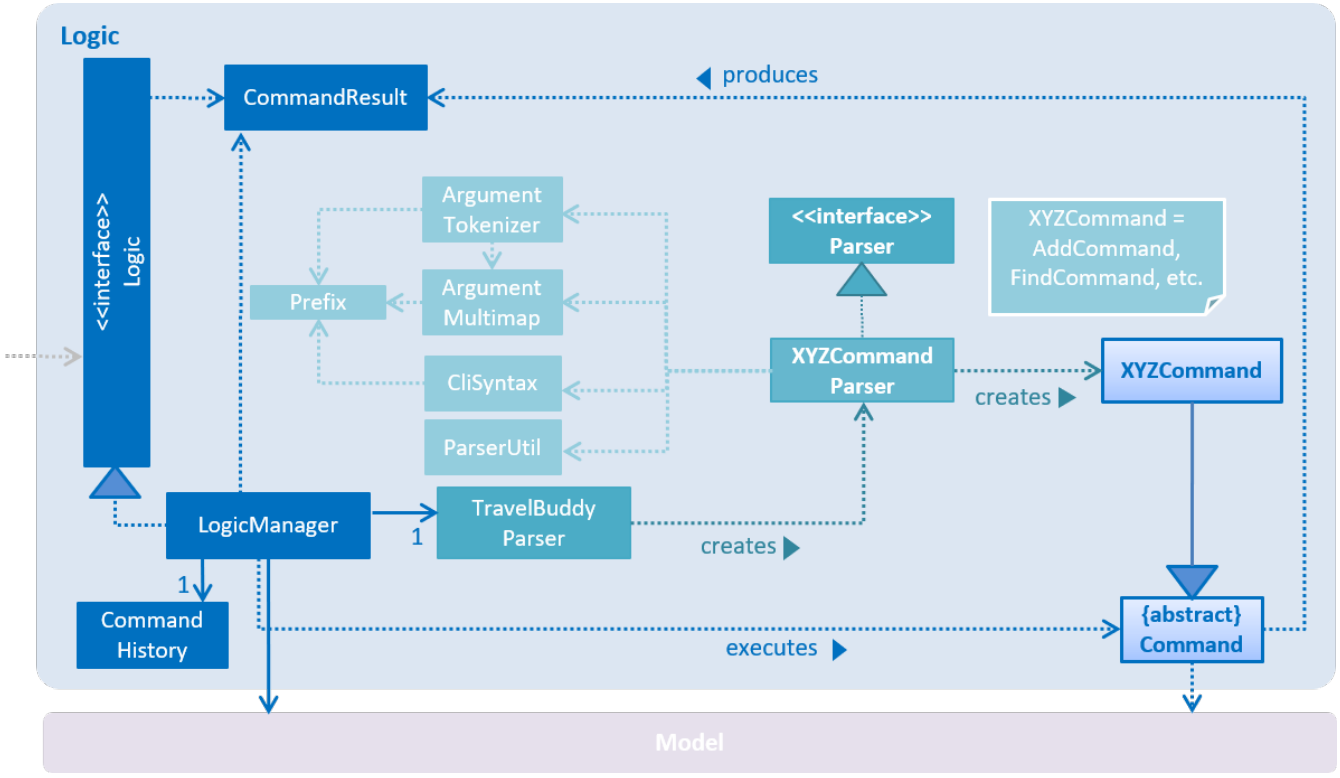


Figure 3.3.1: Structure of the Logic component

API : `Logic.java`

Below is an overview of the `Logic` interface:

1. `Logic` uses the `TravelBuddyParser` 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 place).
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 in Figure 3.3.2 below is the sequence diagram for interactions within the `Logic` component for the `execute("delete 1")` API call.

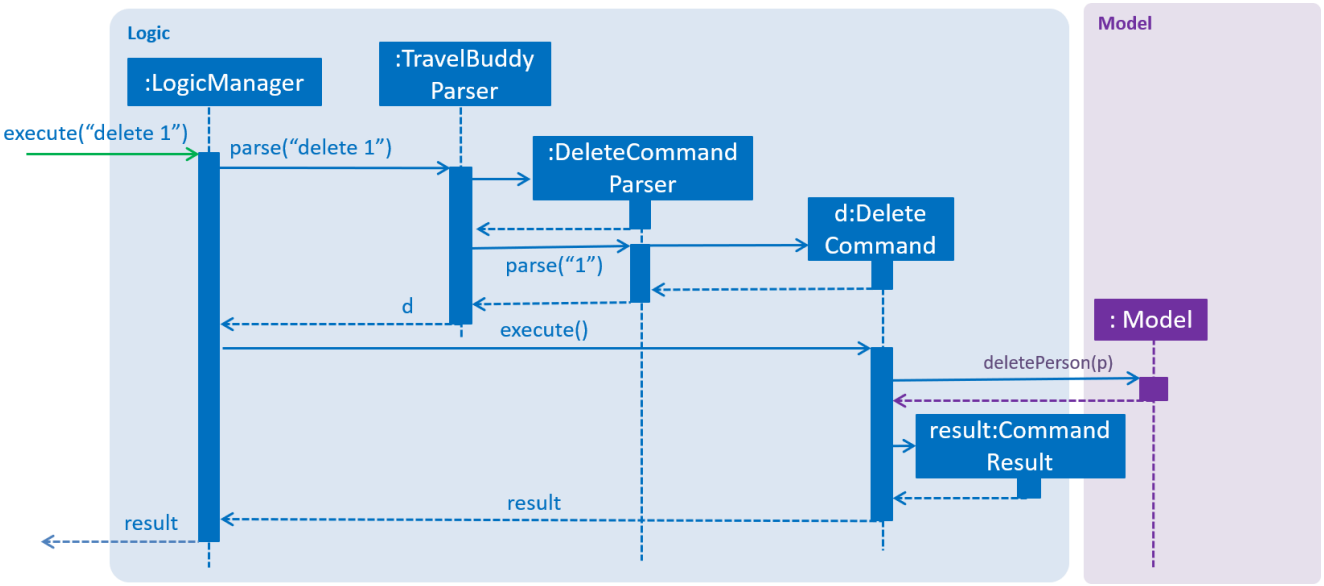


Figure 3.3.2: Interactions inside the Logic component for the `delete 1` command

3.4. Model Component

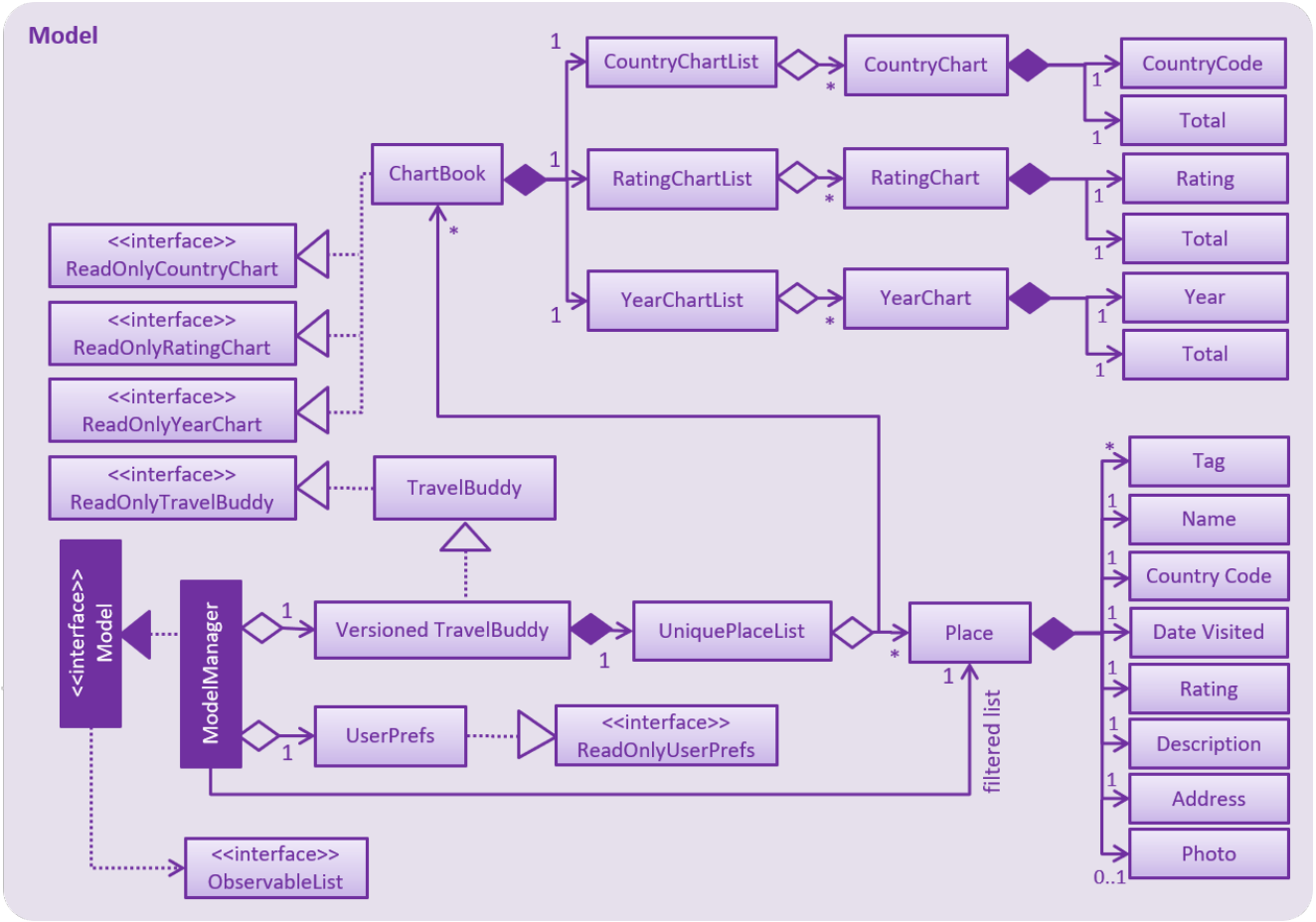


Figure 3.4.1: Structure of the Model component

API : `Model.java`

Below is an overview of the `Model` interface:

- `Model` stores a `UserPref` object that represents the user's preferences.
- `Model` stores the place data.
- `Model` exposes an unmodifiable `ObservableList<Place>` 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.
- `Model` does not depend on any of the other three components.

NOTE

As a more Object Oriented Programming (OOP) model, we can store a `Tag` list in `TravelBuddy`, which `Place` can reference. This would allow `TravelBuddy` to only require one `Tag` object per unique `Tag`, instead of each `Place` needing their own `Tag` object. An example of how such a model may look like is seen in Figure 3.4.2.

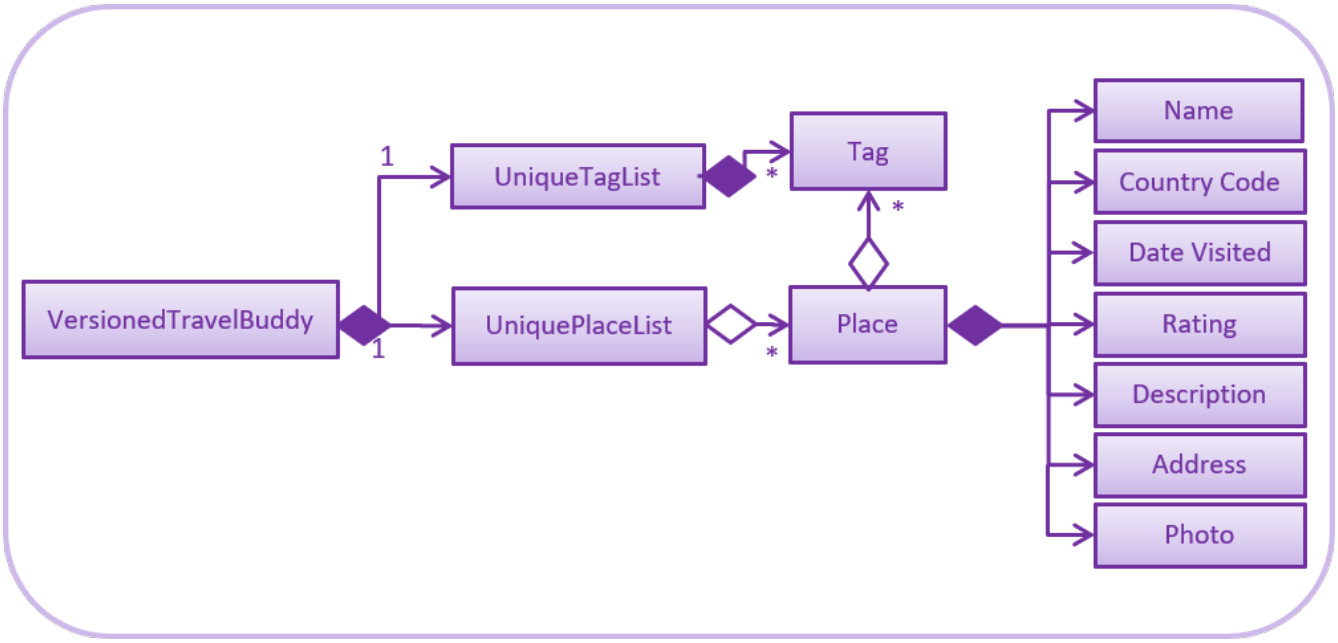


Figure 3.4.2: OOP Class Diagram

3.5. Storage Component

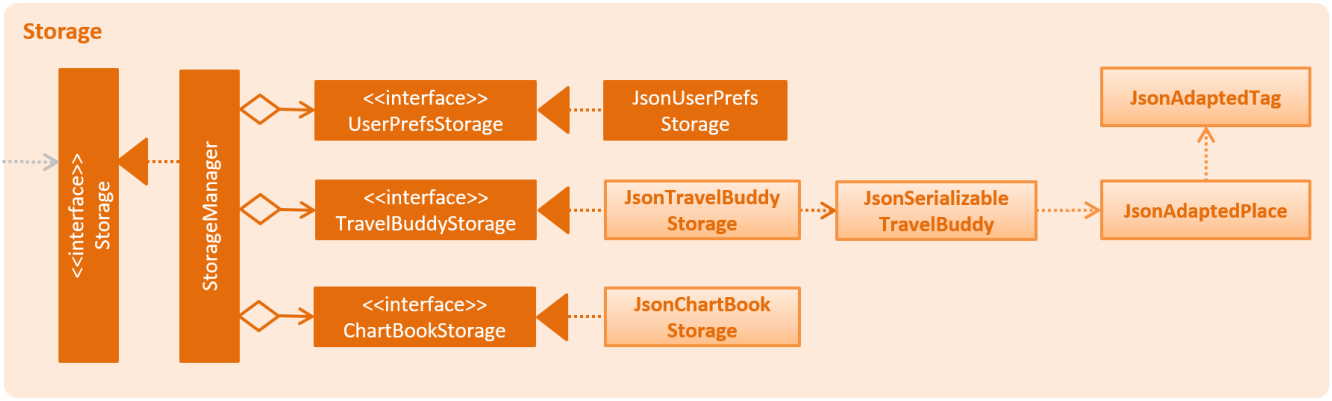


Figure 3.5.1: Structure of the Storage component

API : `Storage.java`

Below is an overview of the `Storage` interface:

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

3.6. Common Classes

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

4. Implementation

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

4.1. Undo/Redo Feature

Called by the `undo` and `redo` commands to undo and redo changes to TravelBuddy Place entries. Undoable commands include `add`, `edit`, `delete`, `deletem`, `clear`, `generate` and `redo`. Redoable commands include `add`, `edit`, `delete`, `deletem`, `clear`, `generate` and `undo`.

4.1.1. Current Implementation

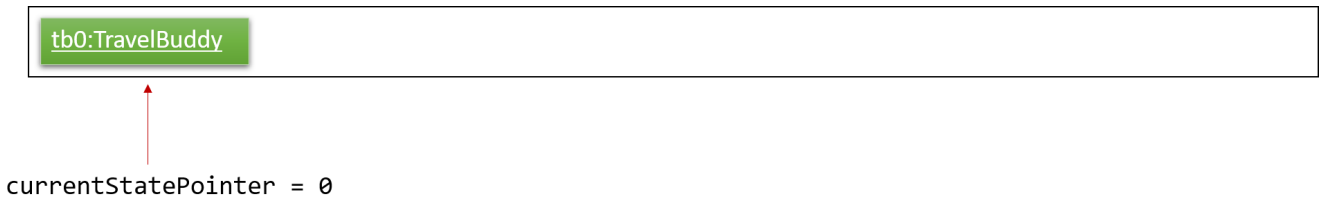
The undo/redo mechanism is facilitated by `VersionedTravelBuddy`. It extends `TravelBuddy` with an undo/redo history, stored internally as a `travelBuddyStateList` and `currentStatePointer`. Additionally, it implements the following operations:

- `VersionedTravelBuddy#commit()` — Saves the current TravelBuddy state in its history.
- `VersionedTravelBuddy#undo()` — Restores the previous TravelBuddy state from its history.
- `VersionedTravelBuddy#redo()` — Restores a previously undone TravelBuddy state from its history.

