CelebManager

Developer Guide

Product of CS2103JAN2018-W14-B4
Last Updated: 14 April 2018

Table of Contents

1 Introduction

CelebManager is a software that aims to allow celebrity managers to maintain schedules of celebrities under them efficiently. CelebManager is optimized for celebrity managers (users) who **prefer to work with a Command Line Interface (CLI)** while still having a Graphical User Interface (GUI) for visual feedback.

In CelebManager, the celebrity managers should be able to enter commands using CLI to:

- Manage contacts (including celebrities) in an address book,
- Manage appointments for celebrities with calendars, and;
- Plan efficient routes for celebrities with interactive map.

If you are new to software development, this developer guide aims to allow you to start working from the basics. Information on how to set up the project, how the software is designed in general, and how the software should be developed and maintained is in this documentation, which will help you contribute to this project.

Similarly, if you are experienced in software development, this developer guide also details specific considerations for important features developed, which you can refer to when working with the project.

2 Setup

In this section, we will discuss the setting up of project for development.

2.1 Prerequisites

There are two prerequisites before you can work on this software.

1. Java Development Kit (JDK)

   The Java programming language is used in this project. To be able to work with this project, you will need to have JDK version `1.8.0_60` installed.

   You can download JDK here.

   

   Some components of this software will not work with earlier versions of Java 8.

2. IntelliJ Integrated Development Environment (IDE)

   This software is developed as a Gradle project, which requires you to work on the software using a IDE. While it is possible to work with any IDE that supports Gradle Projects, this guide will use IntelliJ as a basis.

   You can download IntelliJ here.

   

   IntelliJ make use of Gradle and JavaFx plugins, and the project will need these plugins.

   If you have disabled them, you can go to `File` > `Settings` > `Plugins` to enable them.

2.2 Local Project Setup

To contribute to this project, you will need to work using a local copy of this project. To have a local copy of this project, you will need to:

1. Fork the repository from here.
2. Clone the fork to your computer.
3. Open IntelliJ.

If you are not in the welcome screen, you can go to `File` > `Close Project` to close the exisiting project dialog.

4. 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.
5. Click `Import Project`.
6. Locate the `build.gradle` file, select it and click `OK`.
7. Click `Open as Project`.
8. Click `OK` to accept the default settings.
9. Open a console and run the command `gradlew processResources` (for Windows) or `./gradlew processResources` (for MacOS/Linux).

This will generate all resources required by the software and tests, and finish with `BUILD SUCCESSFUL` message.

2.3 Setup Verification

To ensure that you have set up the local copy of the project correctly, you will need to:

1. Run the `seedu.address.MainApp` and try the commands stated in the User Guide.
2. Run the tests and ensure that all the tests pass.

Please refer to Section 6: Testing for more information on how to run tests for this project.

2.4 Project Configurations

There are three project configurations that needs to be done to ensure that code and documentation quality are not compromised.

2.4.1 Coding Style Configurations

This project follows the oss-generic coding standards. IntelliJ uses a different package importing order, and to rectify:

1. Go to `File` > `Settings`... (for Windows/Linux) or `IntelliJ IDEA` > `Preferences` (for MacOS).
2. Select `Editor` > `Code Style` > `Java`.
3. Click on the `Imports` tab to set the order.
4. Set `Class count to use import with '*'` and `Names count to use static import with '*'` to 999 to prevent IntelliJ from contracting import statements.
5. Set `Import Layout` order as:
    a. `import static all other imports`
    b. `import java.*`
    c. `import javax.*`
    d. `import org.*`
    e. `import com.*`
    f. `import all other imports`
6. Add a <blank line> between each import.

Alternatively, you can follow the UsingCheckstyle documentation to configure coding style.

2.4.2 Documentation Configurations

After forking the repository in Section 2.2 Project Setup, the links in the documentation will still lead to `CS2103JAN2018-W14-B4/main` repository. If you intend to develop this project as a separate product instead of contributing to the `CS2103JAN2018-W14-B4/main` repository, you should replace the variable `repoURL` in `DeveloperGuide.adoc` and `UserGuide.adoc` with the URL of your fork.

2.4.3 Continuous Integration (CI) Configurations

There are two CI configurations, Travis and AppVeyor, that you can set up.

Travis is an Unix-based software, while AppVeyor is a Windows-based software.

To set up Travis and AppVeyor for your fork, please refer to the UsingTravis documentation and UsingAppVeyor documentation respectively.

Having both Travis CI and AppVeyor CI ensures your App works on both Unix-based platforms and Windows-based platforms.

2.4.4 Coverage Reporting Configurations

You should also set up coverage reporting for your team fork, if you have one. Please refer to UsingCoveralls documentation.

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

3 Design

In this section, we will discuss the design of the software. This section is essential before you start working on the project, together with the product scope in Appendix A: Product Scope.

3.1 Software Architecture

The architecture diagram in *Figure 1* shows the high-level design of the project.

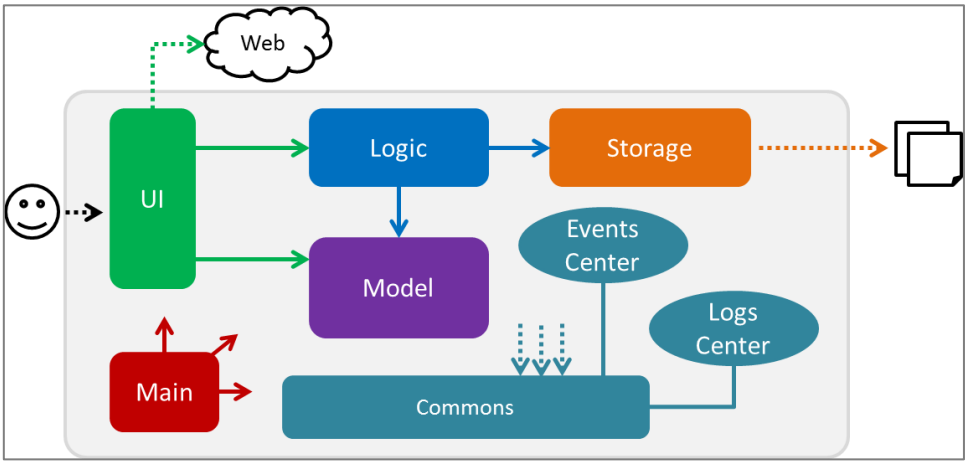

Figure 1: Architecture diagram for CelebManager

Components, denoted by the rounded rectangles in *Figure 1*, are packages in the `seedu.address` package containing its respective class files.

The project consists of six components: `Main`, `Commons`, `UI`, `Logic`, `Model` and `Storage`.

Each component communicates with some other components, and are discussed with further details in the rest of the subsections.

The `UI`, `Logic`, `Model` and `Storage` components can be accessed by methods in the application programming interface (API) defined in the interface files.

3.2 `Main` Component

The `Main` component is responsible for initializing the other components in the correct sequence and shutting down the rest of the components where necessary. It contains one class file `MainApp` which acts as the driver and main entry point for the CelebManager.

3.3 `Commons` Component

The `Commons` component represents a collection of classes that are used by the other components directly. Two of these classes are vital at the architecture level. They are:

- `EventsCenter`, which is used by components to communicate with other components using events, and;
- `LogsCenter`, which is used by many classes to write log messages to the software's log file.

The `EventsCenter` class is written using the Google's Event Bus Library, and it allows us to implement an event-driven design for the project.

The sequence diagram in *Figure 2* shows how the components interact using `EventsCenter`, when the user issues the command `delete 1`. The `Model` component raises an `AddressBookChangedEvent` when the data has changed, instead of interacting with the `Storage` component to update the hard disk.

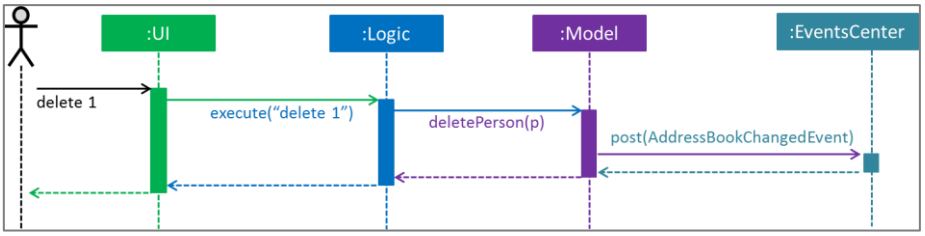

Figure 2: Sequence diagram for `delete 1` command (I)

The sequence diagram in *Figure 3* extends from *Figure 2*, which shows how the `EventsCenter` class reacts to the command. The update is saved to the hard disk, and status bar of the `UI` component is updated to reflect the `lastUpdated` time. The event is propagated through the `EventsCenter` to the `Storage` component and the `UI` component without the `Model` component being coupled to either of them.

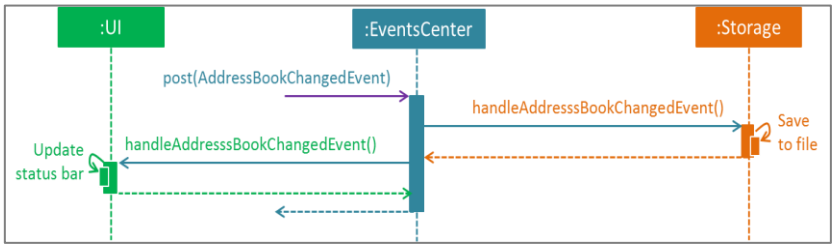

Figure 3: Sequence diagram for `delete 1` command (II)

3.4 `UI` Component

The class diagram of the `UI` component is shown in *Figure 4*. The API is available in `UI.java`.

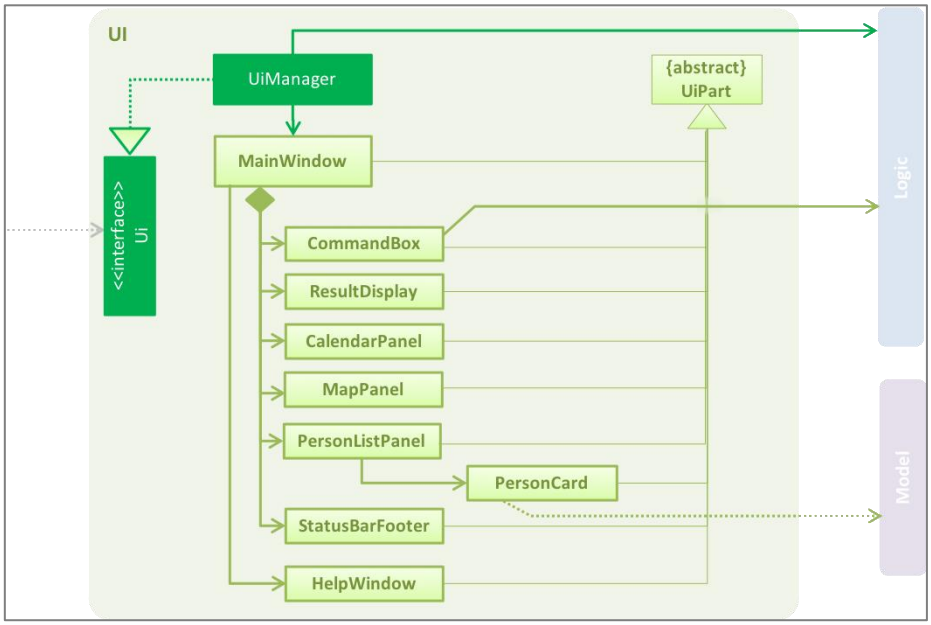

Figure 4: Class diagram of the `UI` component

The `UI` component acts as the user interface of the software, which consists of a `MainWindow` that is made up of elements, such as `CommandBox`, `ResultDisplay` and `PersonListPanel`. These classes inherit from the `UiPart` abstract class.

As the UI component uses JavaFX, the layout of the UI parts is 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`.

JavaFX is a GUI library running on Java Runtime Environment(JRE) for creating desktop applications.

The `UI` component is responsible for:

- executing commands using the `Logic` component,
- binding itself to some data in the `Model` component so that the user interface can update automatically when data in the `Model` component changes, and;
- responding to events raised from various parts of the software and updates the user interface accordingly.

3.5 `Logic` Component

The class diagram of the `Logic` component is shown in *Figure 5*. The API is available in `Logic.java`.

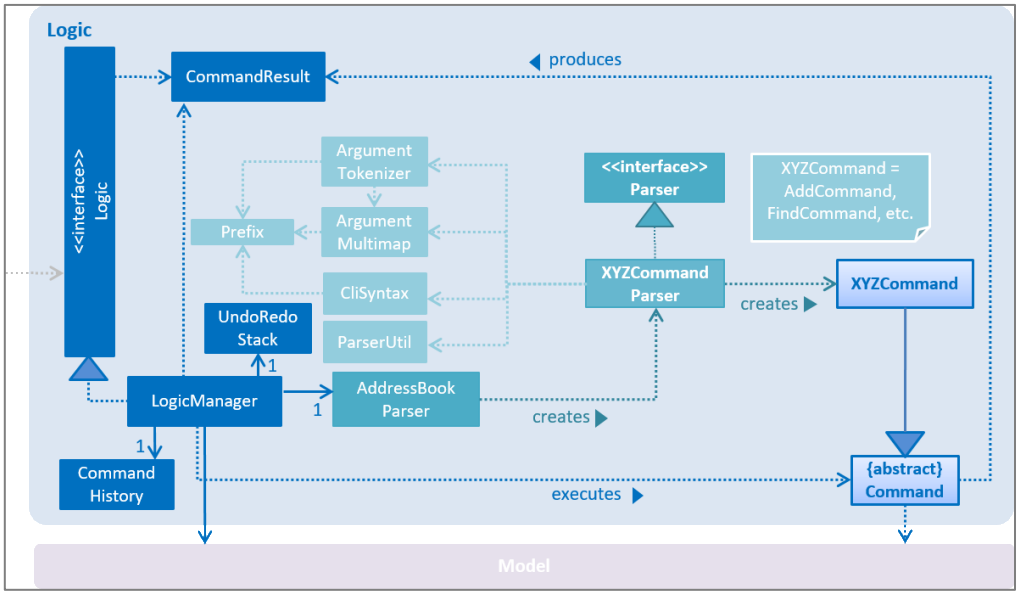

Figure 5: Class diagram of the `Logic` component

The `Logic` component acts as the command executor of the software. It makes use of parsers that creates a `Command`. The class diagram of the `Command` segment in the `Logic` component is shown in *Figure 6*.

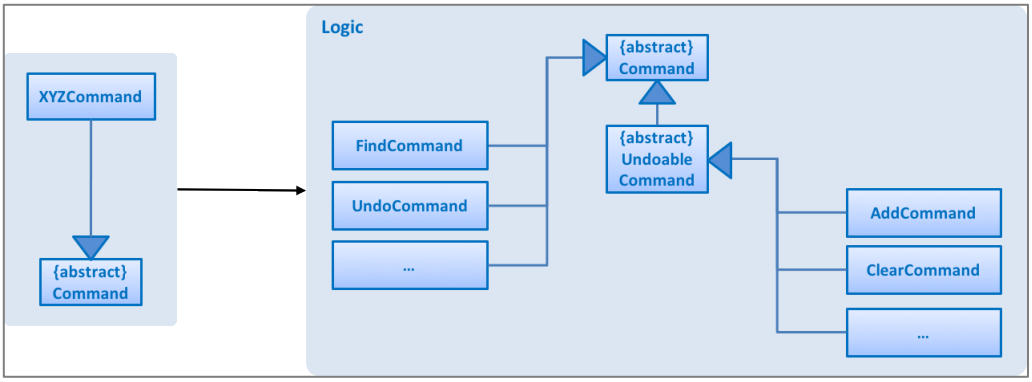

Figure 6: Class diagram of the `Command` segment

The `Logic` component is responsible for:

- parsing input as a `Command` object, and;
- handing results of the commands to `Model` component.

Extending from *Figure 2* in Section 3.2: `Commons` Component, the sequence diagram in *Figure 7* describes the interactions within the `Logic` component for executing `delete 1` command.

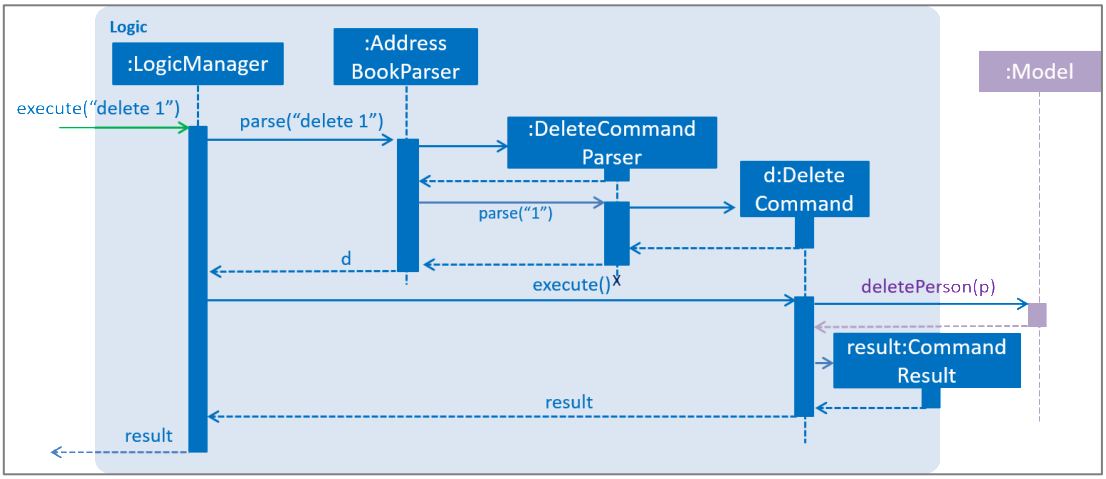

Figure 7: Sequence diagram for `delete 1` command in the `Logic` component

3.6 `Model` Component

The class diagram of the `Model` component is shown in *Figure 8*. The API is available in `Model.java`.

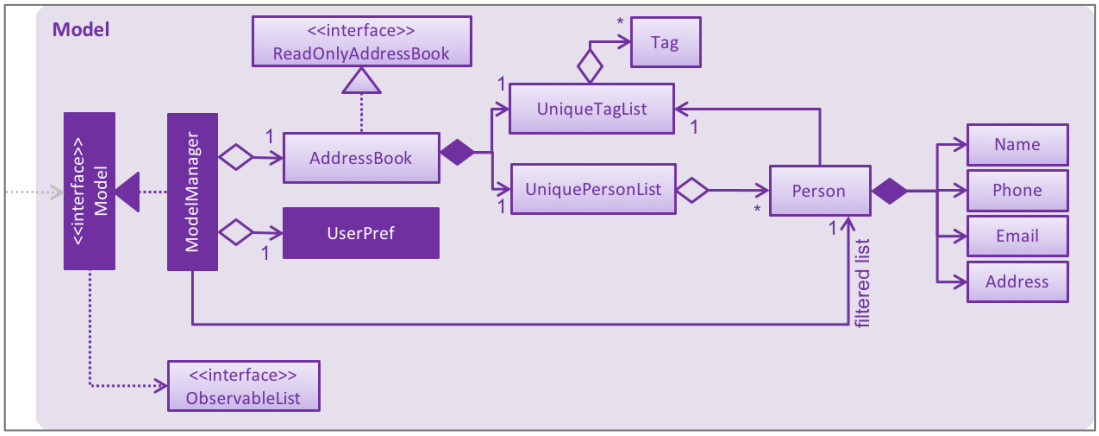

Figure 8: Class diagram of the `Model` component

From *Figure 8*, there are no outgoing arrows from the `Model` component, which represents zero dependency on the other components.

The `Model` component acts as the container for in-memory data of the software. It is responsible for:

- storing user's preferences a `UserPref` object,
- storing the address book and calendar data, and;
- exposing an unmodifiable `ObservableList<Person>` that can be observed which allows the `UI` component to update automatically when data changes.

3.7 `Storage` Component

The class diagram of the `Storage` component is shown in *Figure 9*. The API is available in `Storage.java`.

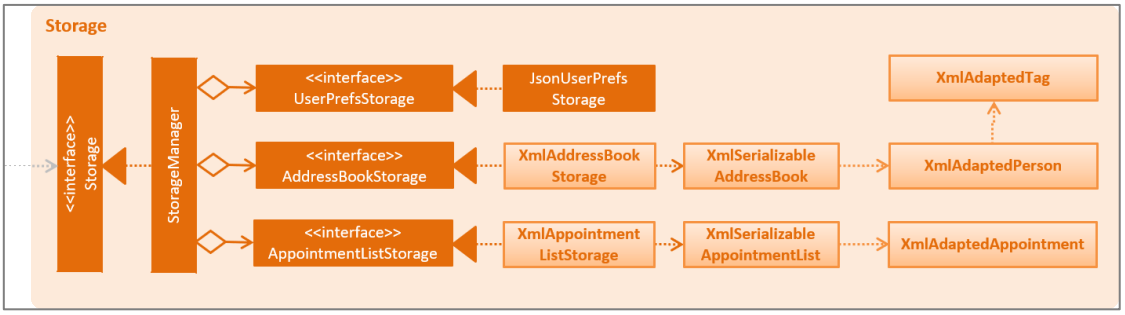

Figure 9: Class diagram of `Storage` component

The `Storage` component acts as the drive for reading and writing data the software. It is responsible for saving `UserPref` objects in `.json` format, contacts and appointments data in `.xml` format, and allowing readback from these files.

4 Implementation

This section describes noteworthy features that are implemented in CelebManager that involves the address book, the calendar and the interactive map.

4.1 Remove Tag Feature

This feature allows the user to remove a particular tag from all the contacts in the address book panel.

4.1.1 Current Implementation

The tag removal mechanism is facilitated by both the `RemoveTagCommand` class residing in the `Logic` component, and the `removeTag` method residing in the `Model` component.

The following code snippet shows the implementation of the `RemoveTagCommand` class in the `Logic` component:

```
public class RemoveTagCommand extends UndoableCommand {
    ...

    public CommandResult executeUndoableCommand() throws CommandException {
        requireNonNull(tagToRemove);

        if (tagToRemove.equals(CELEBRITY_TAG)) {
            throw new CommandException(MESSAGE_CANNOT_REMOVE_CELEBRITY_TAG);
        }
        int numberOfAffectedPersons = 0;
        try {
            numberOfAffectedPersons = model.removeTag(tagToRemove);
        } catch (TagNotFoundException tnfe) {
            throw new CommandException(String.format(MESSAGE_TAG_NOT_FOUND,
                    tagToRemove.toString()));
        } catch (DuplicatePersonException dpe) {
            throw new CommandException(MESSAGE_DUPLICATE_PERSON);
        } catch (PersonNotFoundException pnfe) {
            throw new AssertionError("The target person cannot be missing");
        }
        return new CommandResult(String.format(
                MESSAGE_DELETE_TAG_SUCCESS,
                tagToRemove.toString(),
                numberOfAffectedPersons));
    }
    ...
}
```

The `RemoveTagCommand` class disallows the removal of celebrity tag or a non-existing tag. Upon a successful execution, it will print out successful message with the number of person(s) affected by this removal.