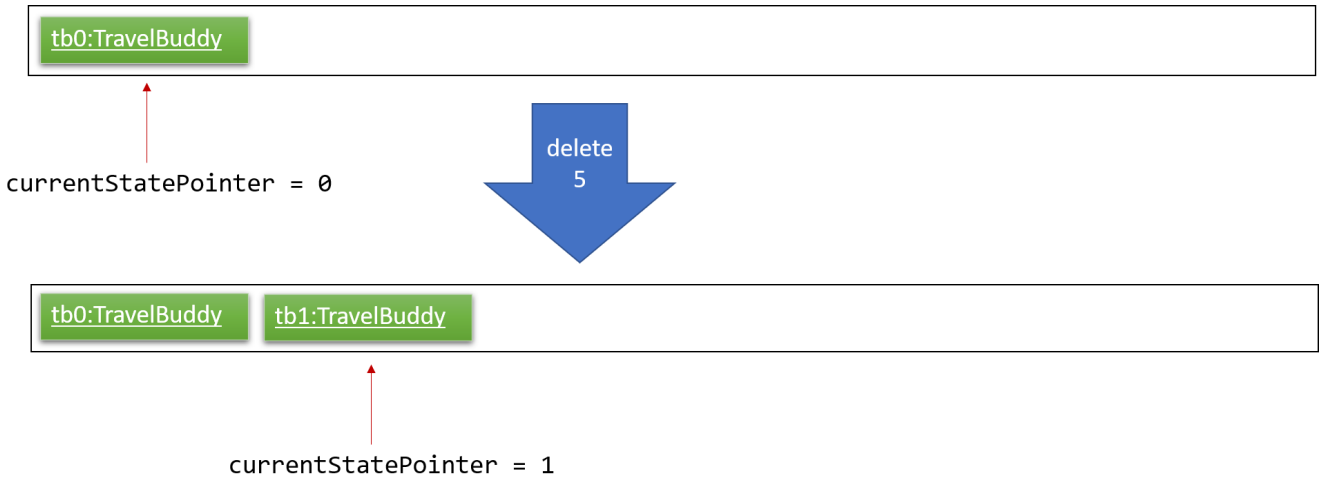
These operations are exposed in the `Model` interface as `Model#commitTravelBuddy()`, `Model#undoTravelBuddy()` and `Model#redoTravelBuddy()` respectively.

Given below is an example usage scenario and how the undo/redo mechanism behaves at each step.

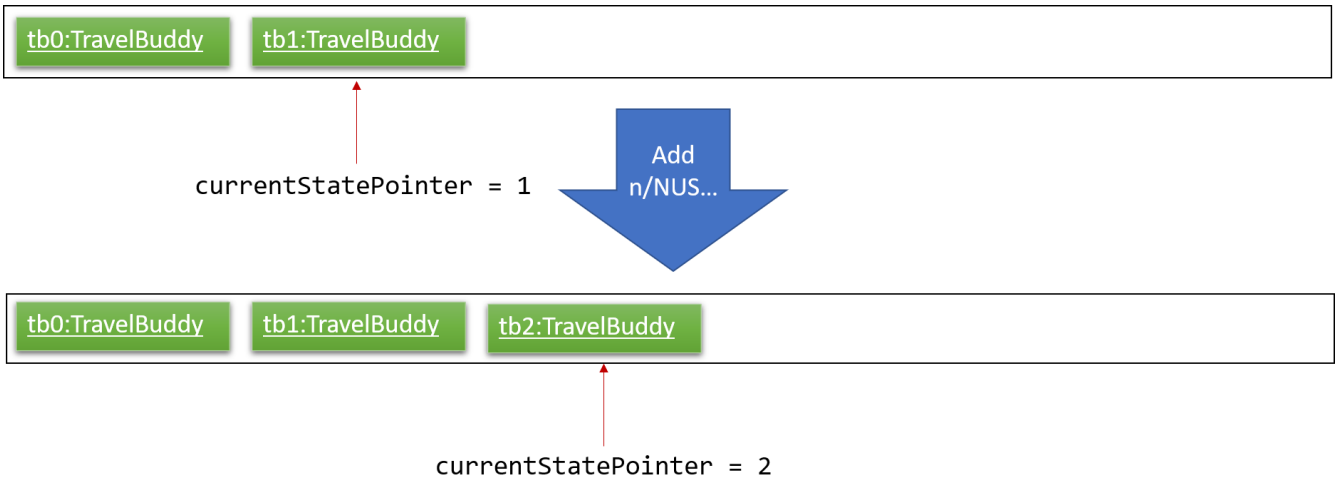
Step 1. The user launches the application for the first time. The `VersionedTravelBuddy` will be initialized with the initial TravelBuddy state, and the `currentStatePointer` pointing to that single TravelBuddy state.



Step 2. The user executes `delete 5` command to delete the 5th place in TravelBuddy. The `delete` command calls `Model#commitTravelBuddy()`, causing the modified state of TravelBuddy after the `delete 5` command executes to be saved in the `travelBuddyStateList`, and the `currentStatePointer` is shifted to the newly inserted TravelBuddy state.

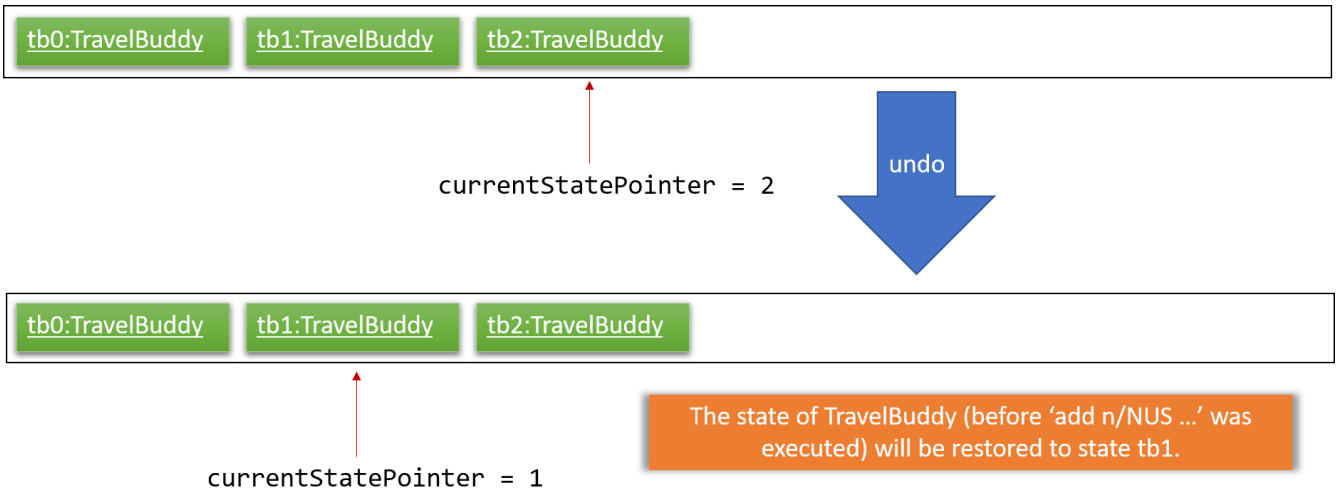


Step 3. The user executes `add n/David …` to add a new place. The `add` command also calls `Model#commitTravelBuddy()`, causing another modified TravelBuddy state to be saved into the `travelBuddyStateList`.



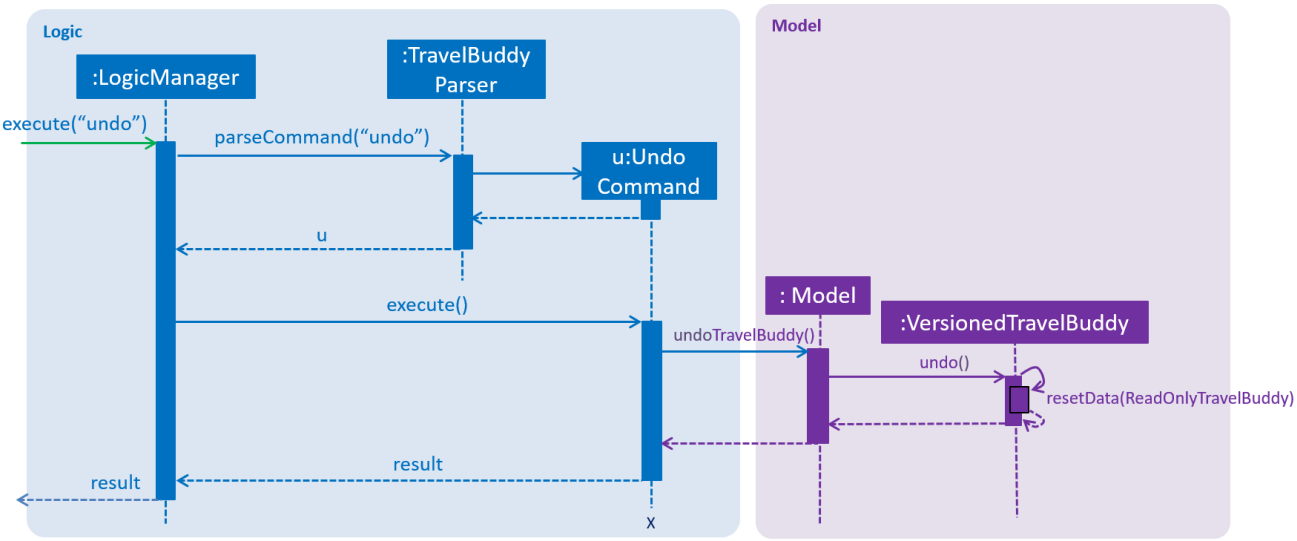
NOTE If a command fails its execution, it will not call `Model#commitTravelBuddy()`, so the TravelBuddy state will not be saved into the `travelBuddyStateList`.

Step 4. The user now decides that adding the place was a mistake, and decides to undo that action by executing the `undo` command. The `undo` command will call `Model#undoTravelBuddy()`, which will shift the `currentStatePointer` once to the left, pointing it to the previous TravelBuddy state. This restores the TravelBuddy to that state.



NOTE If the `currentStatePointer` is at index 0, pointing to the initial TravelBuddy state, then there are no previous TravelBuddy states to restore. The `undo` command uses `Model#canUndoTravelBuddy()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the undo.

The following sequence diagram shows how the undo operation works:

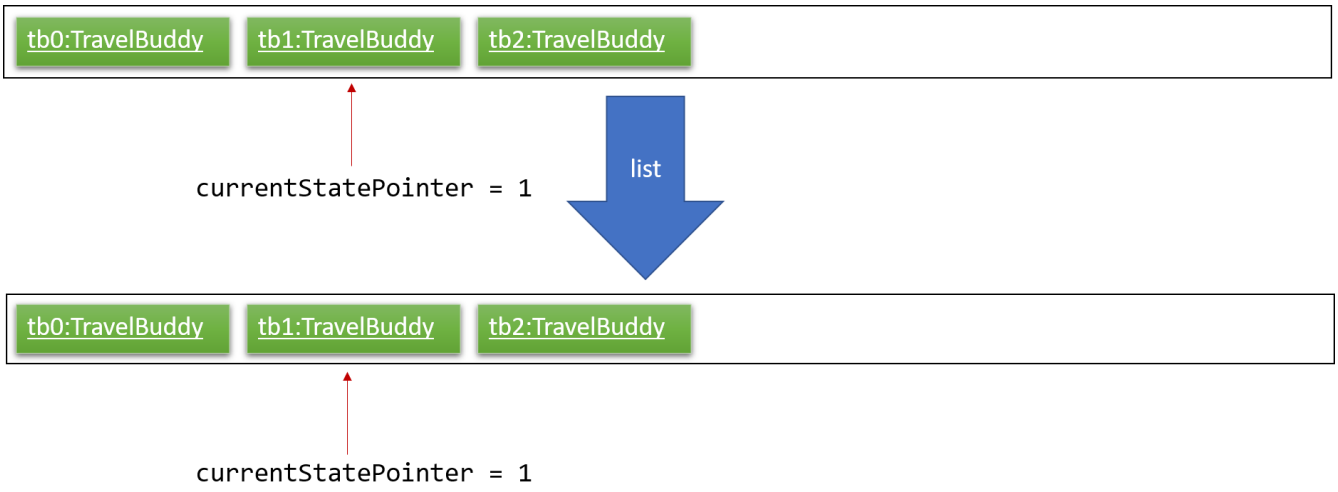


The `redo` command does the opposite — it calls `Model#redoTravelBuddy()`, which shifts the `currentStatePointer` once to the right, pointing to the previously undone state, and restores TravelBuddy to that state.

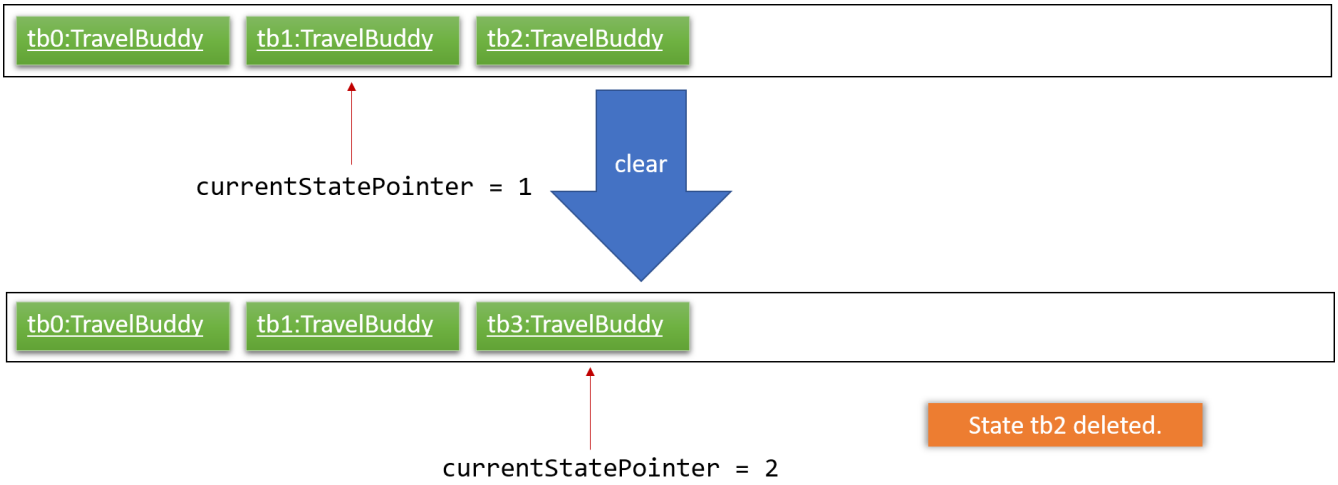
NOTE

If the `currentStatePointer` is at index `travelBuddyStateList.size() - 1`, pointing to the latest TravelBuddy state, then there are no undone TravelBuddy states to restore. The `redo` command uses `Model#canRedoTravelBuddy()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the redo.

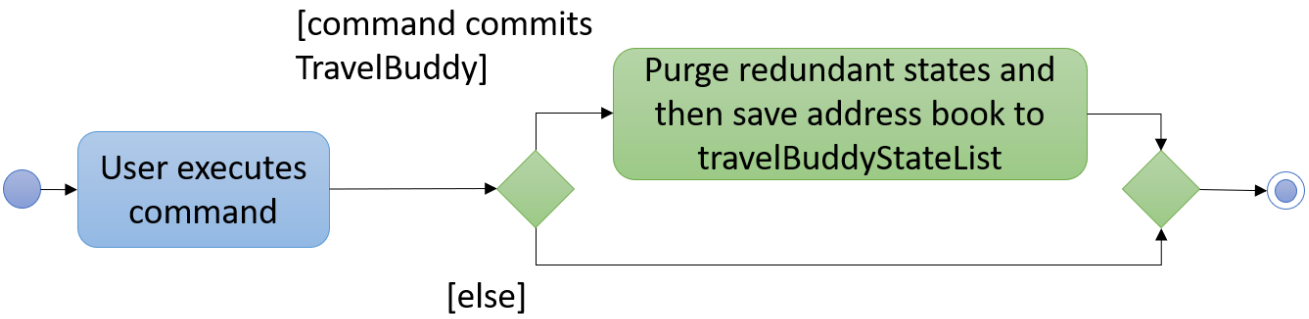
Step 5. The user then decides to execute the command `list`. Commands that do not modify TravelBuddy, such as `list`, will usually not call `Model#commitTravelBuddy()`, `Model#undoTravelBuddy()` or `Model#redoTravelBuddy()`. Thus, the `travelBuddyStateList` remains unchanged.