The following shows the implementation of the `removeTag` method in the `Model` component:

```
public class AddressBook {

    public int removeTag(Tag tag) throws PersonNotFoundException,
                 DuplicatePersonException, TagNotFoundException {
        boolean tagExists = false;
        for (Tag existingTag: tags) {
            if (existingTag.equals(tag)) {
                    tagExists = true;
            }
        }
        if (!tagExists) {
            throw new TagNotFoundException();
        }
        int count = 0;
        for (Person person: persons) {
            if (person.hasTag(tag)) {
                //get the new tag set with the specified tag removed
                Set<Tag> oldTags = person.getTags();
                Set<Tag> newTags = new HashSet<>();
                for (Tag tagToKeep: oldTags) {
                    if (tagToKeep.equals(tag)) {
                        continue;
                    }
                    newTags.add(tagToKeep);
                }
                EditCommand.EditPersonDescriptor editPersonDescriptor = new
                         EditCommand.EditPersonDescriptor();
                editPersonDescriptor.setTags(newTags);
                Person editedPerson = createEditedPerson(person,
                         editPersonDescriptor);
                Person syncedEditedPerson = syncWithMasterTagList(editedPerson);
                persons.setPerson(person, syncedEditedPerson);
                removeUnusedTags();
                count++;
            }
        }
        return count;
    }
    ...
}
```

The `removeTag` method makes use of the `EditPersonDescriptor` class to create a new person without the tag for replacing the original person with the tag.

Additionally, the `removeUnusedTags` method is called when there is at least one person affected by the removal. The unused tag should be removed from the list of tags inside the `Model` component.

Figure 10 shows the sequence diagram of the `removeTag` command.

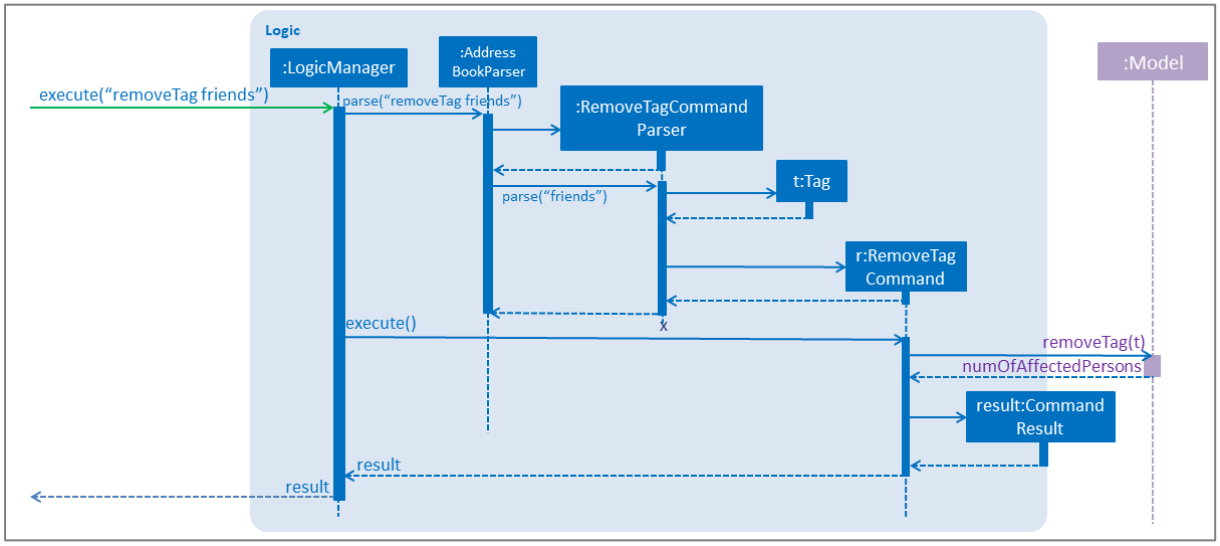

Figure 10: Sequence diagram of `removeTag` command

For example, assuming the `Model` component has two types of tags: "friends" and "colleagues". *Figure 11* shows the address book panel in CelebManager the two tags.

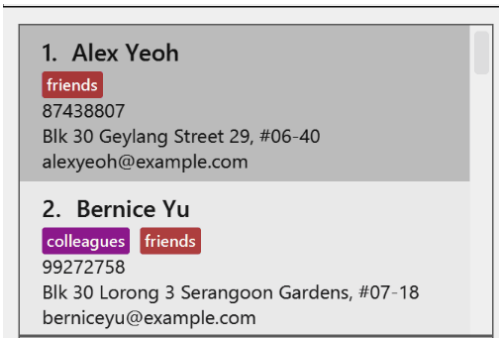

Figure 11: Address book panel before executing `removeTag friends` command

Figure 12 shows the state of PersonalListPanel after executing `removeTag friends` command, with the tag "friends" missing from the address book panel.

Figure 12: Address book panel before executing `removeTag friends` command

4.1.2 Design Considerations

Design Aspect: Command result for removal of `celebrity` tag

- **Alternative 1 (Current choice):** It provides an error message stating that the celebrity tag cannot be removed.
    - **Pros:** It prevents `removeTag` from affecting the calendars as celebrities will not get affected by this operation.
    - **Cons:** It results in no available method to remove `celebrity` tag when necessary.
- **Alternative 2:** It allows removal of `celebrity` tag and clears all calendars.
    - **Pros:** It provides an easy way to remove `celebrity` tag conveniently.
    - **Cons:** It requires the creation of additional events to interact with the calendar panel as calendars are under `UI` component. However, `Logic` component has no directly access to `UI` component.

4.2 Undo / Redo Feature

This undo and redo feature allows the user to reverse the effect of the previous command and the effect of undoing commands respectively.

4.2.1 Current Implementation

The feature is facilitated by an `UndoRedoStack` class in the `Logic` component. It consists of 2 stacks (`undoStack`, `redoStack`), and supports undoing and redoing of commands that modify the state of the address book. In the `Logic` component implementation, these commands will inherit from `UndoableCommand` class, while the commands that cannot be undone will inherit from the `Command` class instead.

The inheritance diagram for `UndoableCommand` class is shown in *Figure 13*.

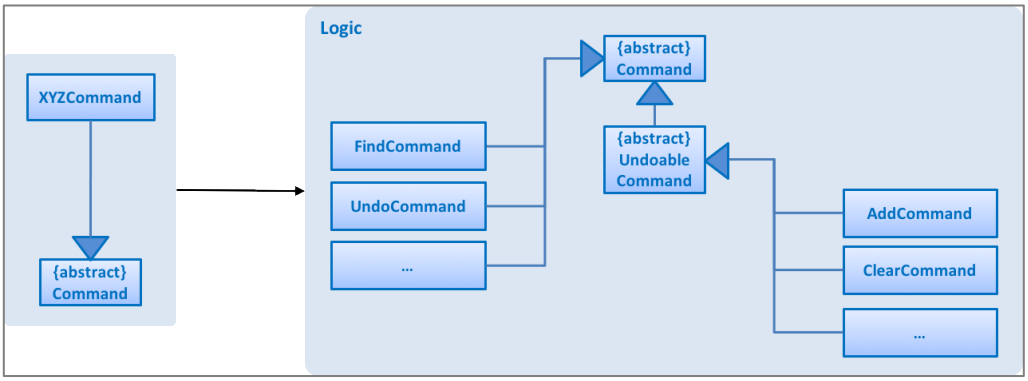

Figure 13: Inheritance Diagram for `UndoableCommand`

The `UndoableCommand` abstract class provides an interface between the `Command` abstract class and command classes that can be undone, such as the `DeleteCommand` class. It contains high-level algorithms for additional tasks, such as saving the application state before command execution. The command classes then implement the methods to execute the specific command. These classes require additional tasks to be completed, such as saving the application state before command execution.

The technique of containing the high-level algorithms in the parent class, while implementing lower-level algorithms in child classes is also known as the template pattern.

The `UndoableCommand` abstract class is implemented in the code snippet below:

```
public abstract class UndoableCommand extends Command {
    @Override
    public CommandResult execute() {
        // ... undo logic ...

        executeUndoableCommand();
    }
}

public class DeleteCommand extends UndoableCommand {
    @Override
    public CommandResult executeUndoableCommand() {
        // ... delete logic ...
    }
}
```

Command classes that does not that are not undoable are implemented as below:

```
public class ListCommand extends Command {
    @Override
    public CommandResult execute() {
        // ... list logic ...
    }
}
```

4.2.2 Illustrations

When the user starts using the software, the `UndoRedoStack` in the `Logic` component will be empty at first.

For example, when the user executes a `delete 5` command, an undoable command, the current state of the address book is saved. The command will then be pushed onto the `undoStack` in the `UndoRedoStack`. The current state of the application is then saved together with the command, which is illustrated in *Figure 14*.

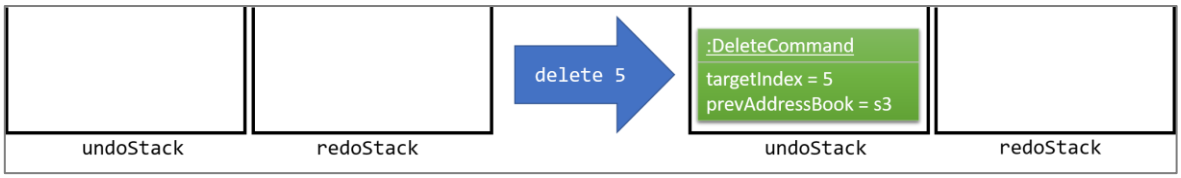

Figure 14: State of `UndoRedoStack` after `delete5` command

As the user continues to execute commands that are undoable, more commands are added into the `undoStack`. For example, the user may execute an `add n/David …` command to add a new person, which is illustrated in *Figure 15*.

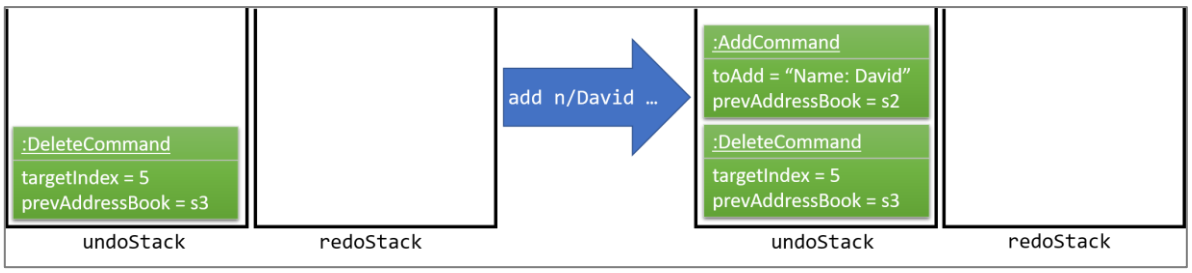

Figure 15: State of `UndoRedoStack` after `add n/David …` command

If the command fails, it will not be pushed to the `undoStack`.

If the user entered an `undo` command, the `undoStack` will pop the most recent command, and push the command into the `redoStack`. The `Model` component will restore to the state before the add `n/David …` command executed, which is illustrated in *Figure 16*.

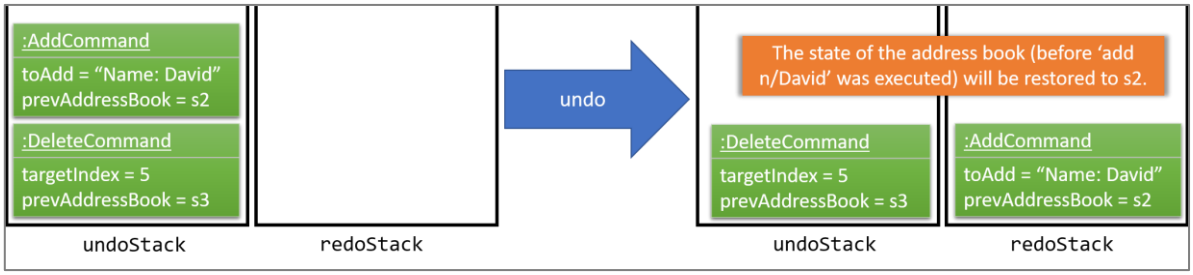

Figure 16: Illustration after `undo` command

If the `undoStack` is empty, then there are no other commands left to be undone. An `Exception` will be thrown when popping the `undoStack`.

The sequence diagram for the `undo` command is illustrated in *Figure 17*.

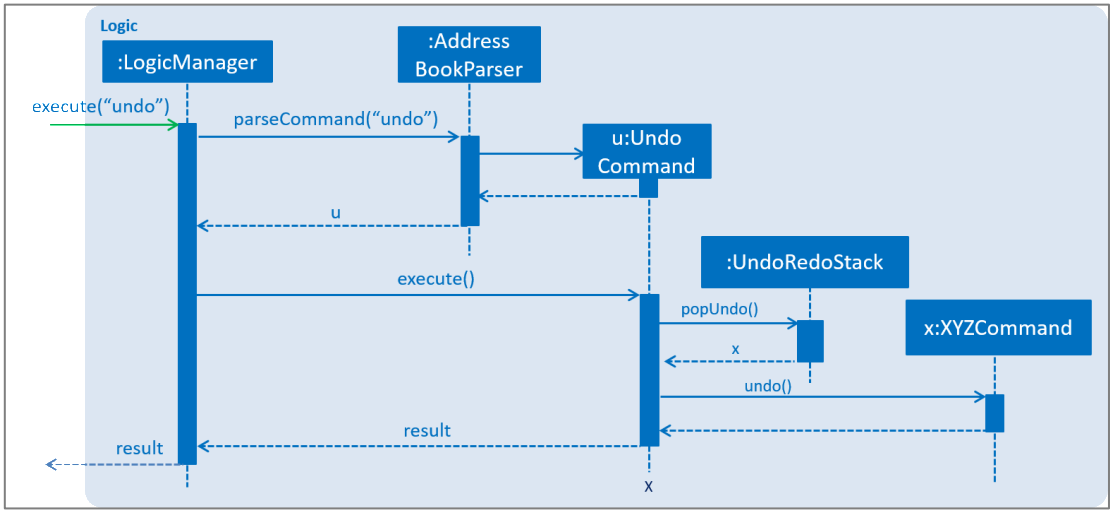

Figure 17: Sequence diagram for `undo` command

Similarly, the `redo` command pops the most recent undone command from `redoStack` and push the command to the `undoStack`. This will also restore the address book to the state after the command is executed.

If the `redoStack` is empty, then there are no other commands left to be redone. An `Exception` will be thrown when popping the `redoStack`.

4.2.3 Design Considerations

Design Aspect: Implementation of `UndoableCommand`

- **Alternative 1 (Current choice):** The implementation makes use of a new abstract method `executeUndoableCommand`.
    - **Pros:** It will not lose any undone/redone functionality as it is now part of the default behaviour.
    - **Cons:** It is hard for new developers to understand the template pattern.
- **Alternative 2:** The implementation overrides the method `execute()` from `Command` class.
    - **Pros:** It does not involve the template pattern, which is easier for new developers to understand.
    - **Cons:** It requires classes that inherit from `UndoableCommand` to call `super.execute()` every single time.

Design Aspect: Execution of `undo` and `redo` commands

- **Alternative 1 (Current choice):** The software saves the entire address book after each command.
    - **Pros:** It is easy to implement.
    - **Cons:** It may have performance issues in terms of memory usage.
- **Alternative 2:** The commands implemented are able to undo or redo by itself using its own methods.
    - **Pros:** It will require less memory.
    - **Cons:** It requires the implementation of each individual command are correct.

Design Aspect: Type of commands that can be undone/redone

- **Alternative 1 (Current choice):** The implementation includes commands that modify the address book (`add`, `clear`, `edit`).
    - **Pros:** It reverts the changes that are hard to change back.
    - **Cons:** It may be possible that user misunderstands `undo` also applies to commands when the list is modified without changing of data.
- **Alternative 2:** The software allows all commands to be undone/redone.
    - **Pros:** It may be more intuitive to the user.
    - **Cons:** It results in difficulty if the user wants to reset application state and not the view.

Design Aspect: Type of data structure to support the commands

- **Alternative 1 (Current choice):** The implementation uses two separate stack for `undo` and `redo`.
    - **Pros:** It is easier to understand for new developers.
    - **Cons:** It duplicates the same logic.
- **Alternative 2:** The implementation uses a `HistoryManager` class for `undo` and `redo`.
    - **Pros:** It is not necessary to maintain a separate stack.
    - **Cons:** It requires dealing with commands that have already been undone, and violates Single Responsibility Principle and Separation of Concerns as `HistoryManager` now needs to do two different tasks.

4.3 Add Appointment Feature

This feature allows user to add appointments to calendars when using the software.

4.3.1 Current Implementation

The external CalendarFX package is used for the calendar implementation and the API for all the CalendarFX classes and methods used for the software can be found here.

The calendar is represented by the `CelebCalendar` class, which extends from the default `Calendar` class in CalendarFX. In this class, it contains appointments using the `Appointment` class extended from the `Entry` class in CalendarFX.

The fields required for the `Appointment` class and its format can be seen in the following code snippet:

```
public class Appointment extends Entry {

    public static final String MESSAGE_NAME_CONSTRAINTS =
            "Appointment names should only contain alphanumeric characters and
             spaces, and it should not be blank"; // used for name and location

    public static final String MESSAGE_TIME_CONSTRAINTS =
            "Time should be a 2 digit number between 00 to 23 followed by a :"
            + " followed by a 2 digit number beetween 00 to 59. Some examples
              include "
            + "08:45, 13:45, 00:30";
    public static final String MESSAGE_DATE_CONSTRAINTS =
            "Date should be a 2 digit number between 01 to 31 followed by a -"
            + " followed by a 2 digit number between 01 to 12 followed by a -"
            + " followed by a 4 digit number describing a year. Some months might
              have less than 31 days."
            + " Some examples include: 13-12-2018, 02-05-2019, 28-02-2018";

    public static final DateTimeFormatter TIME_FORMAT =
            DateTimeFormatter.ofPattern("HH:mm");

    public static final DateTimeFormatter DATE_FORMAT =
            DateTimeFormatter.ofPattern("dd-MM-uuuu")
            .withResolverStyle(ResolverStyle.STRICT); // prevent incorrect dates
    ...
}
```

All `CelebCalendar` instances reside in an instance of the `CalendarSource` class, which is used to store a group of calendars in CalendarFX. The `CalendarSource` instance is then attached to the `CalendarView` class, which represents the calendar GUI.

The `AddAppointment` command adds an appointment to an existing calendar, which is facilitated by the `AddAppointmentCommand` class in the `Logic` component. The appointment can be viewed in the calendar panel if added successfully. This is done by retrieving the list of calendars stored in the `Model` component and adding the appointment to one or more of these calendars.

The `parse` method in the `AddAppointmentCommand` class can create sensible appointments if at least one of the non-compulsory fields are not included, which can be seen in the following code snippet:

```
public AddAppointmentCommand parse(String args) throws ParseException {
        ...
        try {
            String appointmentName =
            ParserUtil.parseGeneralName(argMultiMap.getValue(PREFIX_NAME)).get();
            Optional<LocalTime> startTimeInput = ParserUtil.parseTime(
                    argMultiMap.getValue(PREFIX_START_TIME));
            Optional<LocalDate> startDateInput = ParserUtil.parseDate(
                    argMultiMap.getValue(PREFIX_START_DATE));
            Optional<LocalTime> endTimeInput = ParserUtil.parseTime
                    (argMultiMap.getValue(PREFIX_END_TIME));
            Optional<LocalDate> endDateInput = ParserUtil.parseDate
                    (argMultiMap.getValue(PREFIX_END_DATE));
            Optional<MapAddress> locationInput = ParserUtil.parseMapAddress
                    (argMultiMap.getValue(PREFIX_LOCATION));
            Set<Index> celebrityIndices = ParserUtil.parseIndices
                    (argMultiMap.getAllValues(PREFIX_CELEBRITY));
            Set<Index> pointOfContactIndices = ParserUtil.parseIndices
                    (argMultiMap.getAllValues(PREFIX_POINT_OF_CONTACT));
            ...
            if (startTimeInput.isPresent()) {
                startTime = startTimeInput.get();
                endTime = startTimeInput.get();
            }
            if (endTimeInput.isPresent()) {
                endTime = endTimeInput.get();
            }
            if (startDateInput.isPresent()) {
                startDate = startDateInput.get();
                endDate = startDateInput.get();
            }
            if (endDateInput.isPresent()) {
                endDate = endDateInput.get();
            }
            if (locationInput.isPresent()) {
                location = locationInput.get();
            }
        ...
    }
    ...
}
```

The sequence diagram of an `addAppointment` command can be seen in *Figure 18*.

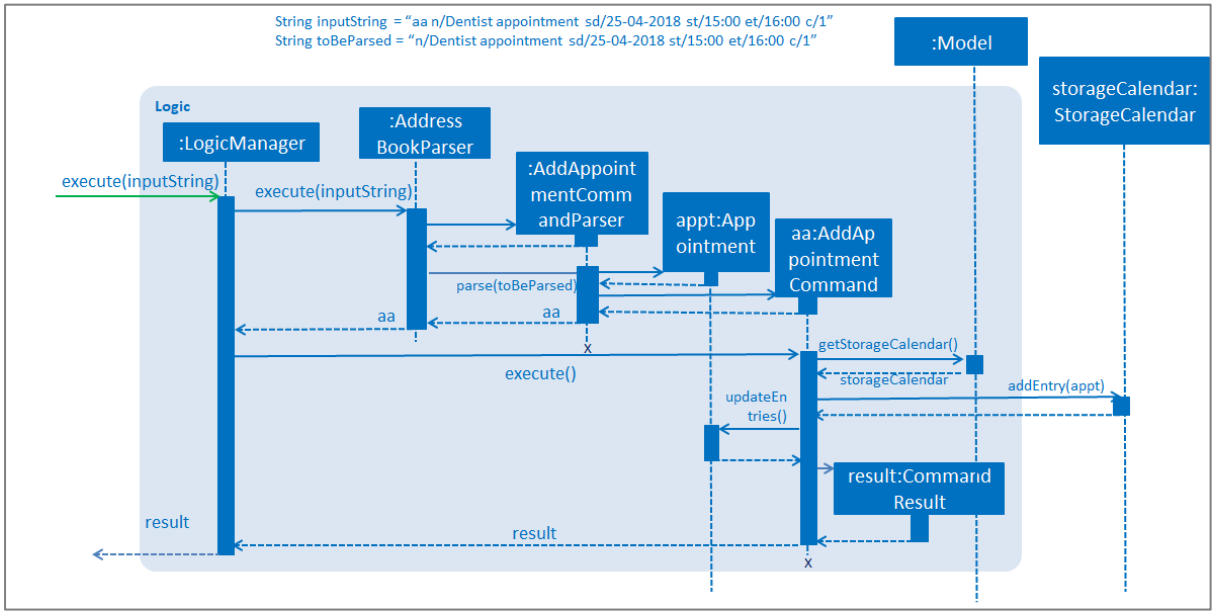

Figure 18: Sequence diagram for `addAppointment` command

The change in calendar panel before and after executing an `addAppointment` command can be seen in *Figure 19* and *Figure 20*.