Step 6. The user executes `clear`, which calls `Model#commitTravelBuddy()`. Since the `currentStatePointer` is not pointing at the end of the `travelBuddyStateList`, all TravelBuddy states after the `currentStatePointer` will be purged. We designed it this way because it no longer makes sense to redo the `add n/David …` command. This is the behavior that most modern desktop applications follow.



The following activity diagram summarizes what happens when a user executes a new command:



4.1.2. Design Considerations

Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Saves the entire TravelBuddy.
  - Pros: Easy to implement.
  - Cons: May have performance issues in terms of memory usage.
- **Alternative 2:** Individual command knows how to undo/redo by itself.
  - Pros: Will use less memory (e.g. for `delete`, just save the place being deleted).
  - Cons: We must ensure that the implementation of each individual command are correct.

Aspect: Data structure to support the undo/redo commands

- **Alternative 1 (current choice):** Use a list to store the history of TravelBuddy states.
  - Pros: Easy for new Computer Science student undergraduates to understand, who are likely to be the new incoming developers of our project.
  - Cons: Logic is duplicated twice. For example, when a new command is executed, we must remember to update both `HistoryManager` and `VersionedTravelBuddy`.
- **Alternative 2:** Use `HistoryManager` for undo/redo
  - Pros: We do not need to maintain a separate list, and just reuse what is already in the codebase.

- Cons: Requires dealing with commands that have already been undone: We must remember to skip these commands. Violates Single Responsibility Principle and Separation of Concerns as `HistoryManager` now needs to do two different things.

4.2. Search Feature

4.2.1. 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 on the GUI, which allows users to retrieve a list of places according to their input.

Logic: The `search` mechanism is executed by `SearchCommand`, which extends from `Command`. A code snippet is shown below:

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

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.

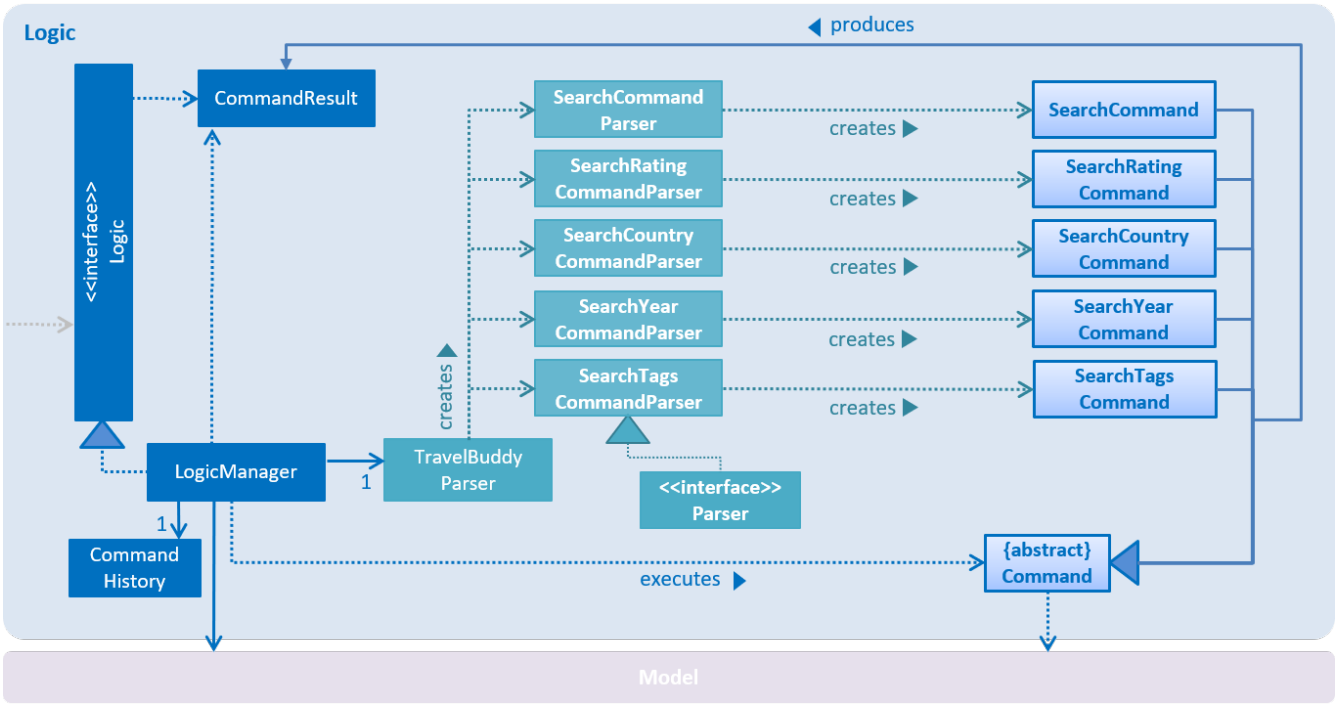


Figure 4.2.1.1: Class Diagram for `search` command.

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

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.

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

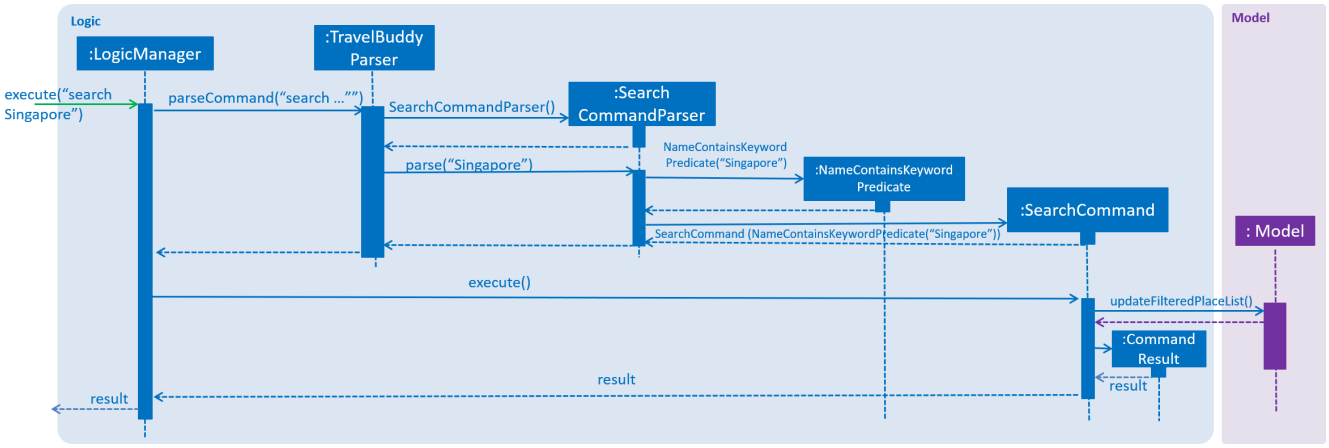


Figure 4.2.1.2: Sequence Diagram for `search` command.

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 based on the `regex` which is `\\s+` 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 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.

Search Name Feature

The command word for search name is `search` and is parsed by `TravelBuddyParser`. The arguments are then passed into `SearchCommandParser`.

The name arguments entered by the user are stored in a list of keywords and passed into `NameContainsKeywordsPredicate`, where 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.

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

Step 1. The user launches the application and sees the GUI with the user's list of places as shown in Figure 4.2.1.3 below.

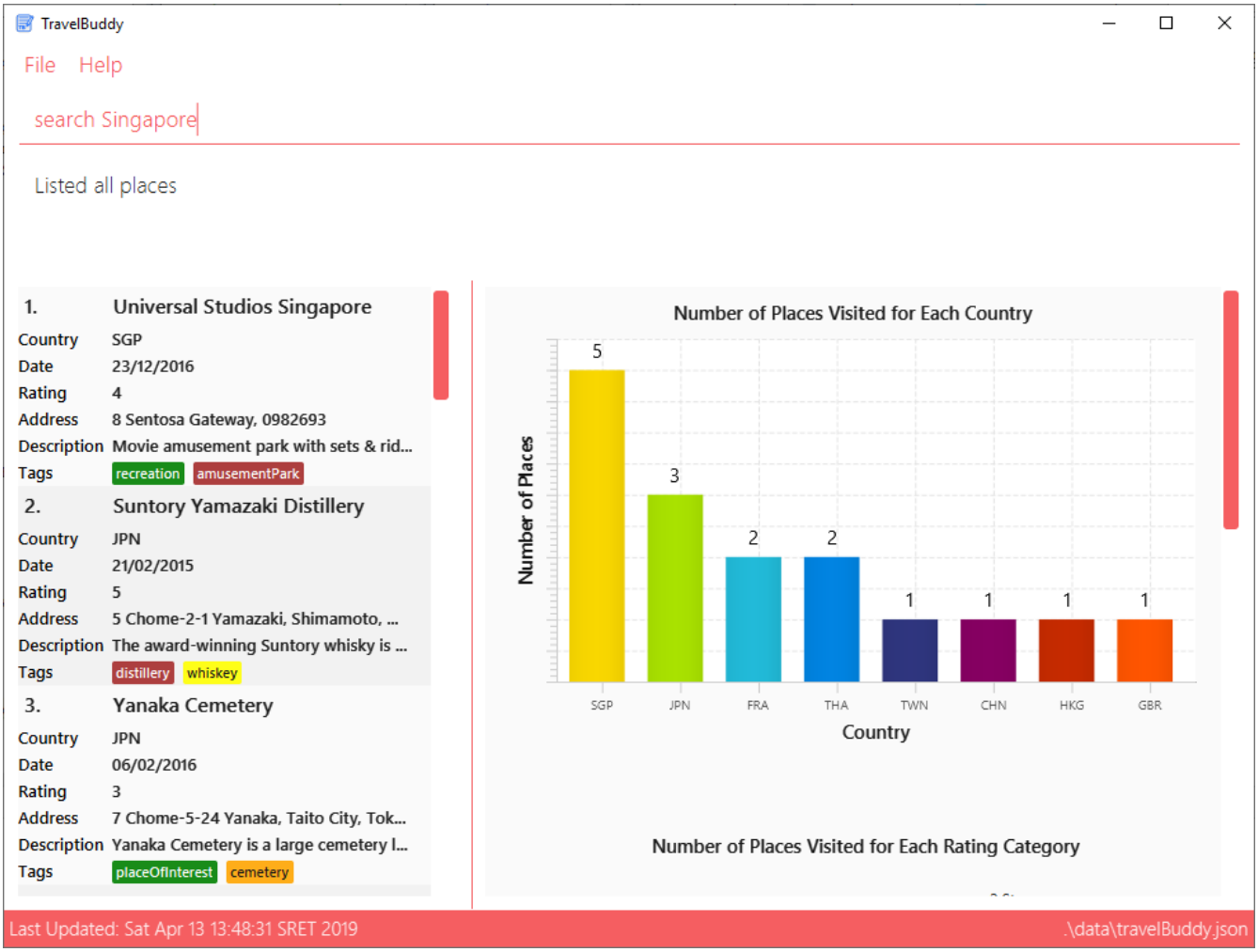


Figure 4.2.1.3: GUI with user's list of places, prior to running `search Singapore` command.

Step 2. The user executes `search Singapore` command to search for all entries in TravelBuddy with `Singapore` in its name. The user input will be passed into `LogicManager#execute()`, which in turn uses `TravelBuddyParser#parseCommand()`. Since the command is `search`, `SearchCommandParser#parse(arguments)` will be called to parse the arguments to be used in `SearchCommand`. The parsed arguments will be compared to every entry in TravelBuddy and matching entries will be displayed.

Step 3. The filtered list is now displayed according to the requirements set by the user input as shown in Figure 4.2.1.4 below.

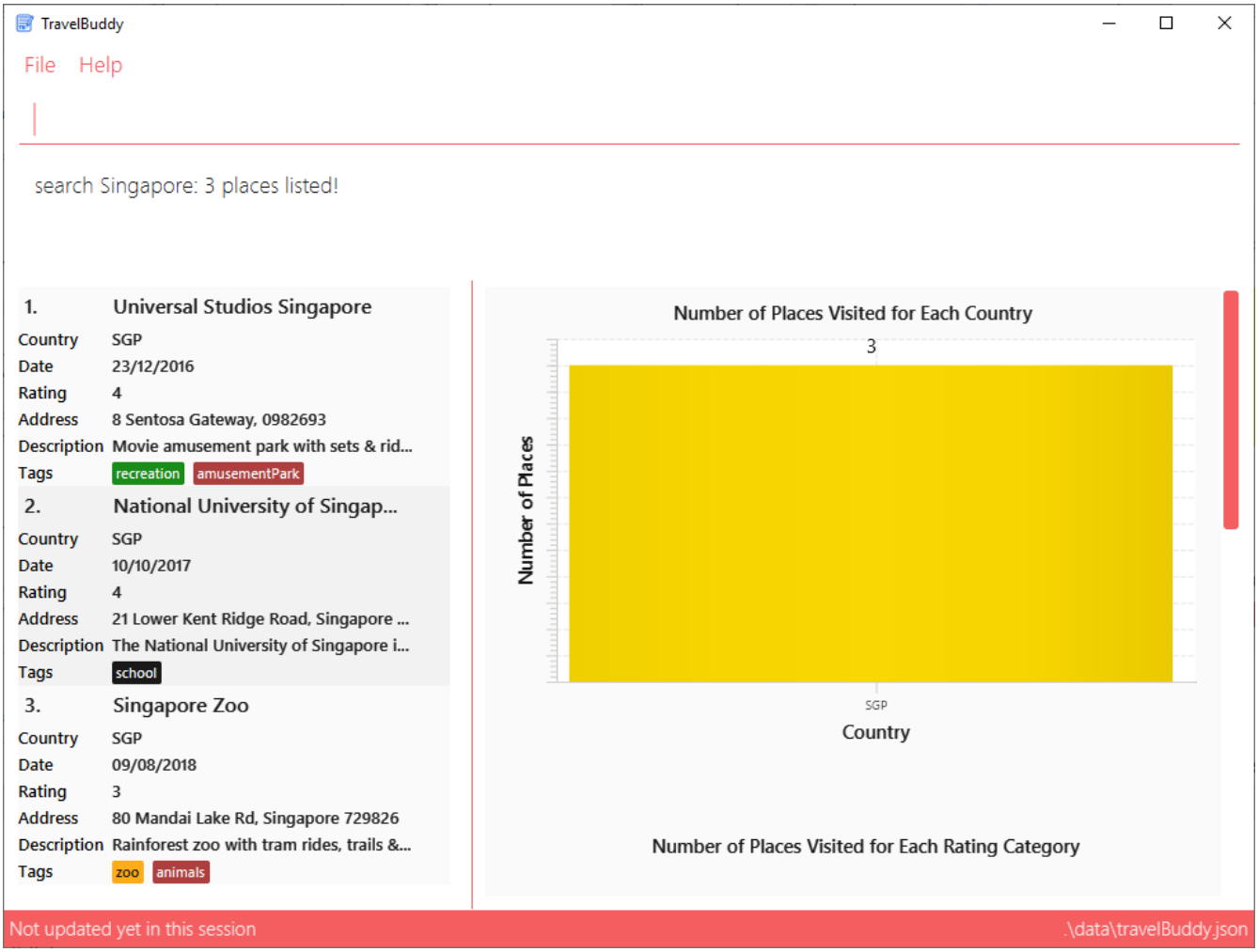


Figure 4.2.1.4: Application Interface displaying the results of `search Singapore` command.

Search Rating Feature

The command word for search rating is `searchr` and is parsed by `TravelBuddyParser`. The arguments are then passed into `SearchRatingCommandParser`.

The rating arguments entered by the user are first checked for validity before being stored in a list of keywords and passed into `RatingContainsKeywordsPredicate`. The list is then 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.

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

Step 1. The user launches the application and sees the GUI with the user's list of places as shown in Figure 4.2.1.5 below.