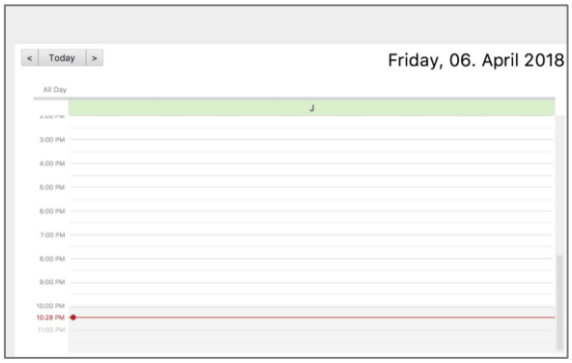

Figure 19: Calendar panel before executing `addAppointment` command

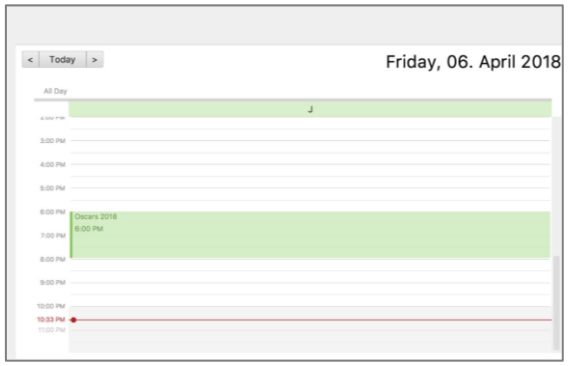

Figure 20: Calendar panel after executing `addAppointment` command

4.3.2 Design Considerations

Design Aspect: Ability to undo `addAppointment` command

- **Alternative 1 (Current choice):** The implementation disallows undoing the command.
    - **Pros:** It does not need to remember previous state of the storage calendar.
    - **Cons:** It is impossible to remove or edit additions made by mistake without looking at the list of appointments.
- **Alternative 2:** The implementation allows undoing the command.
    - **Pros:** It allows removing additions made by mistake.
    - **Cons:** It requires a drastic change in the way calendars are currently saved and loaded, as calendars are part of the `UI` component while appointments are part of the `Model` component.

4.4 Delete Appointment Feature

This feature allows user to delete appointments from the calendars in the application.

4.4.1 Current Implementation

The feature is facilitated by the `DeleteAppointmentCommand` class in the `Logic` component, and the `deleteAppointment` method in the `Model` component. Deletion of an appointment can only be done when the calendar panel is showing the list of appointments saved. The command requires the user to put in an index which is taken from the currently displayed appointment list.

In the implementation, the appointment list will reflect the change in the calendar panel after deleting a specified appointment. In the event when there are no more appointments in the appointment list, the calendar panel will show the calendars instead.

The implementation of the `deleteAppointment` method in `Model` component is shown in the following code snippet:

```
public class ModelManager extends ComponentManager implements Model {
    ...
    @Override
    public Appointment deleteAppointment(int index) throws CommandException {
        Appointment apptToDelete = getChosenAppointment(index);
        apptToDelete.removeAppointment();
        indicateAppointmentListChanged();

        apptToDelete = removeAppointmentFromInternalList(index);

        if (getAppointmentList().size() < 1) {
            setIsListingAppointments(false);
            setCelebCalendarViewPage(DAY_VIEW_PAGE);
        }
        return apptToDelete;
    }

    /** Makes changes to model's internal appointment list */
    private Appointment removeAppointmentFromInternalList(int index) {
        return getAppointmentList().remove(index);

    }
    ...
}
```

The sequence diagram of the `deleteAppointment` command can be seen in *Figure 21*.

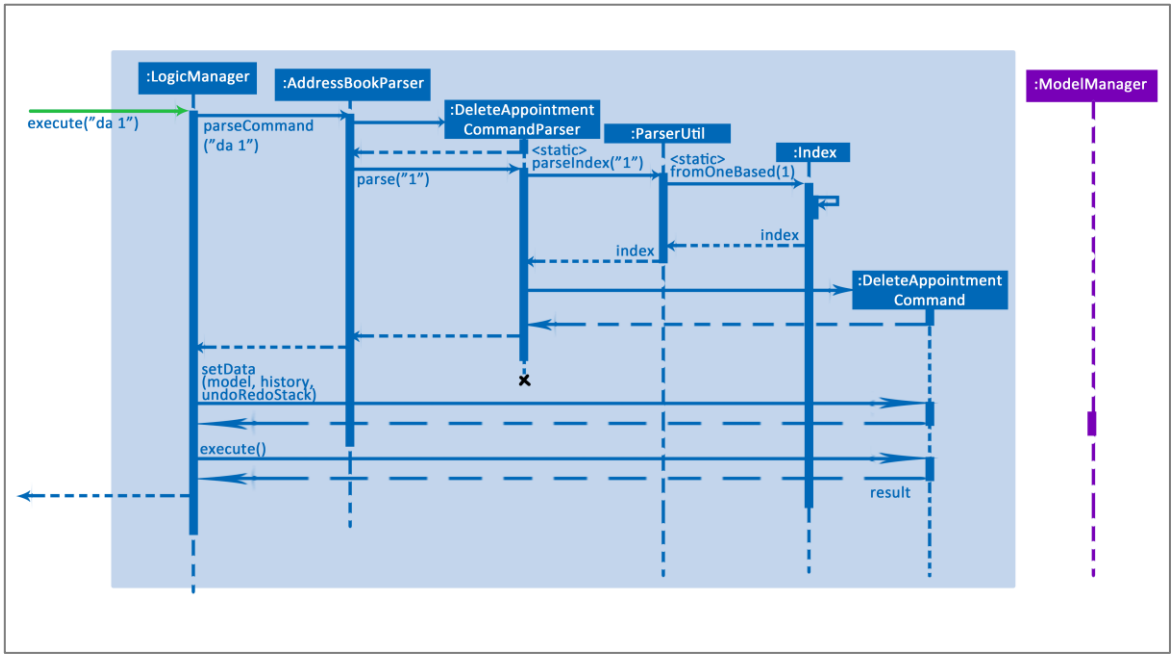

Figure 21: Sequence diagram for `deleteAppointment` command

The change in state of the calendar panel after executing `deleteAppointment` command can be seen in *Figure 22* and *Figure 23*.

Figure 22: Calendar panel before executing `deleteAppointment` command

Figure 23: Calendar panel after executing `deleteAppointment` command

4.4.2 Design Considerations

Design Aspect: State of calendar panel after deletion of the only appointment

- **Alternative 1 (Current choice):** The calendar panel switches to the combined calendar view.
    - **Pros:** It is consistent with `listAppointment` as CelebManager does not show an empty list when there is no appointment to list, but instead outputs an error message.
    - **Cons:** It is difficult for the user to see if the appointment gets deleted correctly.
- **Alternative 2:** The calendar panel remains in the appointment list view and shows an empty list.
    - **Pros:** It shows the effect of deletion immediately.
    - **Cons:** It is inconsistent with `listAppointment` command's inability to show an empty list when there is no appointment to list.
- **Alternative 3:** The calendar panel swtiches to the combined calendar view with reference to the deleted appointment's start date.
    - **Pros:** It is consistent with `listAppointment` while making it easy for users to check if the appointment gets deleted visually on calendar.
    - **Cons:** It will takes a long time to run.

Design Aspect: Ability to undo `deleteAppointment` command

- **Alternative 1 (Current choice):** The implementation does not allow the undoing of the command.
    - **Pros:** It does not need to remember previous appointments' and calendar's status.
    - **Cons:** It is impossible to restore deletions made by mistake.
- **Alternative 2:** The implementation allows the undoing of the command.
    - **Pros:** It allows restoring of deletions made by mistake.
    - **Cons:** It requires a drastic change in the way calendars are currently saved and loaded, as calendars are part of the `UI` component while appointments are part of the `Model` component.

4.5 View Appointment Feature

This feature allows user to view a specific appointment in the calendar panel.

4.5.1 Current Implementation

The feature is facilitated by the `ViewAppointmentCommand` class in the `Logic` component. It supports viewing of a specific appointment in the calendar panel by displaying the appointment details. The specific appointment is selected using an index based on the list generated by the `listAppointment` command.

As this feature relies on the list generated by the `listAppointment` command, it retrieves appointments that will happen within the earliest start date and latest end date. The `getChosenAppointment` method is used to generate these appointments internally from the `StorageCalendar` class in the `Model` component.

The implementation of retrieving selected appointment is shown in the following code snippet:

```
public CommandResult execute() throws CommandException {
        selectedAppointment = model.getChosenAppointment(chosenIndex);
        try {
            ShowLocationCommand showLocation = new ShowLocationCommand(
                    new MapAddress(selectedAppointment.getLocation()));
            showLocation.execute();
            return new CommandResult(MESSAGE_SUCCESS +
                    getAppointmentDetailsResult());
        } catch (NullPointerException npe) {
            return new CommandResult(MESSAGE_SUCCESS +
                    getAppointmentDetailsResult());
        }
    }
```

The sequence diagram of the `ViewAppointmentCommand` method is illustrated in *Figure 24*.

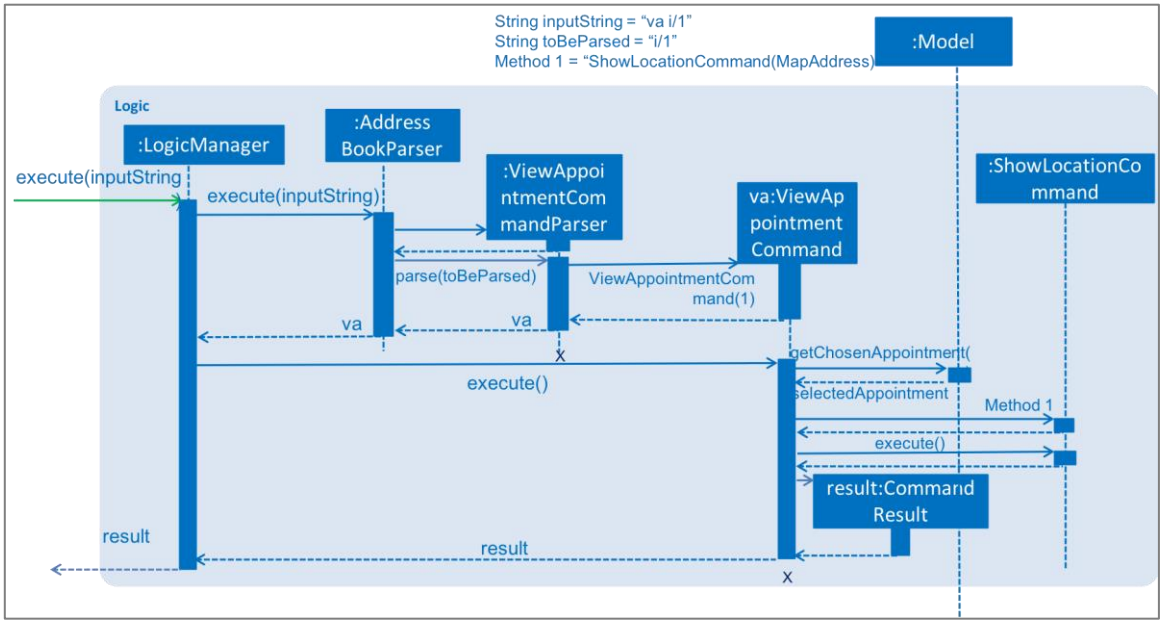

Figure 24: Sequence diagram for the `ViewAppointmentCommand` method

4.5.2 Design Considerations

Design Aspect: Implementation of `viewAppointmentCommand` class

- **Alternative 1 (Current choice):** The class extends the `Command` class.
  - **Pros:** It is easier to understand for new developers who will be developing this project as the command is at the same abstraction level as other commands.
  - **Cons:** It does not have the undo/redo feature as it is not part of the `UndoableCommand` class
- **Alternative 2:** The class extends the `UndoableCommand` class.
  - **Pros:** It allows for the command to be undone/redone.
  - **Cons:** It requires more work that may not fit into development timeline.