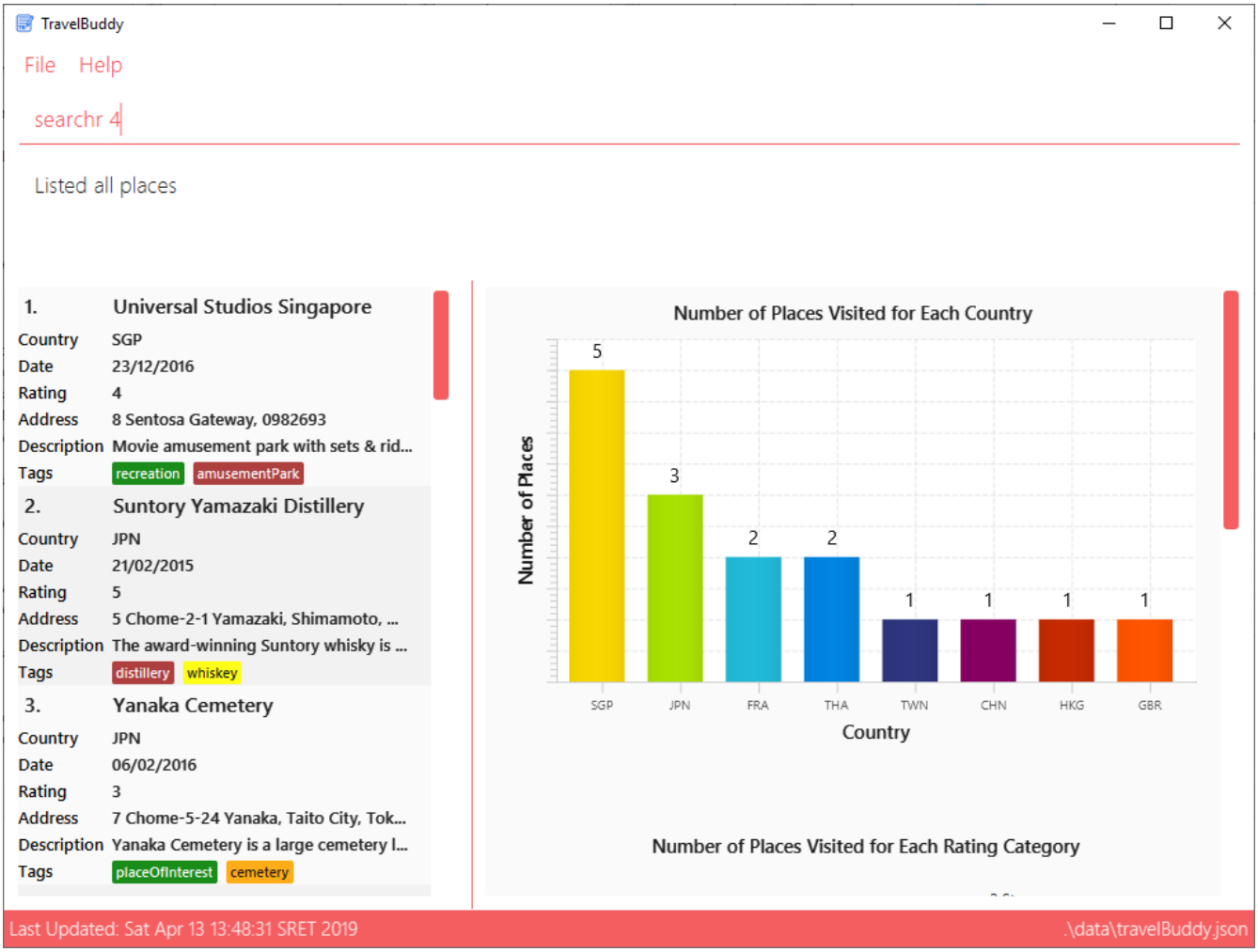


Figure 4.2.1.5: GUI with user's list of places, prior to running `searchr 4` command.

Step 2. The user executes `searchr 4` command to search for all entries in TravelBuddy with `4` as its rating. The user input will be passed into `LogicManager#execute()`, which in turn uses `TravelBuddyParser#parseCommand()`. Since the command is `searchr`, `SearchRatingCommandParser#parse(arguments)` will be called to parse the arguments to be used in `SearchRatingCommand`. The arguments are checked for validity (i.e. rating value between 1 to 5) as seen in the code snippet below before being parsed.

```
String[] ratingKeywords = trimmedArgs.split("\\s+");
for (String rating : ratingKeywords) {
    if (!Rating.isValidRating(rating)) {
        throw new ParseException(String.format(Rating.MESSAGE_CONSTRAINTS,
            SearchRatingCommand.MESSAGE_USAGE));
    }
}
return new SearchRatingCommand(new RatingContainsKeywordsPredicate(Arrays.asList
(ratingKeywords)));
```

The parsed arguments will be compared to every entry in TravelBuddy and matching entries will be displayed.

NOTE	The arguments for `searchr` range from 1 to 5. Non-integer values outside the range are not recognised by the application.

Step 3. The filtered list is now displayed according to the requirements set by the user input as shown in Figure 4.2.1.6 below.

Figure 4.2.1.6: Application Interface displaying the results of `searchr 4` command.

Search Tags Feature

The command word for search tags is `searcht` and is parsed by `TravelBuddyParser`. The arguments are then passed into `SearchTagsCommandParser`.

The tags arguments entered by the user are stored in a list of keywords and passed into `TagsContainsKeywordsPredicate`, where 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.

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

Step 1. The user launches the application and sees the GUI with the user's list of places as shown in Figure 4.2.1.7 below.

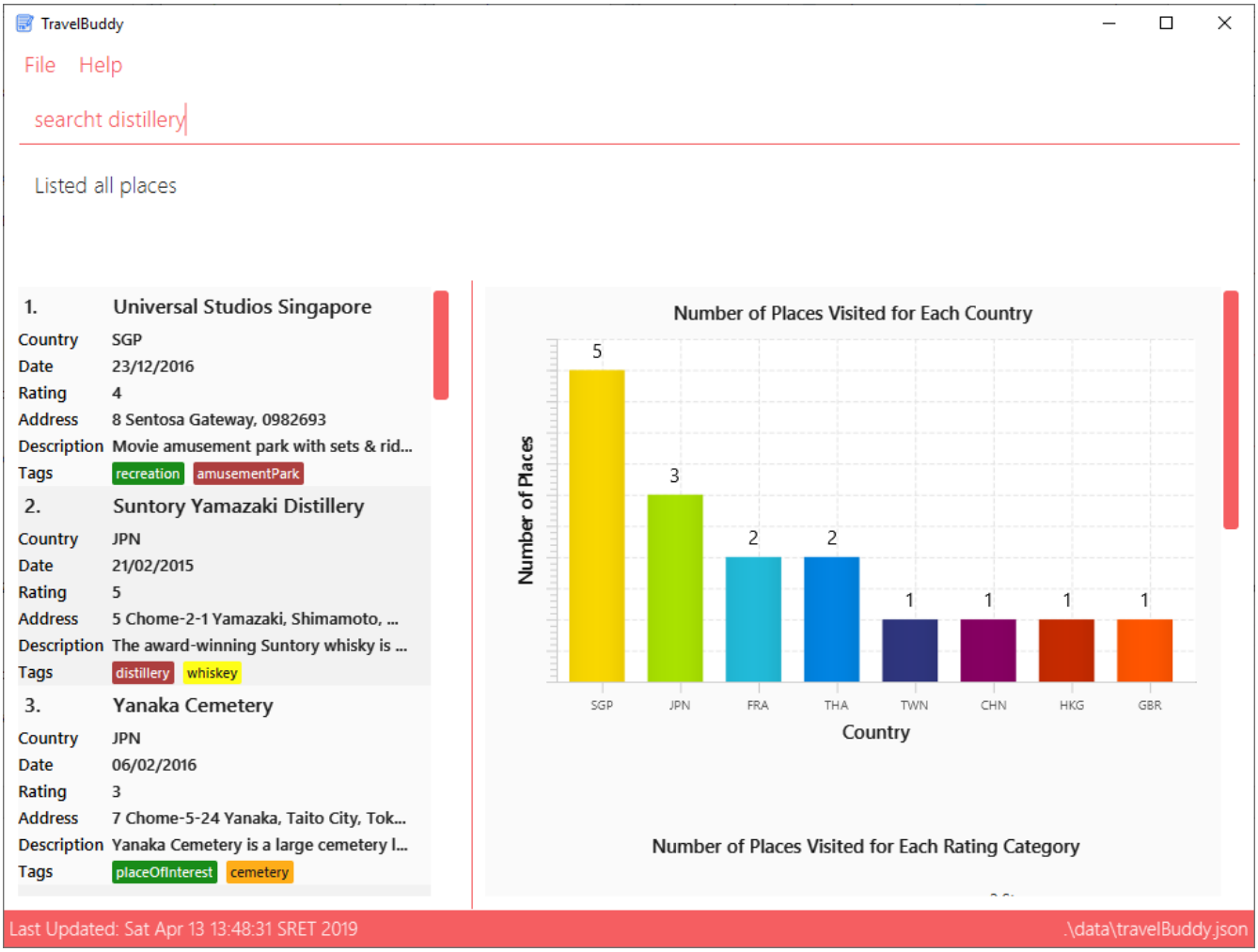


Figure 4.2.1.7: GUI with user's list of places, prior to running `searcht distillery` command.

Step 2. The user executes `searcht distillery` command to search for all entries in TravelBuddy with `distillery` as its tag. The user input will be passed into `LogicManager#execute()`, which in turn uses `TravelBuddyParser#parseCommand()`. Since the command is `searcht`, `SearchTagsCommandParser#parse(arguments)` will be called to parse the arguments to be used in `SearchTagsCommand`. The parsed arguments will be compared to every entry in TravelBuddy and matching entries will be displayed.

Step 3. The filtered list is now displayed according to the requirements set by the user input as shown in Figure 4.2.1.8 below.

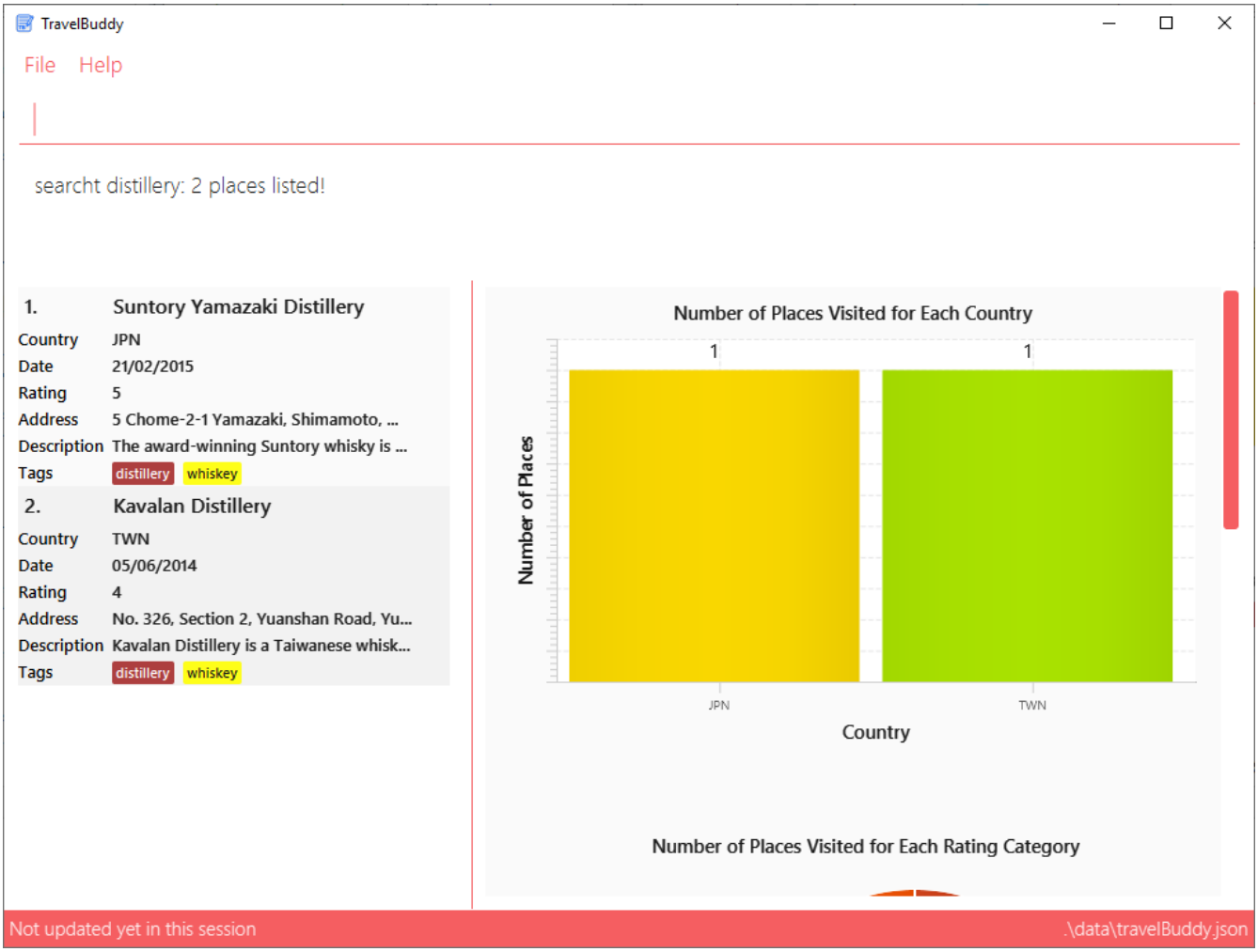


Figure 4.2.1.8: Application Interface displaying the results of `searcht distillery` command.

Search Country Feature

The command word for search country is `searchc` and is parsed by `TravelBuddyParser`. The arguments are then passed into `SearchCountryCommandParser`.

The country code arguments entered by the user are stored in a list of keywords and passed into `CountryCodeContainsKeywordsPredicate`, where the list 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.

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

Step 1. The user launches the application and sees the GUI with the user's list of places as shown in Figure 4.2.1.9 below.

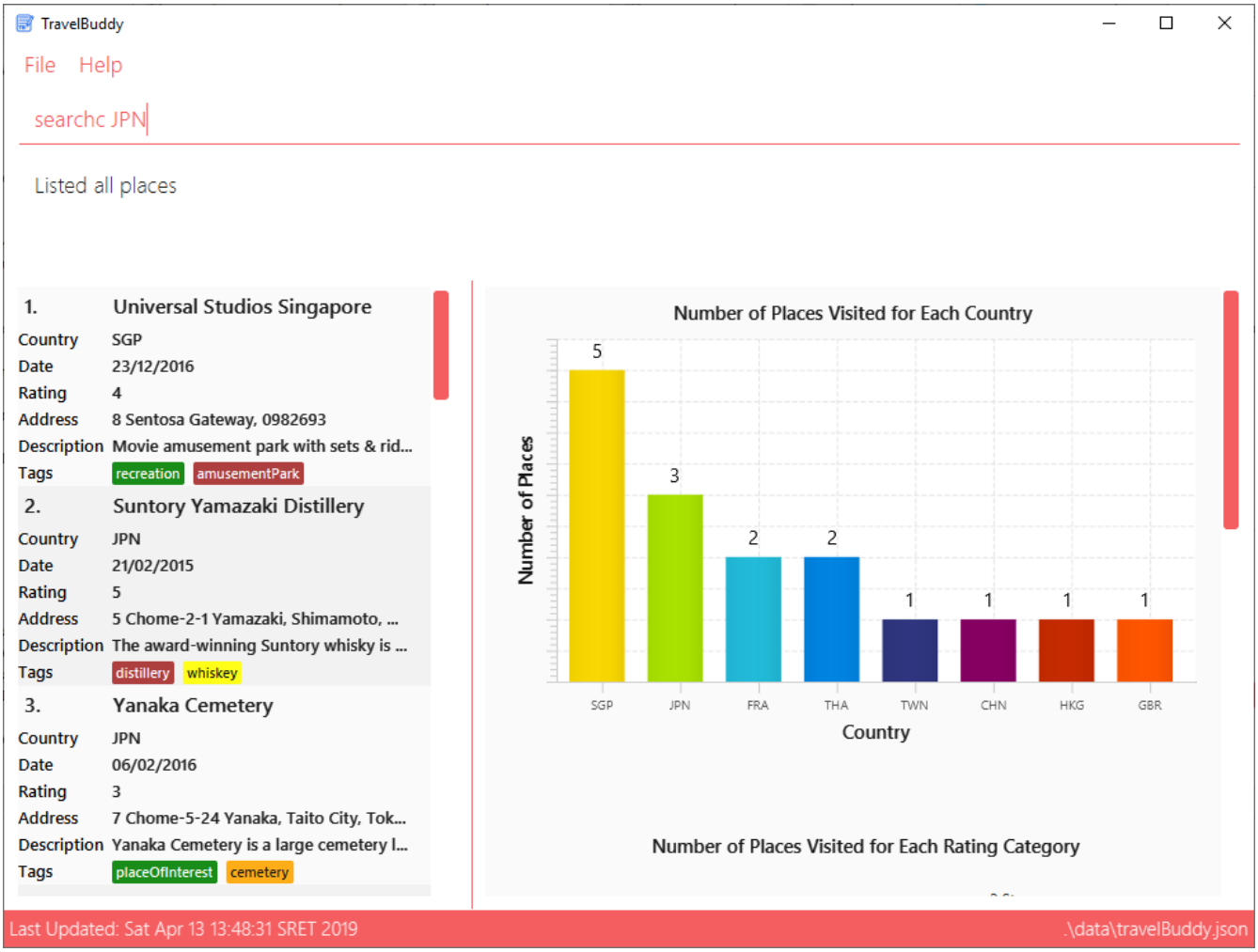


Figure 4.2.1.9: GUI with user's list of places, prior to running `searchc JPN` command.