Design Aspect: Inclusion of showing location in the map panel

- **Alternative 1 (Current choice):** The map panel shows the location of the appointment.
  - **Pros:** It reduces the hassle of keying an extra command.
  - **Cons:** It reduces independent usage of `ShowLocationCommand`.
- **Alternative 2:** The map panel does not show the location of the appointment.
  - **Pros:** It allows restoring of deletions made by mistake.
  - **Cons:** It requires an extra command input to show location when required.

4.6 Store Appointment Feature

This feature allows user to save and read appointments from the hard disk automatically after adding, editing and deleting appointments in the address book.

4.6.1 Current Implementation

The storing of appointment is facilitated by the `XmlStorageCalendarStorage` class in the `Storage` component. It supports the retrieval and storage for appointments made by the user.

During start-up of application, the storage component will be initialized by the `MainApp` class, which retrieves information from the specified file path in the `UserPrefs` file. The storage component will be initialized by the `MainApp` class, as seen in the following code snippet:

```
public void init() throws Exception {
        // initializes application.

        UserPrefsStorage userPrefsStorage = new
                JsonUserPrefsStorage(config.getUserPrefsFilePath());
        userPrefs = initPrefs(userPrefsStorage);
        AddressBookStorage addressBookStorage = new
                XmlAddressBookStorage(userPrefs.getAddressBookFilePath());
        StorageCalendarStorage storageCalendarStorage =
                new XmlStorageCalendarStorage
                        (userPrefs.getStorageCalendarFilePath());
        storage = new StorageManager(addressBookStorage, userPrefsStorage,
                storageCalendarStorage);

        // initializes other component in the application.
    }
```

While the `XmlStorageCalendarStorage` class allows access to data stored on the hard disk, the `XmlSerializableStorageCalendar` class represents the data of the appointment list for the calendar. In `XmlSerializableStorageCalendar`, it contains a `List` of XML formats of appointments `XmlAdaptedAppointment`. `XmlAdaptedAppointment` will then contain essential information of different `Appointment` in `StorageCalendar` of the `Model` component.

In the `XmlStorageCalendarStorage` class, it allows the use of `readStorageCalendar` method to retrieve a `StorageCalendar` instance and save it using the `saveStorageCalendar` method. The reading is done by checking if the file exist, and loading the list of appointments from a `XmlSerializableStorageCalendar` instance.

The class also allows writing of information into `filePath` specified in `userPrefs` file. The writing is done by creating a new file and rewriting to the list in the same `XmlSerializableStorageCalendar` instance.

4.6.2 Design Considerations

Design Aspect: Usage of data structures for saving appointments

- **Alternative 1 (Current choice):** The implementation uses a single `List` to store appointments.
    - **Pros:** It allows simplicity.
    - **Cons:** It slows the application if there are too many appointments.
- **Alternative 2:** The implementation uses a single `Set` such as `TreeSet`.
    - **Pros:** It lowers impact in speed when there are many appointments.
    - **Cons:** It complicates implementation when speed is not an issue.

4.7 Show Location Feature

This feature allows user to view a location in the map panel of the software.

4.7.1 Current Implementation

The feature is facilitated by the `ShowLocationCommand` class in the `Logic` component. It supports the viewing of location in the map panel by updating its state. This is done by centering the interactive map using GMAPSFX API, with the latitude and longitude of the location generated from Google Maps Web Services API. The location is then identified with a location marker shown in *Figure 25*.

Figure 25: Location marker used in the map panel

The inheritance diagram for the `ShowLocationCommand` class is shown in *Figure 26*.

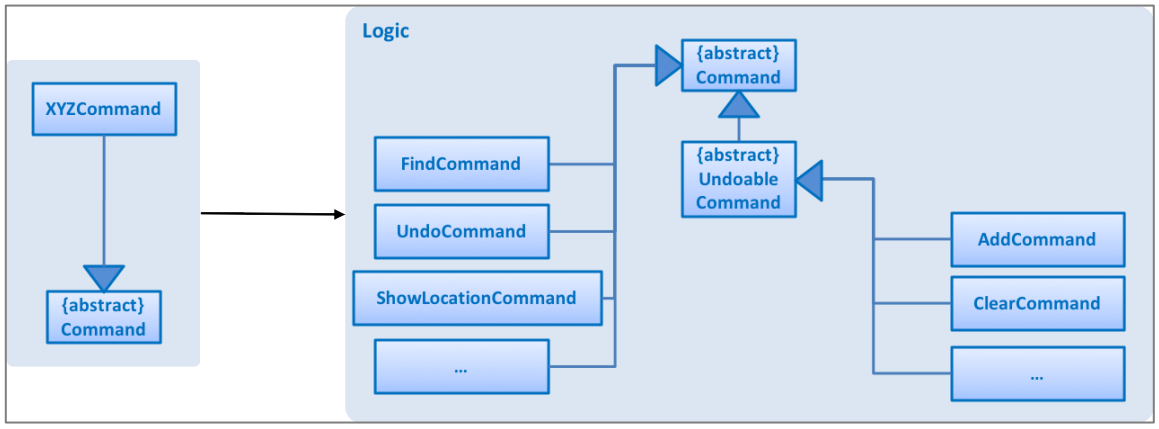

Figure 26: Inheritance diagram for `showLocationCommand` class

The feature uses the `MapAddress` class, which checks for blank space and the validity of location in Google server. The implementation is shown in the following code snippet:

```
public class MapAddress {
    public static final String MESSAGE_ADDRESS_MAP_CONSTRAINTS =
            "Address should be in location name, road name, block and road name or
postal code format.\n"
                    + "Note:(Person address may not be valid as it consist of too
many details like unit number)"
    /*
     * The first character of the address must not be a whitespace,
     * otherwise " " (a blank string) becomes a valid input.
     */
    public static final String ADDRESS_VALIDATION_REGEX = "[^\\s].*";
    ...
    /**
     * Returns true if a given string is a valid map address.
     */
    public static boolean isValidAddress(String test) {
        boolean isValid;
        Geocoding testAddress = new Geocoding();
        isValid = testAddress.checkIfAddressCanBeFound(test);
        return test.matches(ADDRESS_MAP_VALIDATION_REGEX) && isValid;
    }
    ...
}
```

The sequence diagram for the `showLocation` command can be seen in *Figure 27*.

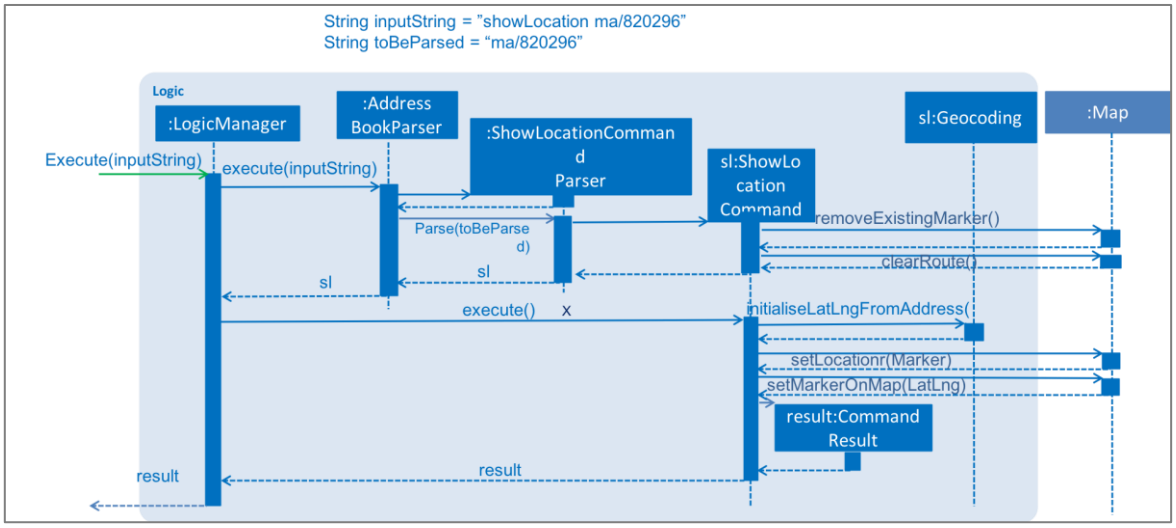

Figure 27: Sequence diagram for `showLocation` command

Every new `showLocation` command, valid or invalid, will remove the previous route or location marker and add the new marker into the map, which can be seen in the difference between map panels in *Figure 28* and *Figure 29*.

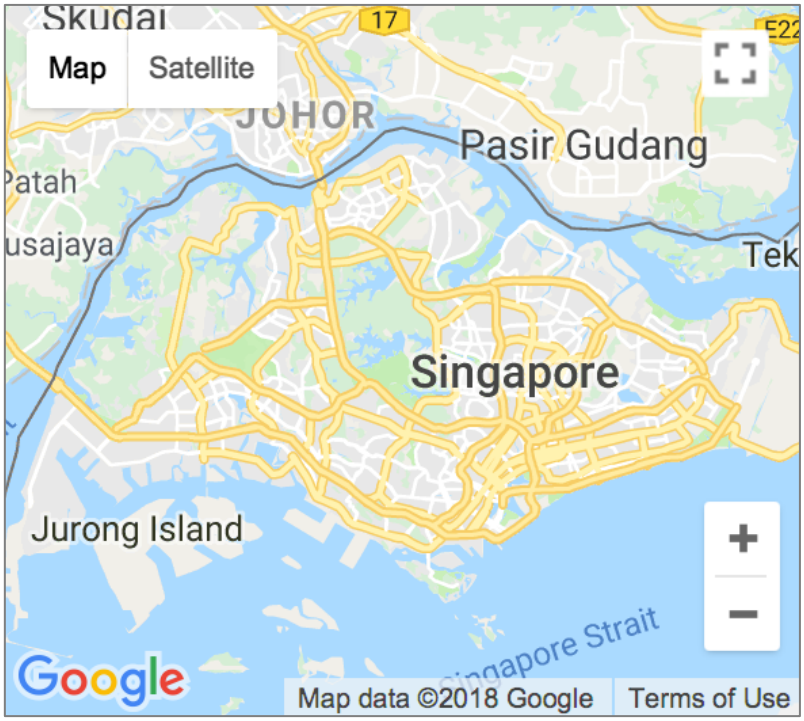

Figure 28: Map panel before executing `showLocation` command

Figure 29: Map panel after executing `showLocation` command

4.7.2 Design Considerations

Design Aspect: Implementation of `showLocationCommand` class

- **Alternative 1 (Current choice):** The class extends the `Command` class.
    - **Pros:** It is easier to understand for new developers who will be developing this project as the command is at the same abstraction level as other commands.
    - **Cons:** It does not have the undo/redo feature as it is not part of `UndoableCommand`.
- **Alternative 2:** The class extends the `UndoableCommand` class.
    - **Pros:** It allows for the command to be undone/redone.
    - **Cons:** It requires more work that may not fit into development timeline.

Design Aspect: Type of address class used

- **Alternative 1 (Current choice):** The implementation uses the `MapAddress` class.
    - **Pros:** It allows the clear distinction of requirements.
    - **Cons:** It is confusing as both models are similar.
- **Alternative 2:** The implementation uses the `Address` class.
    - **Pros:** It reduces the amount of code/class in the project.
    - **Cons:** It lacks proper organisation and confuses developers due to different requirements in models.

4.8 Estimate Route Feature

This feature allows user to estimate route between two locations in the map panel of the software.

4.8.1 Current Implementation

The feature is facilitated by the `EstimateRouteCommand` class in the `Logic` component. It supports the viewing of estimated route in the map panel by updating the state of the `MapPanel`. This is done by centering the interactive map using GMAPSFX API, with the latitude and longitude of the location generated from Google Maps Web Services API. *Figure 30* and *Figure 31* shows the marker that is used to identify the start and end location in the map panel.