Step 2. The user executes `searchc SGP JPN` command to search for all entries in TravelBuddy with `SGP` or `JPN` as its country. The `searchc` command will call `LogicManager#execute()`, which in turn uses `TravelBuddyParser#parseCommand()`. Since the command is `searchc`, `SearchCountryCommandParser#parse(arguments)` will be called to parse the arguments to be used in `SearchCountryCommand`. The arguments are then checked for validity (i.e. valid ISO-3166 country code). The parsed arguments will be compared to every entry in TravelBuddy and matching entries will be displayed.

NOTE	The country code arguments for `searchc` must be valid three-letter ISO-3166 country codes.

Step 3. The filtered list is now displayed according to the requirements set by the user input as shown in Figure 4.2.1.10 below.

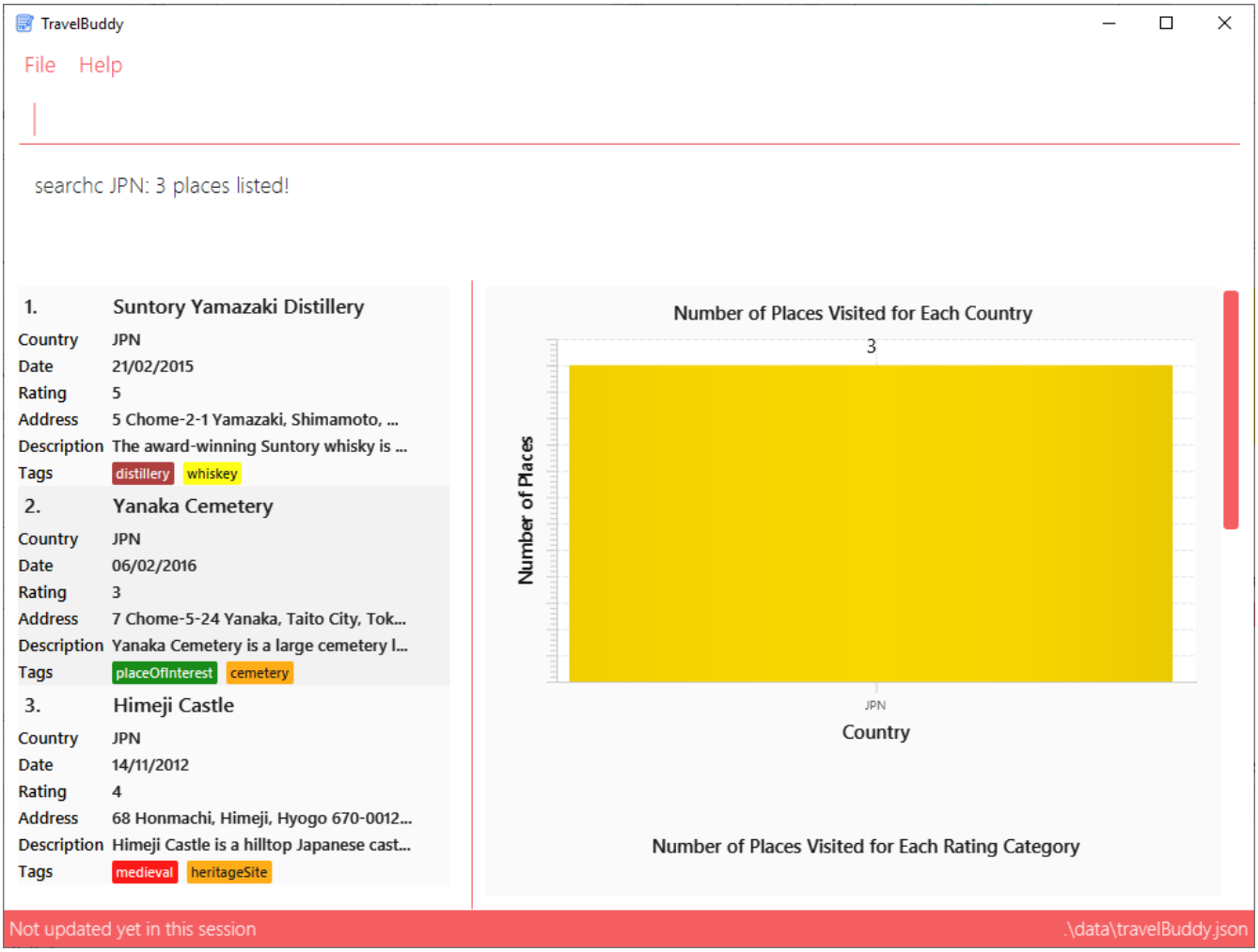


Figure 4.2.1.10: Application Interface displaying the results of `searchc JPN` command.

Search Year Feature

The command word for search year is `searchyear` and is parsed by `TravelBuddyParser`. The arguments are then passed into `SearchYearCommandParser`.

The year arguments entered by the user are stored in a list of keywords and passed into `YearContainsKeywordsPredicate`, where the list 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.

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

Step 1. The user launches the application and sees the GUI with the user's list of places as shown in Figure 4.2.1.11 below.

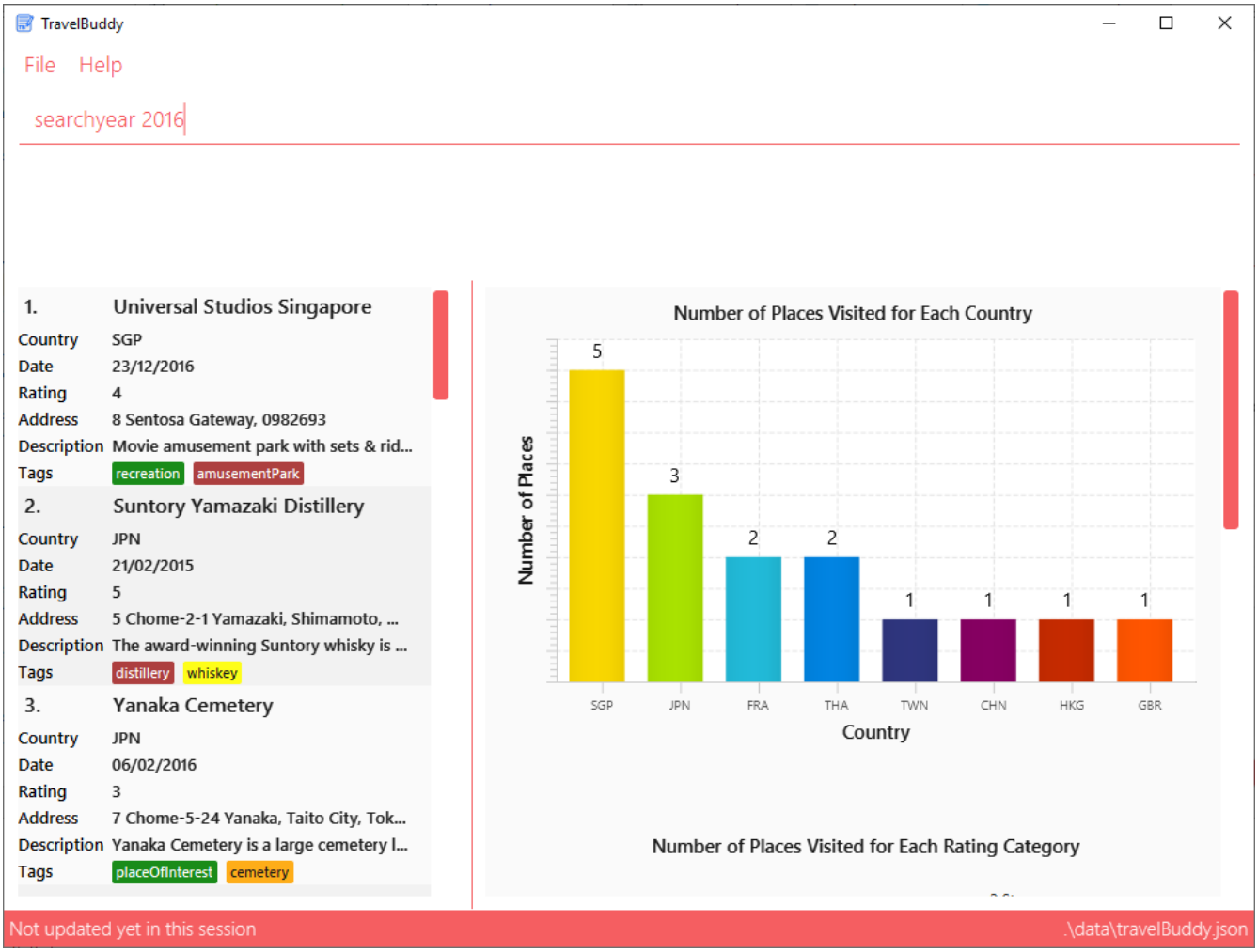


Figure 4.2.1.11: GUI with user's list of places, prior to running `searchyear 2016` command.

Step 2. The user executes `searchyear 2016` command to search for all entries in TravelBuddy with `2016` as its year visited. The `searchyear` command will call `LogicManager#execute()`, which in turn uses `TravelBuddyParser#parseCommand()`. Since the command is `searchyear`, `SearchYearCommandParser#parse(arguments)` will be called to parse the arguments to be used in `SearchYearCommand`. The arguments are then checked for validity (i.e. valid year from 1900 to the current year). The parsed arguments will be compared to every entry in TravelBuddy and matching entries will be displayed.

NOTE	The year arguments range from 1900 to the current year. A single year, multiple years or a range of years can be passed in as arguments.

Step 3. The filtered list is now displayed according to the requirements set by the user input as shown in Figure 4.2.1.12 below.

Figure 4.2.1.12: Application Interface displaying the results of `searchyear 2016` command.

The activity diagram, Figure 4.2.1.13, below summarises what happens when a user inputs a search command:

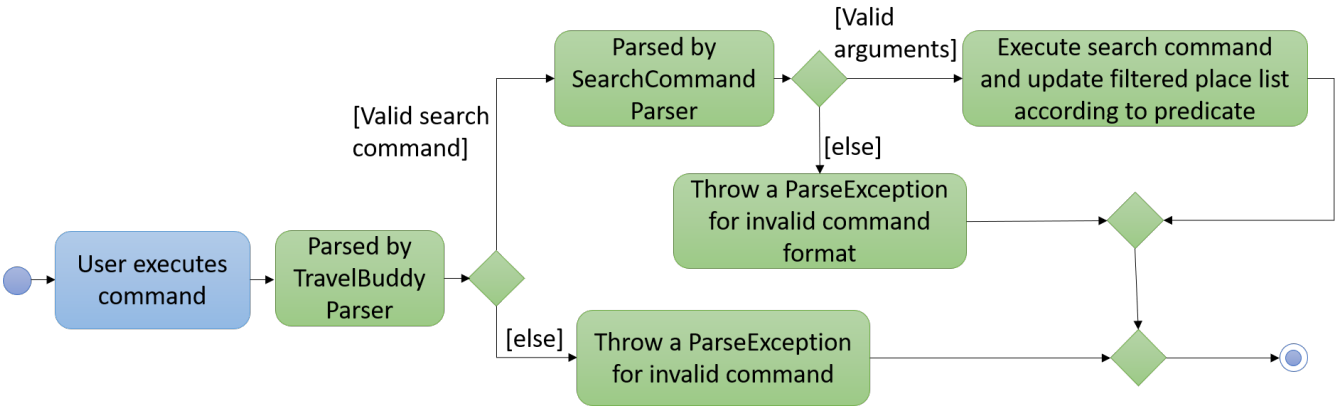


Figure 4.2.1.13: Activity Diagram showing the process flow when a search command is issued.

4.2.2. 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.	Remove whitespaces and 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 provides a wider range of results, which may devolve into a messy clutter if the keyword is too general, defeating 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 are likely to be student undergraduates, so a simple data structure would be more optimal for educational purposes.

4.3. Add Feature

The `add` command is used to add a place into TravelBuddy. The user can add the following details related to the place: name, country code, date visited, rating, address, description, photo (Optional) and Tag (Optional).

NOTE	The country code adheres to the three-letter ISO-3166 standard. The full list of country codes can be found here.

4.3.1. Current Implementation

Figure 4.3.1.1 is a sequence of steps that illustrates the interaction between various classes when the `add` command is entered.

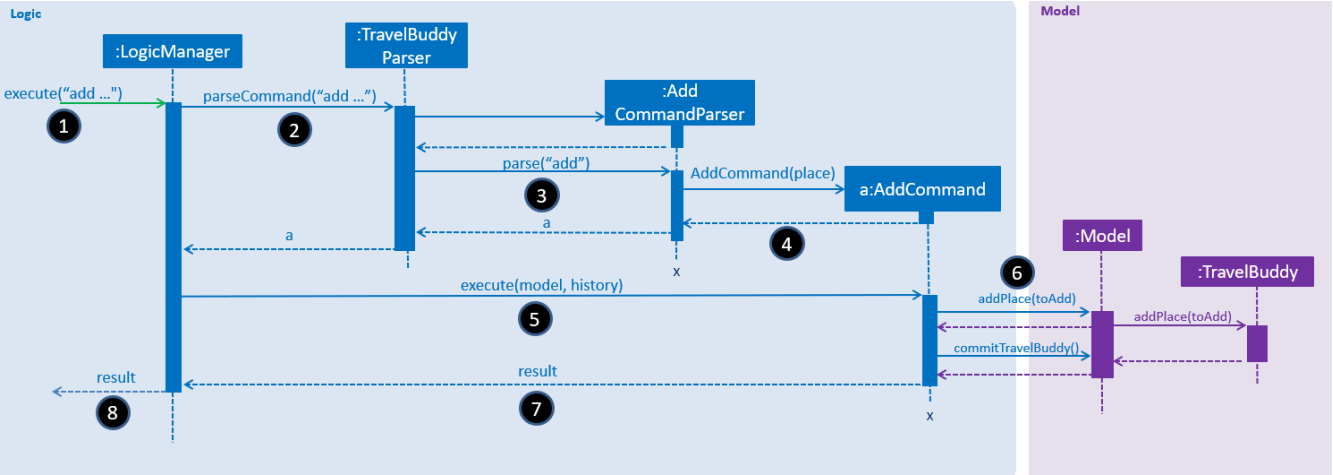


Figure 4.3.1.1: Execution sequence of the `add` command

`add n/NUS Computing cc/SGP dv/10/10/2017 r/3 d/My School a/13 Computing Drive, 117417 t/faculty`

1) The `String` user input is passed into the `LogicManager::execute` method of the LogicManager instance as the only parameter.

2) The `LogicManager::execute` method calls `TravelBuddyParser::parseCommand` which receives the user input as a parameter.

- The user input is formatted: the first `String` token is taken as the command word and the rest of the String is grouped as arguments to be used later by the `AddCommandParser`.
- From the command word, the `TravelBuddyParser` instance identifies the user input as an `add` command and constructs an instance of `AddCommandParser`.

3) TravelBuddyParser calls the `AddCommandParser::parse` method. The `AddCommandParser` takes in the rest of the string, which is `n/NUS Computing cc/SGP dv/10/10/2017 r/3 d/My School a/13 Computing Drive, 117417 t/faculty`

- The string is tokenised to arguments based on their prefixes.

```
ArgumentMultimap argMultimap = ArgumentTokenizer.tokenize(args, PREFIX_NAME,
     PREFIX_COUNTRY_CODE, PREFIX_DATE_VISITED, PREFIX_RATING,
     PREFIX_DESCRIPTION, PREFIX_ADDRESS, PREFIX_PHOTO, PREFIX_TAG);
```

- A check is made on the presence of the relevant prefixes `n/`, `cc/`, `dv/`, `r/`, `d/`, `a/`, `p/` and `t/`.
- When the mandatory prefixes are not present, a `ParseException` will be thrown with an error message on the proper usage of the `add` command.

```
if (!arePrefixesPresent(argMultimap, PREFIX_NAME, PREFIX_COUNTRY_CODE,
    PREFIX_DATE_VISITED, PREFIX_ADDRESS, PREFIX_RATING, PREFIX_DESCRIPTION)
    || !argMultimap.getPreamble().isEmpty()) {
    throw new ParseException(String.format(MESSAGE_INVALID_COMMAND_FORMAT,
        AddCommand.MESSAGE_USAGE));
}
```

- Otherwise, a `Place` object is constructed and used as a field in the creation of a `AddCommand` object.