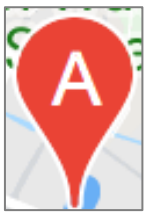

Figure 30: Start location marker used in the map panel

Figure 31: End location marker used in the map panel

In addition, the Google Maps Web Services API is also used to create the `DistanceEstimate` class, which calculates the estimated time and distance of travel between two locations by driving and check if two locations can be reached by driving.

The sequence diagram for `estimateRoute` command can be seen in *Figure 32*.

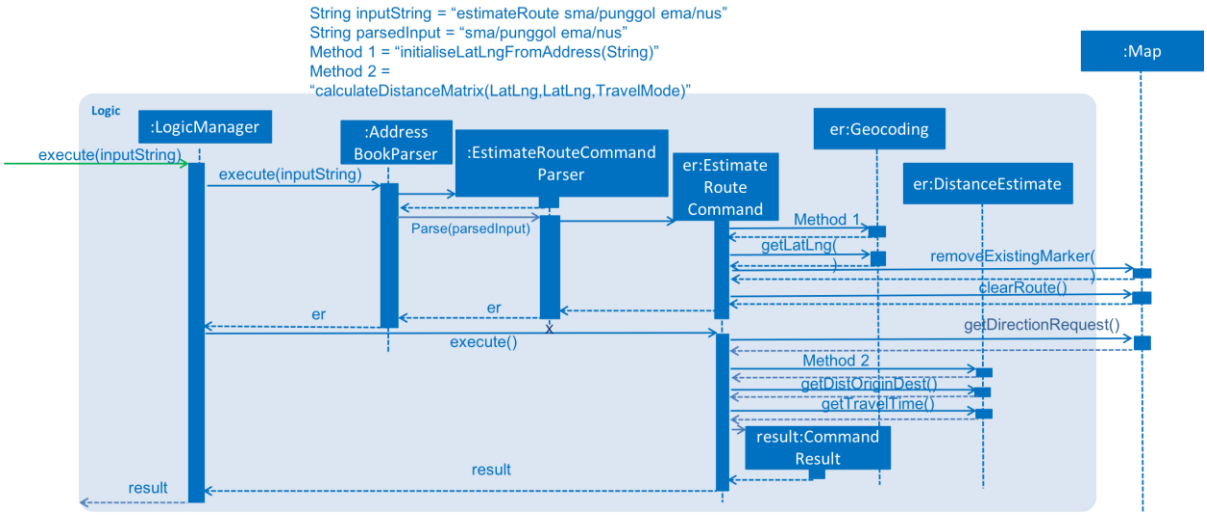

Figure 32: Sequence diagram for `estimateRoute` command

Every new `estimateRoute` command, valid or invalid, will remove the previous route or location marker and add the new marker into the map, which can be seen in the difference between map panels in *Figure 28* and *Figure 29*.

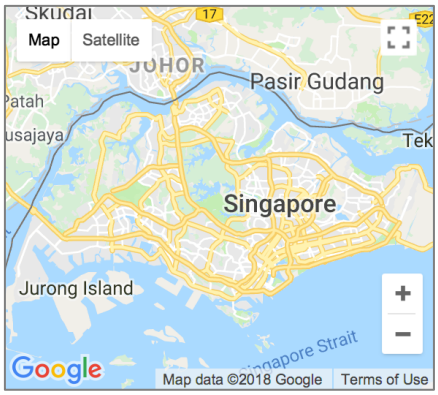

Figure 33: Map panel before executing `estimateRouteCommand`

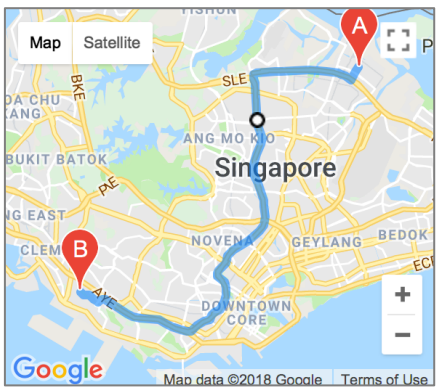

Figure 34: Map panel after executing `estimateRouteCommand`

4.8.2 Design Considerations

Design Aspect: Implementation of `estimateRouteCommand`

- **Alternative 1 (Current choice):** The class extends the `Command` class.
    - **Pros:** It is easier to understand for new developers who will be developing this project as the command is at the same abstraction level as other commands.
    - **Cons:** It does not have the undo/redo feature as it is not part of `UndoableCommand`.
- **Alternative 2:** The class extends the `UndoableCommand` class.
    - **Pros:** It allows for the command to be undone/redone.
    - **Cons:** It requires more work that may not fit into development timeline.

Design Aspect: Type of address class used

- **Alternative 1 (Current choice):** The implementation uses the `MapAddress` class.
    - **Pros:** It allows the clear distinction of requirements.
    - **Cons:** It is confusing as both models are similar.
- **Alternative 2:** The implementation uses the `Address` class.
    - **Pros:** It reduces the amount of code/class in the project.
    - **Cons:** It lacks proper organisation and confuses developers due to different requirements in models.

Design Aspect: Type of input from appointments

- **Alternative 1 (Current choice):** The implementation makes use of `Location` value in appointments.
    - **Pros:** It allows the function to be used independently.
    - **Cons:** It requires keying in of location instead of just an index.
- **Alternative 2:** The implementation makes use of the unique index of `Appointment`.
    - **Pros:** It reduces the amount of typing.
    - **Cons:** It restricts the use of the command, as it is unusable without an appointment index.

5 Documentation

This section describes how to document your project efficiently.

5.1 Editing of Documentation

We will be using AsciiDoc, a lightweight markup language, for writing documentation.

Please refer to UsingGradle documentation for instructions on 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.

AsciiDoc, a markup language, is chosen over markdown language as it provides more flexibility with regards to formatting.

5.2 Publishing of Documentation

Please refer to UsingTravis documentation for instructions on how to deploy GitHub pages using Travis.

5.3 Conversion of Documentation

You can use Google Chrome to convert documents to `.pdf` format, as Chrome's PDF engine preserves hyperlinks used in web pages.

To convert your project documentation files to `.pdf` format:

1. Follow the instructions in UsingGradle documentation to convert the `.adoc` files in `docs/` directory to `.html` format.
2. Look for the generated `.html` files in `build/docs/` directory.
3. Right-click on the `.html` files and select `Open with` -> `Google Chrome`.
4. Click on the `Print` option in Google Chrome's menu.

   After completion of Step 4, the print menu will appear similar to *Figure 35*.
5. Set the destination to `Save as PDF`, and click `Save` to save a copy of the file in `.pdf` format.

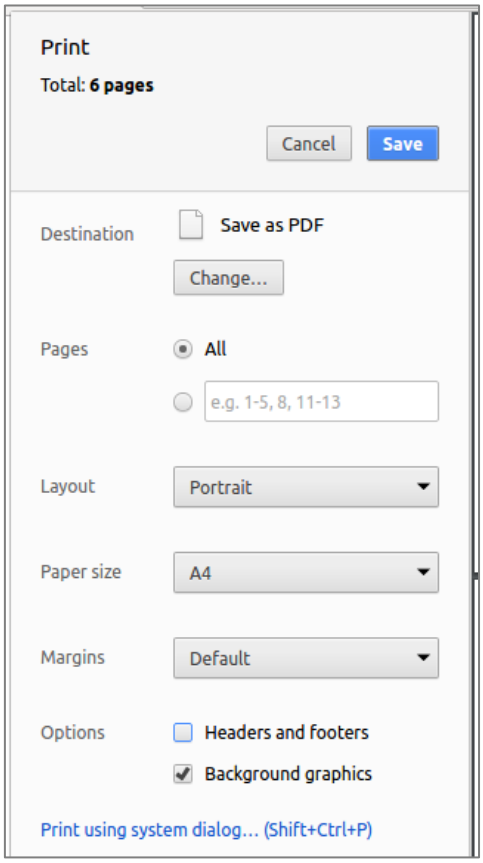

Figure 35: Print menu on Google Chrome

For the best results, please use the settings indicated in *Figure 35*.

6 Testing

This section describes how to conduct testing for your project.

Testing is very important in software development as it allows us to find application defects that were made during development. It should be done constantly as it can be expensive if software testing is done only in the later stages of development when a bug may affect different components of the project.

6.1 Types of Test

There are two types of tests that we can run during the development of the project.

1. GUI Tests

   These are tests involving the GUI. They include:

   a. System tests, which test the entire application by simulating user actions on the GUI, and are in the `systemtests` package.

   b. Unit tests, which test the individual components of the software, and are in `seedu.address.ui` package.

2. Non-GUI Tests

   These are tests not involving the GUI. They include:

   a. Unit tests, which test the lowest leve methods and classes, such as `seedu.address.commons.StringUtilTest`

   b. Integration tests, which test the integration of multiple code units assuming that the units are working perfectly, such as `seedu.address.storage.StorageManagerTest`

   c. Hybrids of unit and integration tests, which test multiple code units as well as the cohesiveness of the code units, such as `seedu.address.logic.LogicManagerTest`

6.2 Testing Methods

There are three ways to test the application.

- Method 1: Using Gradle (headless)

  Open a console and run the command `gradlew clean headless allTests` for Windows users, or `./gradlew clean headless allTests` for Mac/Linux users.

> GUI tests can be run in headless mode due to the TestFX library. GUI tests do not show up on the screen in headless mode, which allows you to work on other matters while tests are running.
>
> Using Gradle (headless) is the most reliable way to run tests. Other testing methods may fail some GUI tests due to platform/resolution-specific idiosyncrasies.
>
> Refer to UsingGradle documentation for more information on how to run tests using Gradle.

- Method 2: Using Gradle

  Open a console and run the command `gradlew clean allTests` (for Windows), or `./gradlew clean allTests` (for Mac/Linux).

- Method 3: Using IntelliJ Junit test runner
  - Right-click on the `src/test/java` folder and choose `Run 'All Tests'`, or;
  - Right-click on a test package or a test class, and choose `Run 'Tests in '<test package or test class>''`.

> If you run into `HelpWindowTest` failing with a `NullPointerException`, it is due to one dependencies `src/main/resources/docs/UserGuide.html` missing.
> You can execute `gradlew processResources` (for Windows), or `./gradlew processResources` (for Mac/Linux) to solve this issue.

7 DevOps

This section describes how to unify software development and software operation.

DevOps refers to a software engineering culture and practice that encourages automation and monitoring of software construction.

7.1 Build Automation

We will be using Gradle for build automation.

Please refer to Section 2.2: Local Project Setup and UsingGradle documentation for more information.

7.2 Continuous Integration

We will be using Travis CI and AppVeyor to perform Continuous Integration.

Please refer to Section 2.4.3: Continuous Integration (CI) Configurations, UsingTravis documentation and UsingAppVeyor documentation for more information.

7.3 Coverage Reporting

We will be using Coveralls to track the code coverage.

Please refer to Section 2.4.4: Coverage Reporting Configurations and UsingCoveralls documentation for more information.

7.4 Documentation Preview

We will be using Netlify to see a preview how HTML version of asciidoc files will look like after a pull request is merged.

Please refer to UsingNetlify documentation for more information.

7.5 Application Release

To create a new release of the application, you would need to:

1. Update the version number in `MainApp.java`,
2. Generate a `.jar` file using Gradle,
3. Tag the repository with the version number,
4. Create a new release using GitHub, and;
5. Upload the `.jar` file created in Step 2.

Refer to UsingGradle documentation for more information on how to generate `.jar` files using Gradle.

Refer to GitHub documentation for more information on how to create releases.

7.6 Dependencies Management

Management of dependencies on third-party libraries is done using Gradle. There is no need to include those libraries in the repository or download them manually.

Appendix A: Product Scope

The target users:

- Need to manage a significant number of contacts.
- Prefer desktop applications over applications on other platforms.
- Can type fast.
- Prefers typing over mouse input.
- Are resonably comfortable using CLI applications.
- Need to manage several (celebrities') schedules.
- Need to link contacts to appointments.

The product should be able to manage contacts and appointments faster than a typical mouse-driven/GUI-driven application.

Appendix B: Feature Contribution

Major Features

- Adding, deleting and editing appointments (By Muruges)
    - User can create, delete and edit appointments within the application.
- Listing appointments (By Muruges)
    - User can list appointments within a date range.
- Showing location and routes on map (By Damien)
    - User can see the location of an appointment using address in maps.
    - User can generate a rough route that is used to calculate the distance and time of travel.
    - User can generate the estimated distance and time of travel between two locations.
- Storing appointments (By Tzer Bin)
    - User can automatically save their appointments when running commands.
- Switching between calendar views (By Jinyi)
    - User are able to switch calendar views by day, week and month.

Minor Features

- Adding celebrities and point-of-contacts (by Muruges)
    - User can add celebrities and point-of-contacts to appointments.
- Removing of tags (by Jinyi)
    - User can remove tags from the application.
- Viewing of appointments (by Damien)
    - User can view a specific appointment in result display based on index.

Appendix C: User Stories

Priorities:

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

Priority	As a …	I want to …	So that I can …
`***`	new user	see usage instructions	refer to instructions when i forget how to use the app
`***`	user	add a new person	access contacting information of the person from the application
`***`	user	delete a person	remove contacts that i no longer need
`***`	user	find a contact by name	locate details of contacts without having to go through the entire list
`***`	user	undo a previous command	remove the change made by mistake
`***`	user	redo a previos command	restore the change removed by mistake
`***`	user	have a calendar inside the address book	know the date and day
`***`	user	display appointments on calendar by day, week and month	check appointments in different time frames
`***`	user	add an appointment to calendar	schedule different appointments without time clashes
`***`	user	delete an appointment from a calendar	remove appointments that are cancelled
`***`	user	edit and appointment in a calendar	change the information about the appointment when there is a change of plan or arrangement

* * *	user	save appointments	get appointments loaded in the calendar automatically when i re-launch the application
* * *	user	get alerted for upcoming appointments	set my priorities straight
* * *	user	see the location of an appointment in a map	plan for travelling
* * *	user	see various landmarks around a specific location in a road map	understand better the roads around the location
* * *	user	see various landmarks around a specific location in a satellite map	see in real-time the actual layout of the surrounding
* * *	user	zoom in and out of a location in a map	view the location in different levels of details
* * *	celebrity managers	see the best route of travel by driving between two locations in a map	plan for the shortest travel
* * *	celebrity managers	know if two locations can be reached by driving	foresee any problems and plan ahead
* * *	celebrity managers	know the estimated distance between two locations by drivin	cater enough time for travelling to avoid being late
* * *	celebrity managers	know the estimated time of travel between two locations by driving	reduce the time of travel to reach an appointment location

* * *	celebrity managers	have multiple calendars to display appointments for different celebrities	manage multiple celebrities' appointments
* *	user	hide private contact details by default	minimize change of somone else seeing them by accident
* *	user	change the size of different window of the application	customize the window sizes
* *	user who contacts different parties involved in an appointment	draft the message about appointment details automatically	save time to draft the email
* *	user who frequently contacts certain people	sort the contacts by contacting frequency	find those people i frequently contact easily
* *	user	change the colour scheme of the application	choose my preferred colour scheme
* *	celebrity manager	group celebrities by different talents	find celebrities by talent easily
* *	celebrity manager managing celebrities of the same group	add an appointment to the calendars of these celebrities at the same time	save time

*	user with many international contacts	group contacts by country code	see my contacts from different countries
*	user of previous versions of the application	transfer my contacts to the new version	save the trouble of adding the contacts again
*	user with poor eyesight	have the address book to read out the contacts to me	use it more efficiently
*	user	output the contacts to a separate list	havae a backup copy of the contacts

Appendix D: Use Cases

For all use cases, the **System** refers to `CelebManager` and the **Actor** is the `user`, unless otherwise specified.

D.1 Use case: Delete person

MSS

1. User requests to list persons.
2. CelebManager shows a list of persons.
3. User requests to delete a specific person in the list.
4. CelebManager deletes the person.

   Use case ends.

Extensions

- 2a. The list is empty.

  Use case ends.
- 3a. The given index is invalid.
  - 3a1. CelebManager shows an error message

    Use case resumes at Step 2.

D.2 Use case: Undo

MSS

1. User requests to undo.
2. CelebManager undoes the latest executed command that mutates the data.

   Use case ends.

Extensions

- 2a. User requests to undo.
  - 2a1. CelebManager undoes the latest executed command that mutates the data.

    Use case ends.

D.3 Use case: Redo

MSS

1. User requests to redo.
2. CelebManager redoes the latest executed undo command.

   Use case ends.

Extensions

- 2a. There is no executed undo command.
    - 2a1. CelebManager shows an error message.

      Use case ends.

D.4 Use case: Remove tag

MSS

1. User requests to remove a tag.
2. CelebManager removes the tag from any person having it.

   Use case ends.

Extensions

- 1a. The tag does not exist.
    - 1a1. CelebManager shows an error message.

      Use case ends.

D.5 Use case: View appointment

MSS

1. User requests to view appointment.
2. CelebManager shows the appointment's details.

   Use case ends.

Extensions

- 1a. Appointment to view does not exist.
    - 1a1. CelebManager shows an error message.

      Use case ends.

D.6 Use case: Add appointment

MSS

1. User requests to add an appointment.
2. CelebManager adds the appointment to the currently displayed calendar.

   Use case ends.

Extensions

- 1a. Appointment to add has incorrect details or format.
    - 1a1. CelebManager shows an error message.

      Use case ends.
- 2a. Appointment to add clashes with existing appointment.
    - 2a1. CelebManager shows an error message.

      Use case ends.

D.7 Use case: List appointment

MSS

1. User requests to list appointments from a start date to an end date.
2. CelebManager shows a list of appointments within the date range (inclusive).

   Use case ends.

Extensions

- 1a. The dates are invalid or in wrong format.
    - 1a1. CelebManager outputs an error message.

      Use case ends.
- 2a. There is no appointment to show in the date range.
    - 2a1. CelebManager outputs a message that says no appointment in the specified date range.

      Use case ends.

D.8 Use case: Delete appointment

MSS

1. User requests to list appointments from a start date to an end date.
2. CelebManager shows a list of appointments within the date range (inclusive).
3. User requests to delete a specific appointment in the list.
4. CelebManager deletes the appointment.

   Use case ends.

Extensions

- 1a. The dates are invalid or in wrong format.
  - 1a1. CelebManager outputs an error message.

    Use case ends.
- 2a. There is no appointment to show in the date range.
  - 2a1. CelebManager outputs a message that says no appointment in the specified date range.

    Use case ends.
- 3a. The given index is invalid.
  - 3a1. CelebManager shows an error message.

    Use case resumes at step 2.

D.9 Use case: Edit appointment

MSS

1. User requests to list appointments from a start date to an end date.
2. CelebManager shows a list of appointments within the date range (inclusive).
3. User requests to edit a specified appointment.
4. CelebManager changes appointment details and displays new appointment details to user.

   Use case ends.

Extensions

- 1a. The dates are invalid or in wrong format.
  - 1a1. CelebManager outputs an error message.

    Use case ends.
- 2a. There is no appointment to show in the date range.
  - 2a1. CelebManager outputs a message that says no appointment in the specified date range.

    Use case ends.
- 3a. The given index is invalid.
  - 3a1. CelebManager shows an error message.

    Use case resumes at step 2.
- 3b. Information entered for edit is invalid.
  - 3b1. CelebManager shows an error message.

    Use case resumes at step 2.

D.10 Use case: Show location on map

MSS

1. User inputs location name or address.
2. CelebManager converts information into LatLong form.
3. Celeb Manager uses the LatLong info to update create a new location marker.
4. CelebManager updates the map with the location marker and re-centre its panel view.

   Use case ends.

Extensions

- 1a. User provides invalid input.
    - 1a1. CelebManager requests User to provide valid input.
    - 1a2. User enters new input.

      Steps 1a1-1a2 are repeated until input is valid.

      Use case resumes from step 2.
- 4a. When there is an existing marker in the map.
    - 4a1. CelebManager removes it.

      Use case ends.

D.11 Use case: Show estimated route by driving on map

MSS

1. User inputs location name or address.
2. CelebManager converts information into LatLong form.
3. Celeb Manager uses the LatLong info to update create a new location marker.
4. CelebManager updates the map with the location marker and re-centre its panel view.

   Use case ends.

Extensions

- 1a. User provides invalid input.
    - 1a1. CelebManager requests User to provide valid input.
    - 1a2. User enters new input.

      Steps 1a1-1a2 are repeated until input is valid.

      Use case resumes from step 2.
- 4a. When there is an existing marker in the map.
    - 4a1. CelebManager removes it.

      Use case ends.

D.12 Use case: Show estimated distance and time of travel by driving

MSS

1. User inputs start and end location name or address.
2. CelebManager converts information into LatLong form.
3. Celeb Manager uses the LatLong info to generate the distance and time required to travel.
4. CelebManager shows the information.

   Use case ends.

Extensions

- 1a. User provides invalid input.
    - 1a1. CelebManager requests User to provide valid input.
    - 1a2. User enters new input.

      Steps 1a1-1a2 are repeated until input is valid.

      Use case resumes from step 2.
- 3a. When both location cannot be reached by driving
    - 3a1. CelebManager shows error message.

      Use case ends.

Appendix E: Non-functional Requirements

The application should:

- work on any mainstream OS as long as it has Java 1.8.0_60 or higher installed.
- be able to hold up to 1000 persons without a noticeable sluggishness in performance for typical usage.
- able to accomplish most of the tasks faster using commands than using the mouse.
- be usable by people with no knowledge about command line input.
- respond to any user command within 10 seconds.
- be backward compatible with data produced by earlier versions of the CelebManager.
- come with automated unit tests and open source code.
- favor DOS style commands over Unix-style commands.

Appendix F: Manual Testing

This appendix will discuss on how to test the application manually.

The tests written in this appendix only provie a starting point for testers to work on. Testers are expected to do more exploratory testing.

F.1 Launch

For each subsequent launch, window size and location should be the same as the settings before closing the application from previous launch.

1. Launch the application for the first time.
    a. Download the `.jar` file.
    b. Copy the `.jar` file into an empty folder.
    c. Double-click on the `.jar` file.
    Expected behaviour: CelebManager shows the GUI with a set of sample contacts.
    d. Resize the window to an optimum size.
    e. Move the window to a different location.
    f. Close the window.
2. Launch the application for the second time.
    a. Launch the application by double-clicking on the `.jar` file.
    Expected behaviour: CelebManager is launched with the most recent window size and locaton from the first launch.

F.2 Deleting a contact

For each deletion, the contact should be removed from the application.

Pre-requisites: List all contacts by using the list command, and there should be multiple contacts.

1. Enter `delete 1` in command box.

   Expected behaviour: First contact is deleted from the list. The details of the deleted contact shown in the status message. The timestamp in the status bar is updated.

2. Enter `delete 0` in command box.

   Expected behaviour: No contact is deleted from the list. The error details are shown in the status message. The timestamp in the status bar is not updated.

3. Enter other variants of `delete` that is not adhering to the format in command box.

   Expected behaviour: No contact is deleted from the list. The error details are shown in the status message. The timestamp in the status bar is not updated.