4) The newly created `AddCommand` object is returned to back to the `LogicManager` instance through the `AddCommandParser` and `TravelBuddyParser` objects.

5) Once the control is returned to the `LogicManager` object, it calls the `AddCommand::execute` method.

- The method takes in a Model object to access the application's data context, the stored data of all places.
- The code snippet below shows the `AddCommand::execute` method.

```
public CommandResult execute(Model model, CommandHistory history)
    throws CommandException {
    requireNonNull(model);
    if (model.hasPlace(toAdd)) {
        throw new CommandException(MESSAGE_DUPLICATE_PLACE);
    }
    model.addPlace(toAdd);
    model.commitTravelBuddy();
    return new CommandResult(String.format(MESSAGE_SUCCESS, toAdd));
}
```

- A check is made on whether the place already exists in TravelBuddy. If it already exists, a `CommandException` will be thrown with an error message on the duplicate entry of the place.

6) The `Place` data is added into TravelBuddy.

- Here, the `Model::addPlace` method is called, and it subsequently calls the `TravelBuddy::addPlace` method.
- Following which, the `Model::commitTravelBuddy` method is called.

7) The `AddCommand::execute` execution completes by returning a new CommandResult that contains a success message to its calling method which is `LogicManager::execute`.

8) Finally, the `CommandResult` is returned to the caller of `LogicManager::execute` and the execution sequence ends.

Add Command

Given below is an example usage scenario and what the user will see in the GUI.

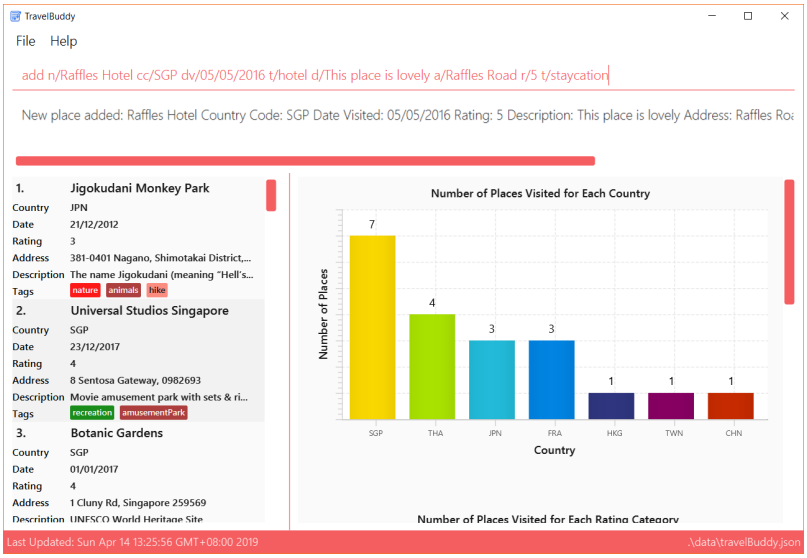


Figure 4.3.1.2: Add command output

The user launches the application and enters the full add command `add n/Raffles Hotel cc/SGP dv/05/05/2016 t/hotel d/This place is lovely a/Raffles Road r/5 t/staycation 117417 t/faculty`. TravelBuddy will start executing the steps mentioned in Figure 4.3.1.1 and the output is shown above in Figure 4.3.1.2.

> **NOTE** The command `add` is in lower-case. Mixed-case or upper-case commands are not recognised by TravelBuddy.

4.3.2. Design Considerations

Aspect: Data structure to store country codes

Given below is a comparison between the alternatives for the data structure that can be used for country codes.

	Alternative 1 (current choice)	Alternative 2
Description	Use `enum` specified in `java.util.Locales`.	Create a data structure containing only the top 30 commonly traveled countries in the world.
Pros	**Ease-of-use**. This approach simply requires the importing of `java.util.Locales` to get the country codes. **Comprehensive**. `java.util.Locales` contains all of the 250 3-letter country codes as specified in ISO-3166.	**Fast**. This approach is computationally less intensive than Alternative 1. This is because there are only 30 country codes in the data structure.

	Alternative 1 (current choice)	Alternative 2
Cons	**Slightly slow**. This approach is computationally more intensive than Alternative 1. This is because there are 250 country codes to search from.	**Tedious**. This approach requires the coder to search for the top 30 visited countries in the world and type out all the 30, 3-letter country codes as specified in ISO-3166. **Not User-friendly**. If the user visited a country that is not in the top 30 list of countries visited, the user would not be able to add it into TravelBuddy.

Decision: Alternative 1 is used for the current implementation. Alternative 2 may not fulfill all of the user requirements of adding any country code. Alternative 1's speed is only slightly slower than Alternative 2.

4.4. Photos Feature

The Photo feature gives users the ability to attach an existing image file to any `Place` object, which is then displayed in the `ExpandedPlacePanel` when the `select` command is called on a particular Place.

4.4.1. Current Implementation

This feature is currently integrated as part of the `add` and `edit` commands. It works by accepting the user-entered absolute file path of the image as an `add` or `edit` command parameter, specified by the prefix `p/`.

Figure 4.4.1.1 below is a sequence of steps, illustrating the interaction between various classes when the add command is used to include a photo with a newly created place.

Step 1: The user enters the command `add n/Himeiji Castle cc/JPN dv/15/12/2017 r/5 d/Wow. a/Kyoto t/castle p/C:\Users\Shaun\Pictures\castle-photo.jpg`.

Step 2 - 5: Identical to `add` command sequence steps 2 - 5, please refer to Section 4.3, "Add Feature" for details

Step 6: The `AddCommandParser` interprets the arguments, and checks the file path input using `ParserUtil.parsePhoto()`.

Step 7: `parsePhoto()` then trims any leading whitespaces. It also trims a single leading and trailing double quotation mark `"`. This is for ease of use with the Windows 10 File Explorer **Copy Path** Home menu function, which returns the filepath of the selected file wrapped with double quotation marks `"`.

Step 8.1: As the `photo` parameter is optional for `add` command, if photo prefix `p/` is not used, `AddCommandParser` calls the 2nd overloaded `Place()` constructor to create a `Place` with a special string `EMPTY_PHOTO_PATH` in the `Photo` attribute.

Step 8.2:

- If photo prefix `p/` is used, he validity of the filepath is then checked with the

`Photo.isValidPhotoFilepath()` method. The method relies on using `ImageIO.read()` to try opening the file using the provided `filepath` string. If the filepath is invalid or the file cannot be opened, then `ImageIO.read()` throws a `IOException`. The exception is caught, control returns to `parsePhoto()` and a `ParseException is thrown`.

```
`    public static boolean isValidPhotoFilepath(String test) {
        if (!(test instanceof String)) {
            throw new NullPointerException();
        }
        try {
            File testPhoto = new File(test);
            Image image = ImageIO.read(testPhoto);
            FileInputStream testStream = new FileInputStream(testPhoto);
            if (image == null) {
                System.out.println("The file " + testPhoto + " could not be opened, it
is not an image");
                return false;
            }
        } catch (IOException e) {
            return false;
        }
        return true;
    }`
```

- Else, control returns to `parseUtil.parsePhoto()`. It uses the all-parameter `Place()` constructor to create a `Place` object, using the `Photo` parameter and other parameters.

NOTE	A valid filepath must be an absolute file path; it starts with the drive-letter (eg. `C:\\`), and ends with the file name and extension (eg. `castle-photo.jpg`). Supported filepaths are `.jpg` `.png` and `.bmp`, which are filepaths supported by `ImageIO.read()` and the Java `Image` class.

Step 9: When the `select` command is executed, the UI component `ExpandedPlacePanel` receives the `Place` object. It then uses the `photo.filepath` attribute to create a JavaFX `Image` object, which is then displayed in a `ImageView` panel at the top of `ExpandedPlacePanel` using the `setImage()` method.

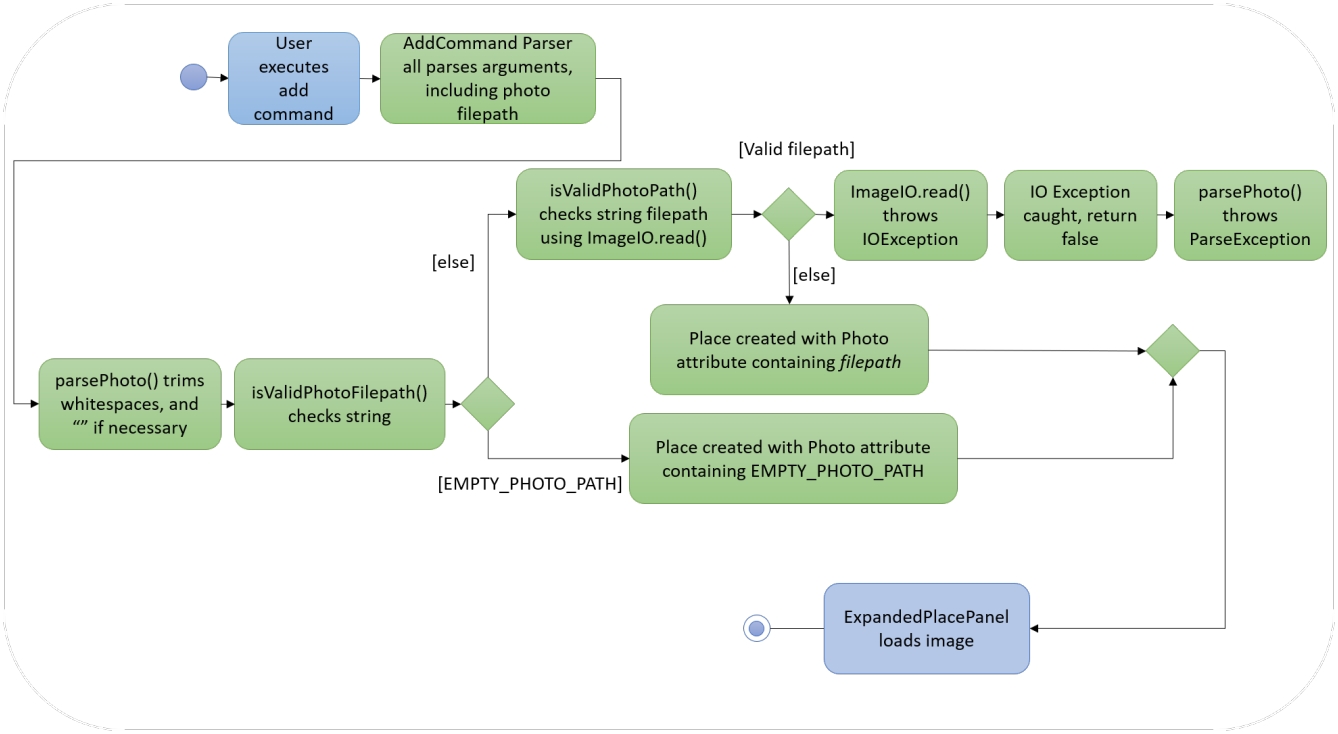


Figure 4.4.1.1: Activity diagram for photo feature

Example: Using `add` to include a photo

Given below is an example usage scenario and what the user will see in the GUI.

Step 1.Input `add n/Himeiji Castle cc/JPN dv/15/12/2017 r/5 d/Wow. a/Kyoto t/castle p/C:\Users\Michael\Pictures\castle-photo.jpg`

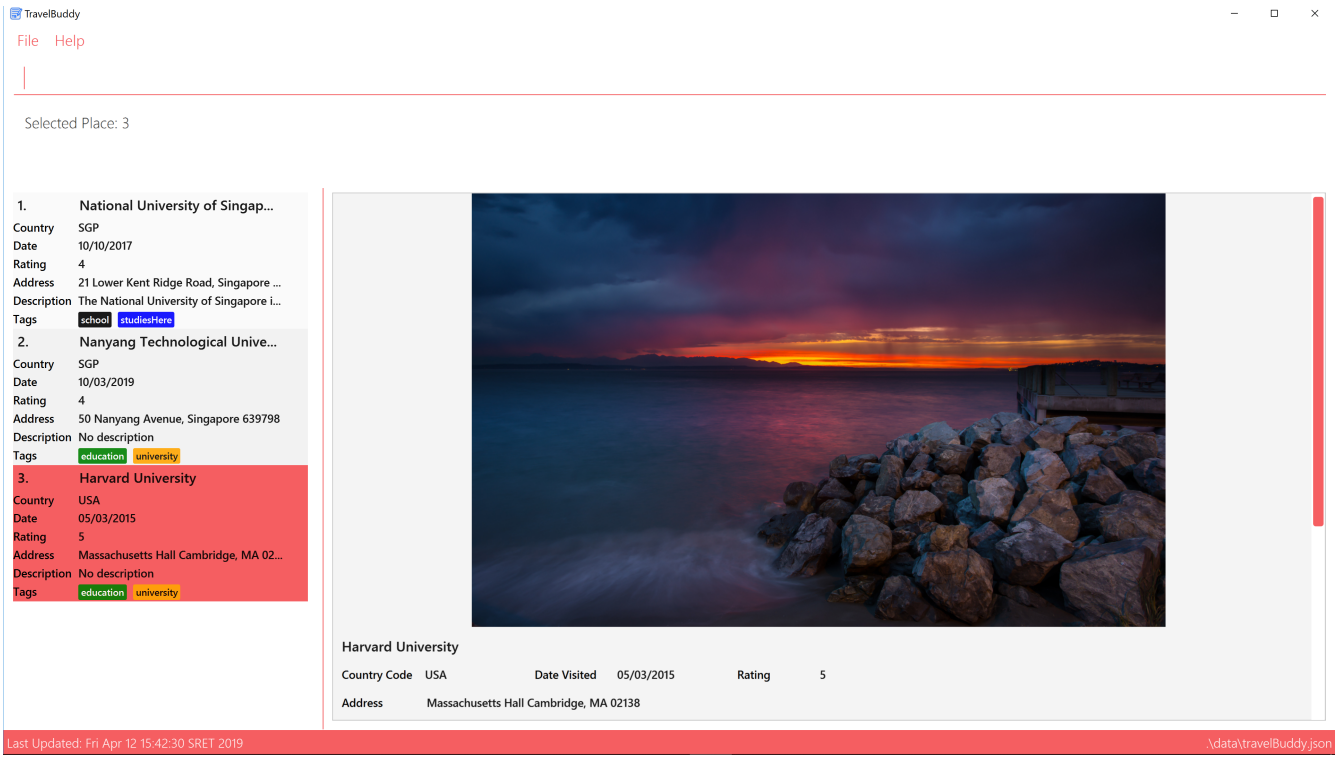


Figure 4.4.1.2: Before the `add` command is executed

Step 2. The user inputs `selects 4`.

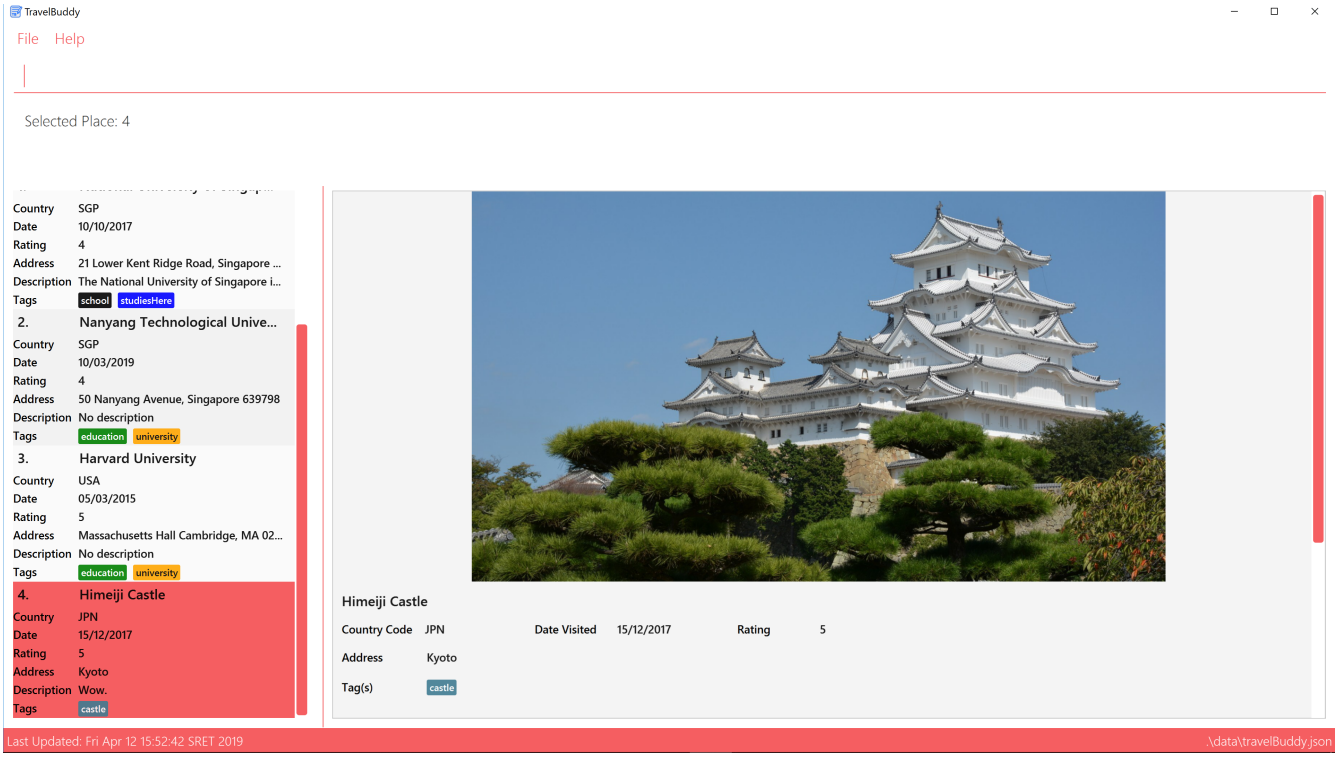


Figure 4.4.1.3: After the `add` command is executed and the newly added Place "Himeiji Castle" is selected

4.4.2. Design Considerations

Decision: What type of file path input should the `add` and `edit` command accept for the photo parameter (prefix `p/`)?

	Implementation 1: Absolute Filepath (current choice)	Implementation 2: Filename and Extension Only
Description	Accept the entire absolute file path of the image to be added/replaced	Accept only file name and file extension of the image to be added/replaced, but requiring the images to be located in the directory of the JAR executable, or some other pre-defined directory.
Pros	**Ease-of-use**. This approach is more user friendly when used together with Windows 10 File Explorer `Copy Path` function.	**Scalability**. This approach is does not require images to be copied or moved from their existing directory to a new directory. Instead, by using `Alt-Tab`, Windows 10 File Explorer `Copy Path` function, `Ctrl-C` + `Ctrl-V`, and the `edit` command, users can quickly attach multiple photos to existing places.
Cons	**Ease-of-use**. This approach is less user-friendly as users will have to manually type in the name and extension of each image file added, or otherwise rename the file to `Ctrl-C` + `Ctrl-V` the file name, and manually type in the extension.	**Scalability**. For a large number of image files, this approach requires the user spent extra time moving files from their existing directory to the new pre-specified directory. If moving files is not an option, then addition storage space is required for all the copied files.

4.5. Chart Feature

The Chart feature displays to users three different charts in TravelBuddy. They are:

- The Number of Places Visited by Rating Category
- The Number of Places Visited by Year
- The Number of Places Visited by Country

The Chart feature is activated in TravelBuddy by default when the application launches. Alternatively, the `generate` command is also used to generate the charts. The `generate` command does not require any parameters.

> **TIP** Instead of typing `generate`, you can simply type the shortcut `g`.

4.5.1. Current Implementation

Logic: The `generate` mechanism is executed by `GenerateCommand`, which extends from `Command`. A code snippet is shown below:

```
public CommandResult execute(Model model, CommandHistory history) {
    requireNonNull(model);
    model.setChartDisplayed(true); ①
    model.commitTravelBuddy(); ②
    if (model.getFilteredPlaceList().isEmpty()) { ③
        return new CommandResult(MESSAGE_EMPTY);
    } else {
        return new CommandResult(MESSAGE_SUCCESS);
    }
}
```

In the snippet, the operations implemented are:

1. `Model#setChartDisplayed(chartDisplayed)` - Signals to the UI to display the charts.
2. `Model#commitTravelBuddy()` - Saves the current TravelBuddy state from its history.
3. `Model#getFilteredPlaceList()` - Verifies if the list is empty.

These operations are exposed in the `Model` interface as `Model#setChartDisplayed(chartDisplayed)`, `Model#commitTravelBuddy()` and `Model#getFilteredPlaceList()`.

Model: The chart generation mechanism is facilitated by `VersionedTravelBuddy`, which extends from `TravelBuddy`, as seen in the Model Class Diagram in Figure 3.4.1.

For this section on Charts, the focus is on the class diagram as seen in Figure 4.5.1.1. The step-by-step explanation of the class diagram can be found below:

1. The `ModelManager` is a container for a `VersionedTravelBuddy` object.
2. VersionedTravelBuddy consists of a `UniquePlaceList` object.

3. `UniquePlaceList` is a container for one or more `ChartBook` objects and for one or more `Place` object.
4. `ChartBook` consists of a `CountryChartList` object, a `RatingChartList` object and a `YearChartList` object.
5. `countryChartList` is a container for one or more `CountryChart` objects. Similarly, `ratingChartList` is a container for one or more `RatingChart` objects and `yearChartList` is a container for one or more `YearChart` objects.
6. `CountryChart` consists of a `CountryCode` object and a `Total` object. Similarly, `RatingChart` consists of a `Rating` object and a `Total` object and `YearChart` consists of a `Year` object and a `Total` object.

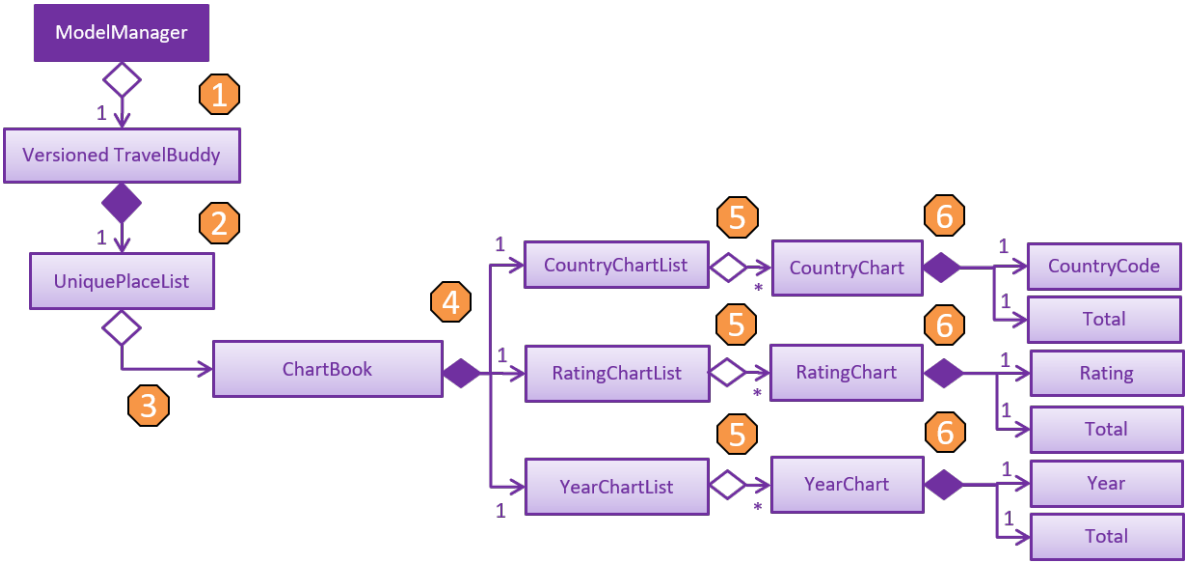


Figure 4.5.1.1: OOP Diagram

Logic & Model Interaction: Having discussed Logic and Model, we can now model the workflow of the `generate` command. This can be accomplished using an activity diagram, as seen in Figure 4.5.1.2.

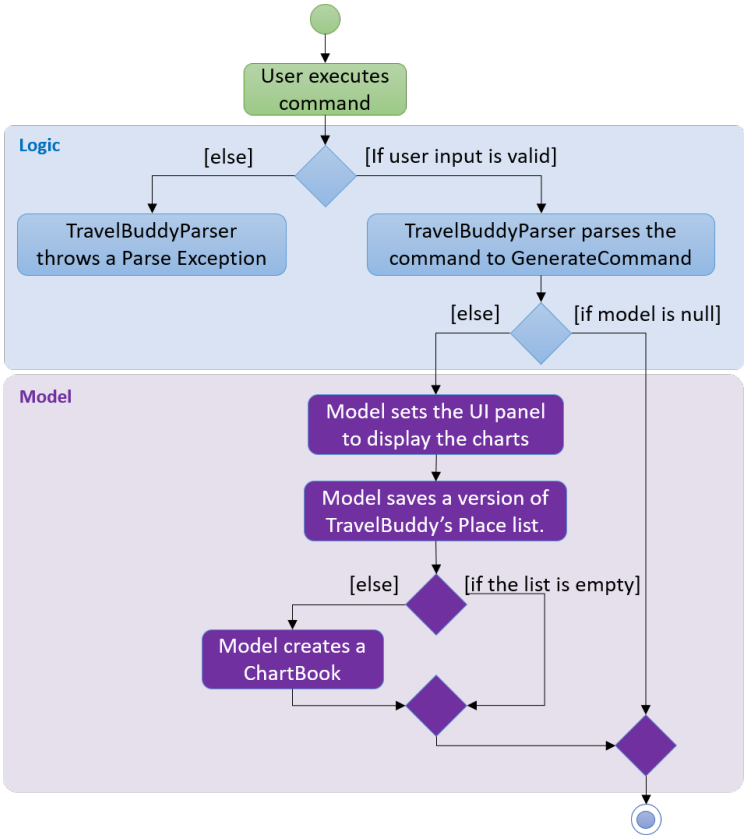


Figure 4.5.1.2: Activity Diagram for the `generate` command

Additionally, we want to be able to capture the interaction between multiple objects for the `generate` command. This can be accomplished using a sequence diagram, as seen in Figure 4.5.1.3 below. The step-by-step explanation of the sequence diagram is as follows:

1. The user enters the command `generate` without any parameters.
2. The command is processed by the Logic component, which will then call `LogicManager#execute()`.
3. The `GenerateCommand#execute()` method is invoked.
4. The `Model#setChartDisplayed()` method is invoked with the argument `true`. The `Model#commitTravelBuddy()` method is also invoked.
5. The `TravelBuddy#commitTravelBuddy()` method is invoked by `Model`.
6. The `ChartBook#commitChart()` method is invoked by `TravelBuddy`.
7. The `CommandResult` object is returned.

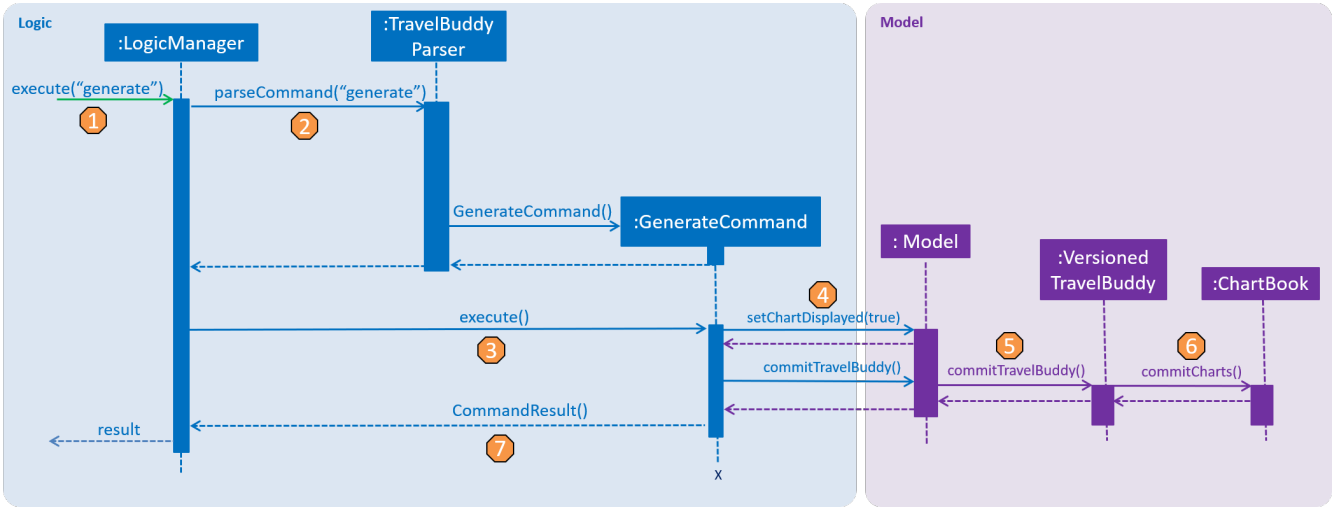


Figure 4.5.1.3: Sequence Diagram for the `generate` command

Storage: The Chart's storage is handled by `JsonChartBookStorage`, which implements from `ChartBookStorage`. The three main data-storage methods it implements are:

- `saveCountryChart(filePath)` - Saves the countries data into a JSON file.
- `saveRatingChart(filePath)` - Saves the ratings data into a JSON file.
- `saveYearChart(filePath)` - Saves the years data into a JSON file.

As an example, a code snippet for `saveCountryChart(filePath)` is shown below.

```
public void saveCountryChart(ReadOnlyCountryChart countryChart, Path filePath) { ①
    requireAllNonNull(countryChart, filePath); ②
    Gson gson = new GsonBuilder().setPrettyPrinting().create(); ③
    try {
        FileWriter fileWriter = new FileWriter(String.valueOf(filePath)); ④
        gson.toJson(countryChart, fileWriter); ⑤
        fileWriter.flush(); ⑥
    } catch (IOException ioe) {
        logger.warning(ioe.getMessage()); ⑦
    }
}
```

In the snippet, a third-party library called Gson was used to convert Java Objects into JSON and back. The Gson API can be found here. A step-by-step explanation is as follows:

1. `saveCountryChart` accepts a `ReadOnlyCountryChart` object, which supplies data for the Country Chart, and a `Path` object, which specifies the file path for the `FileWriter` to write into.
2. Both objects are checked by `requireAllNonNull` to make sure they are not empty.
3. A `Gson` object is instantiated using the `GsonBuilder#setPrettyPrinting()#create()` object.
4. A `FileWriter` object is instantiated with the `Path` object as its parameter.
5. Assuming there are no exceptions, `Gson#toJson()` object will serialize the `ReadOnlyCountryChart` data as a `JsonObject` and place the stream on `FileWriter`.
6. The `FileWriter#flush()` will flush the stream.

7. If an `IOException` occurs, a warning message will be displayed by the `Logger#warning` object.

4.5.2. Generate Command

Preconditions: Given below is a list of preconditions that must be met for the `generate` command to work:

- By default, the charts are automatically generated each time TravelBuddy loads.
- The `generate` command always triggers the display of all three charts.
- The charts always update themselves in real-time.
  Example: When a place is added via the `add` command, the charts are automatically updated so that no `generate` command is necessary.
- The chart will not display anything when the list is empty.
- You can type in any parameters after the `generate` command, TravelBuddy will simply ignore them.

Example: Given below is an example usage scenario of the `generate` command.

Step 1: By default, the charts are displayed when TravelBuddy launches. To navigate away from the charts, type in `select 1`.
Outcome: The first index in the place list will be selected and displayed on the right-hand side of the panel.

Step 2: Type in `generate` to generate the charts.
Outcome: The charts will be displayed on the right-hand side of the panel.

4.5.3. Design Considerations

Aspect: How Chart Generation Executes

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

	Alternative 1 (current choice)	**Alternative 2**
Description	Updates the charts both in real-time and when the `generate` command is used.	Updates the charts only when the `generate` command is used.
Pros	**Ease-of-use**. This approach is more user-friendly, as users do not need to type an additional `generate` command after changes are made to the Place list. **Accuracy**. This approach is more accurate as the charts reflect the latest changes regardless of whether the user types the `generate` command.	**Scalability**. This approach is computationally less intensive, as the charts are only generated when required.

	Alternative 1 (current choice)	Alternative 2
Cons	**Scalability**. This approach is computationally more intensive, especially when the Place list is huge, as all three charts need to be regenerated every time a change is detected.	**Ease-of-use**. This approach is Less user-friendly as users will need to type the `generate` command for the charts to reflect the changes made to the Place list. **Accuracy**. This approach is less accurate as the charts does not reflect the latest changes until the user types the `generate` command.
Example	As shown in Figure 4.5.3.1, before the `edit 1 cc/USA` command was executed, the chart did not have a separate bar for USA. After the command was executed, the chart updated in real-time to include a bar for USA. All this was done without invoking the `generate` command.	N.A.

Decision: Alternative 1, which is to update the charts in real-time, was adopted as it promotes ease-of-use, so users are not required to type in an additional `generate` command whenever changes are made to the Place list. Moreover, Alternative 1 is the more accurate option of the two, as the chart reflects the latest changes even if the user forgets to type the `generate` command.

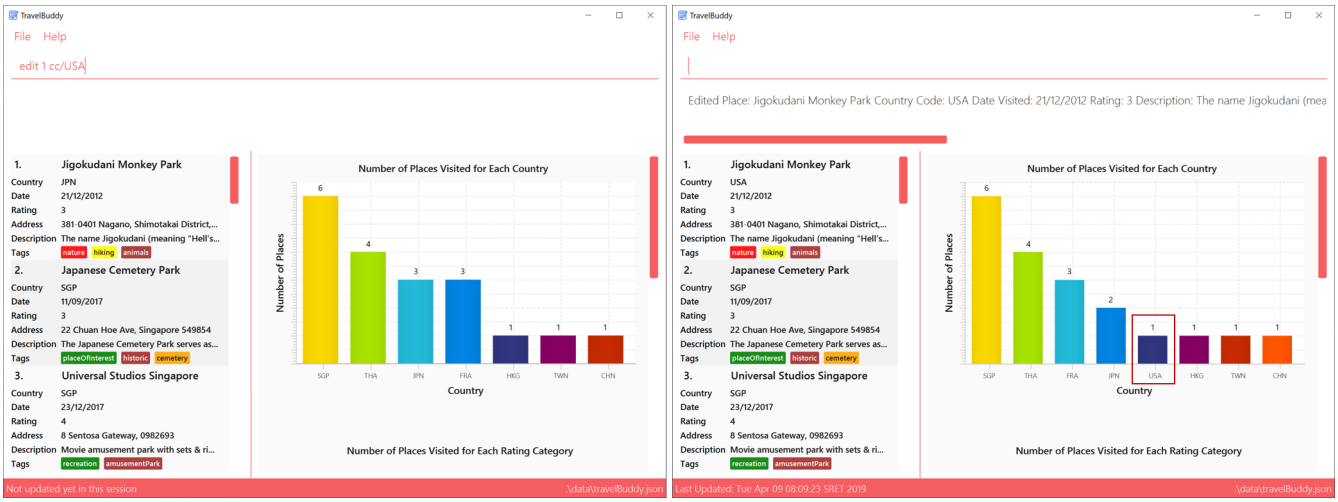


Figure 4.5.3.1: A comparison before and after the `edit` command was executed

4.6. 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 4.7, “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` : Proceed 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

4.7. Configuration

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

5. Documentation

We use asciidoc for writing documentation.

NOTE	We chose asciidoc over Markdown because asciidoc, although a bit more complex than Markdown, provides more flexibility in formatting.

5.1. Editing Documentation

See UsingGradle.adoc to learn how to render `.adoc` files locally to preview the end result of your edits. Alternatively, you can download the AsciiDoc plugin for IntelliJ, which allows you to preview the changes you have made to your `.adoc` files in real-time.

5.2. Publishing Documentation

See UsingTravis.adoc to learn how to deploy GitHub Pages using Travis.

5.3. Converting Documentation to PDF Format

We use Google Chrome for converting documentation to PDF format, as Chrome's PDF engine preserves hyperlinks used in webpages.

Here are the steps to convert the project documentation files to PDF format.

1. Follow the instructions in UsingGradle.adoc to convert the AsciiDoc files in the `docs/` directory to HTML format.
2. Go to your generated HTML files in the `build/docs` folder, right click on them and select `Open with → Google Chrome`.
3. Within Chrome, click on the `Print` option in Chrome's menu.
4. Set the destination to `Save as PDF`, then click `Save` to save a copy of the file in PDF format. For best results, use the settings indicated in the screenshot below.

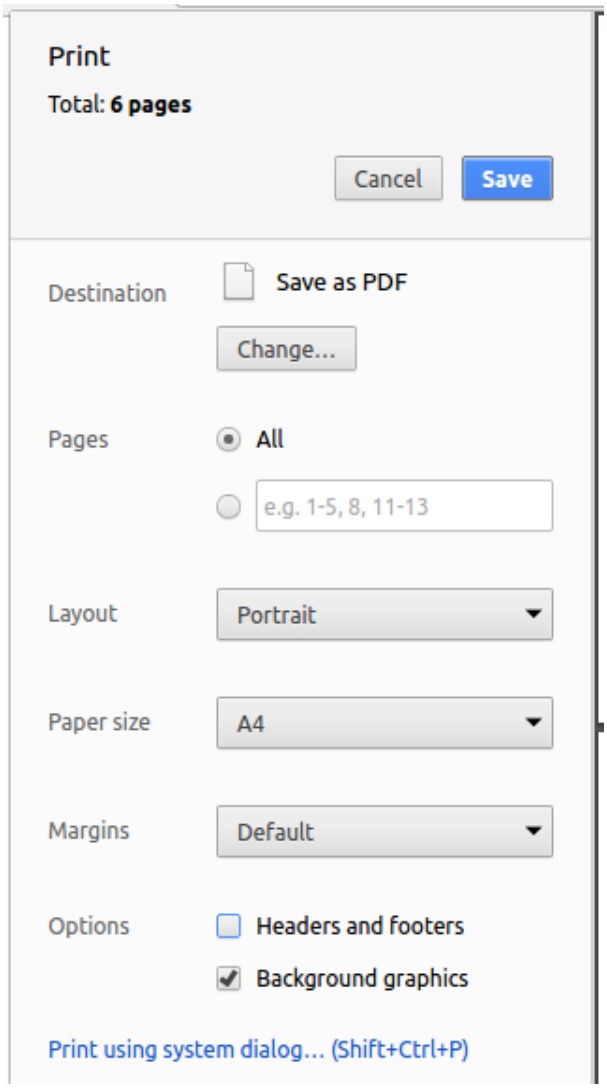


Figure 1. Saving documentation as PDF files in Chrome

5.4. Site-wide Documentation Settings

The `build.gradle` file specifies some project-specific asciidoc attributes which affects how all documentation files within this project are rendered.

TIP Attributes left unset in the `build.gradle` file will use their **default value**, if any.

Table 1. List of site-wide attributes

Attribute name	Description	Default value
`site-name`	The name of the website. If set, the name will be displayed near the top of the page.	*not set*
`site-githuburl`	URL to the site's repository on GitHub. Setting this will add a "View on GitHub" link in the navigation bar.	*not set*

Attribute name	Description	Default value
`site-seedu`	Define this attribute if the project is an official SE-EDU project. This will render the SE-EDU navigation bar at the top of the page, and add some SE-EDU-specific navigation items.	*not set*

5.5. Per-file Documentation Settings

Each `.adoc` file may also specify some file-specific asciidoc attributes which affects how the file is rendered.

Asciidoctor's built-in attributes may be specified and used as well.

TIP Attributes left unset in `.adoc` files will use their **default value**, if any.

Table 2. List of per-file attributes, excluding Asciidoctor's built-in attributes

Attribute name	Description	Default value
`site-section`	Site section that the document belongs to. This will cause the associated item in the navigation bar to be highlighted. One of: `UserGuide`, `DeveloperGuide`, `LearningOutcomes`*, `AboutUs`, `ContactUs` ** Official SE-EDU projects only*	*not set*
`no-site-header`	Set this attribute to remove the site navigation bar.	*not set*

5.6. Site Template

The files in `docs/stylesheets` are the CSS stylesheets of the site. You can modify them to change some properties of the site's design.

The files in `docs/templates` controls the rendering of `.adoc` files into HTML5. These template files are written in a mixture of Ruby and Slim.

WARNING Modifying the template files in `docs/templates` requires some knowledge and experience with Ruby and Asciidoctor's API. You should only modify them if you need greater control over the site's layout than what stylesheets can provide. The SE-EDU team does not provide support for modified template files.

6. Testing

6.1. Running Tests

There are three ways to run tests.

TIP	The most reliable way to run tests is the 3rd one. The first two methods might fail some GUI tests due to platform/resolution-specific idiosyncrasies.

Method 1: Using IntelliJ JUnit test runner

- To run all tests, right-click on the `src/test/java` folder and choose `Run 'All Tests'`
- To run a subset of tests, you can right-click on a test package, test class, or a test and choose `Run 'ABC'`

Method 2: Using Gradle

- Open a console and run the command `gradlew clean allTests` (Mac/Linux: `./gradlew clean allTests`)

NOTE	See UsingGradle.adoc for more info on how to run tests using Gradle.

Method 3: Using Gradle (headless)

Thanks to the TestFX library we use, our GUI tests can be run in the *headless* mode. In the headless mode, GUI tests do not show up on the screen. That means the developer can do other things on the Computer while the tests are running.

To run tests in headless mode, open a console and run the command `gradlew clean headless allTests` (Mac/Linux: `./gradlew clean headless allTests`)

6.2. Types of Tests

We have two types of tests:

1. **GUI Tests** - These are tests involving the GUI. They include,
   a. *System Tests* that test the entire App by simulating user actions on the GUI. These are in the `systemtests` package.
   b. *Unit tests* that test the individual components. These are in `seedu.travel.ui` package.
2. **Non-GUI Tests** - These are tests not involving the GUI. They include,
   a. *Unit tests* targeting the lowest level methods/classes.
   e.g. `seedu.travel.commons.StringUtilTest`
   b. *Integration tests* that are checking the integration of multiple code units (those code units are assumed to be working).
   e.g. `seedu.travel.storage.StorageManagerTest`

c. Hybrids of unit and integration tests. These test are checking multiple code units as well as how the are connected together.
   e.g. `seedu.travel.logic.LogicManagerTest`

6.3. Troubleshooting Testing

Problem: `HelpWindowTest` fails with a `NullPointerException`.

- Reason: One of its dependencies, `HelpWindow.html` in `src/main/resources/docs` is missing.
- Solution: Execute Gradle task `processResources`.

7. Dev Ops

7.1. Build Automation

See UsingGradle.adoc to learn how to use Gradle for build automation.

7.2. Continuous Integration

We use Travis CI and AppVeyor to perform *Continuous Integration* on our projects. See UsingTravis.adoc and UsingAppVeyor.adoc for more details.

7.3. Coverage Reporting

We use Coveralls to track the code coverage of our projects. See UsingCoveralls.adoc for more details.

7.4. Documentation Previews

When a pull request has changes to asciidoc files, you can use Netlify to see a preview of how the HTML version of those asciidoc files will look like when the pull request is merged. See UsingNetlify.adoc for more details.

7.5. Making a Release

Here are the steps to create a new release.

1. Update the version number in `MainApp.java`.
2. Generate a JAR file using Gradle.
3. Tag the repo with the version number. e.g. `v0.1`
4. Create a new release using GitHub and upload the JAR file you created.

7.6. Managing Dependencies

A project often depends on third-party libraries. For example, TravelBuddy depends on the Jackson library for JSON parsing. Managing these *dependencies* can be automated using Gradle. For example, Gradle can download the dependencies automatically, which is better than these alternatives:

a. Include those libraries in the repo (this bloats the repo size)

b. Require developers to download those libraries manually (this creates extra work for developers)

Appendix A: Product Scope

Target user profile: The target audience that we have identified are stated below.

- The user has a need to manage a significant number of places
- The user prefers desktop apps over other types
- The user can type fast
- The user prefers typing over mouse input
- The user is reasonably comfortable using CLI apps

Value proposition: Manage contacts faster than a typical mouse/GUI driven app

Appendix B: User Stories

The priority of user stories are divided into three categories:

- High (must have) - * * *
- Medium (nice to have) - * *
- Low (unlikely to have) - *

Table 3. User Stories

Priority	As a …	I want to …	So that …
* * *	user	search places by their names	I can quickly retrieve information about that place
* * *	user	keep track of all the places I have visited	I can easily search for them in the future
* * *	user	rate the places I have visited	I know which places I want to revisit
* * *	user	be able to search for places by ratings	I can recommend the highly-rated places to others and avoid the lowly-rated ones
* * *	user	be able to search for places by countries	I can plan for future trips

Priority	As a …	I want to …	So that …
* * *	user	search places by their tag(s)	I can quickly compile a list of places with related tags
* *	user	know the countries and continents that I have not visited	I may visit them in future
* *	user	see a graphical representation of the places I have visited	I can plan which country to visit next
* *	user	to know the number of times I have revisited a place	I know which places are popular for me
* *	user	search for the places I have visited by years	I am able to see which places I have visited in by year
*	user	have a map view indicating all of the places I have visited	I can quickly find out the places by countries I have visited in a user-friendly way
*	user	I want to get the locations of highly-rated places in my vicinity	find a place to visit

Appendix C: Use Cases

For all use cases below, the **System** is the `TravelBuddy` and the **Actor** is the `user`, unless specified otherwise.

Use case: Deleting a place

MSS

1. User requests to list places
2. TravelBuddy shows a list of places
3. User requests to delete a specific place in the list
4. TravelBuddy deletes the place

   Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index is invalid.

3a1. TravelBuddy shows an error message "The place index provided is invalid".

Use case resumes at step 2.

3b. The given command is invalid.

3b1. TravelBuddy shows an error message "Unknown command".

Use case resumes at step 2.

Use case: Search for a place

MSS

1. User requests to list places
2. TravelBuddy shows a list of places
3. User searches for a place with a specific word
4. TravelBuddy displays all places with the specified word.

Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given word is not found.

3a1. TravelBuddy shows an error message "0 places listed!".

Use case resumes at step 2.

3b. The given command is invalid.

3b1. TravelBuddy shows an error message "Unknown command".

Use case resumes at step 2.

Use case: Edits a place

MSS

1. User requests to list places
2. TravelBuddy shows a list of places
3. User edits a place with a specified index and a specified parameter.
4. TravelBuddy identifies the place and updates the changes on the specified parameter.

Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index is invalid.

3a1. TravelBuddy shows an error message "Invalid command format!".

Use case resumes at step 2.

3b. The given prefix is invalid.

3b1. TravelBuddy shows an error message "Invalid command format!".

Use case resumes at step 2.

3c. The given parameter is invalid.

3c1. TravelBuddy shows an error message depending on the parameter.

Use case resumes at step 2.

3d. The given command is invalid.

3d1. TravelBuddy shows an error message "Unknown command".

Use case resumes at step 2.

Appendix D: Non-Functional Requirements

The non-functional requirements of the system are listed below:

1. **Operating System (OS) Compatability:** Should work on any mainstream OS as long as it has Java 9 or higher installed.
2. **Storage Requirements:** Should store up to 1,000 places without a noticeable sluggishness in performance for typical usage.
3. **Usage Efficiency:** 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. **Constraints:** Should be backward compatible with data produced by earlier versions of the system;
5. **Technical Requirements:** Should work on both 32-bit and 64-bit environments.
6. **Speed Requirements:** Should respond within two seconds.
7. **Quality Requirements:** Should be usable by a novice who has never used a travel log

Appendix E: Glossary

Table 4. A List of Terms Used in this Developer Guide

Terms	Definitions
Activity Diagram	Activity diagrams are diagrams that can model workflows. Activity diagrams are the UML equivalent of flow charts.
Actor	An actor (in a use case) is a role played by a user. An actor can be a human or another system. Actors are not part of the system; they reside outside the system.
Class Diagram	Class diagrams describe the structure (but not the behavior) of an OOP solution.
Extension	Extensions are "add-on"s to the MSS that describe exceptional or alternative flow of events. They describe variations of the scenario that can happen if certain things are not as expected by the MSS.
Integrated Development Environment	Integrated Development Environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development.
Java Development Kit	Java Development Kit (JDK) is a software development environment used for developing Java applications and applets.
Mainstream OS	Some mainstream operating systems include Windows, Linux, Unix and OS-X.
MSS	Main Success Scenario (MSS) describes the most straightforward interaction for a given use case, which assumes that nothing goes wrong.
Sequence Diagram	Sequence diagrams capture the interactions between multiple objects for a given scenario.
Use Case	A use case is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor.
User Story	User stories are short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.

Appendix F: 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.

Launching 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.

Deleting a Place

1. Deleting a place while all places are listed

    a. Prerequisites: List all places using the `list` command. Multiple places 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 place 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)

    Expected: Similar to previous.