ABC Business Contacts - Developer Guide

Table Of Contents

By: `Team W13-B3` Since: `Jun 2016` Licence: `MIT`

Introduction

Welcome to **ABC**!

ABC is a desktop Business Contact Management application, which is primarily a Command Line Interface (CLI) application that also provides a Graphical User Interface (GUI). It provides a convenient way for users to customize their contacts and keep track of appointments.

This guide provides information that will help you contribute to **ABC**, whether you are an experienced contributor or a first-time user. This includes information on the necessary requirements of this project, an overview of the software architecture, and implementation of key features.

1. Setting up

1.1. Prerequisites

1. **JDK** `1.8.0_60` or later

NOTE	This application is not compatible with earlier versions of Java 8.

2. **IntelliJ** IDE

NOTE	IntelliJ has Gradle and JavaFx plugins installed by default. Do not disable them. If they are disabled, go to `File` > `Settings` > `Plugins` to re-enable them.

1.2. Setting up the project on a computer

1. Fork this repo, and clone the fork to your computer.
2. Open IntelliJ (if the welcome screen is not displayed, 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, select it and 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.

1.3. Verifying the setup

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

1.4. Configurations to do before writing code

1.4.1. Configuring the coding style

This project follows oss-generic coding standards. IntelliJ's default style is mostly compliant with this standard, except for its different import order. 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, follow the UsingCheckstyle.adoc document to configure Intellij to check style-compliance while writing code.

1.4.2. Updating documentation to match the fork

After forking the repo, links in the documentation will still point to the `se-edu/addressbook-level4` repo. If there are plans to develop this as a separate product (i.e. instead of contributing to the `se-edu/addressbook-level4`) , the URL in the variable `repoURL` in `DeveloperGuide.adoc` and `UserGuide.adoc` should be replaced with the URL of the fork.

1.4.3. Setting up Continuous Integration (CI)

Set up Travis to perform CI for the fork. See UsingTravis.adoc for information on how to set it up.

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

NOTE	Having both Travis and AppVeyor ensures that the application works on both Unix-based platforms and Windows-based platforms (Travis is Unix-based and AppVeyor is Windows-based).

1.4.4. Getting started with coding

Before starting to code, it is advisable to get a sense of the overall design by reading the Architecture section below.

2. Design

2.1. Architecture

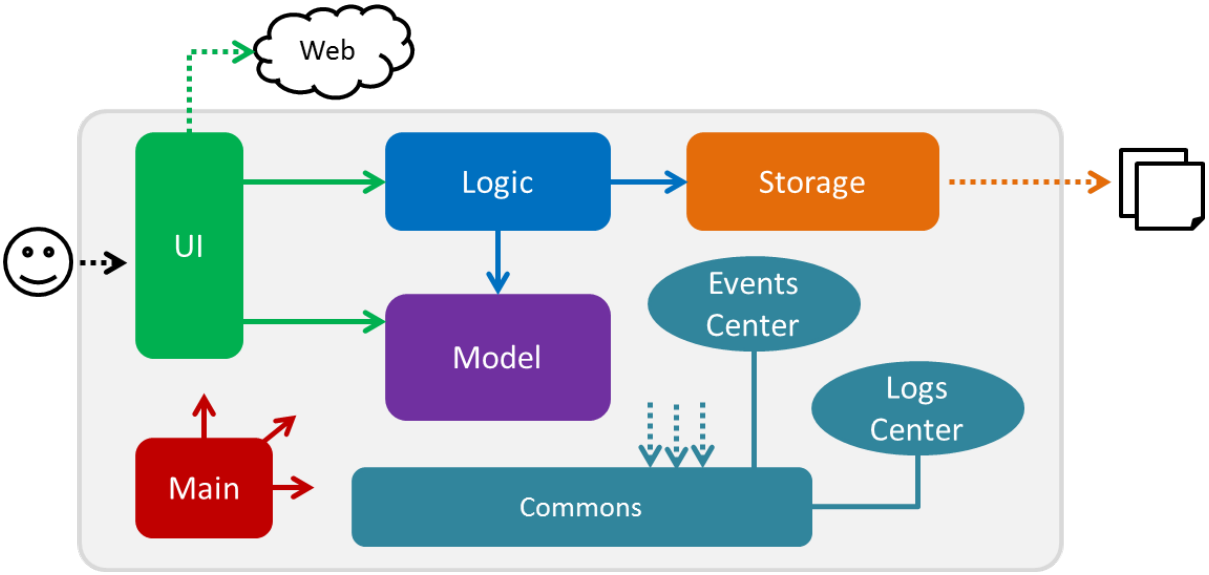

Figure 2.1.1 : Architecture Diagram

The ***Architecture Diagram*** (Figure 2.1.1) given above explains the high-level design of the application 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:

- Initializing the components in the correct sequence at application launch, and connecting them up with each other.
- Shutting down the components and invoking cleanup method where necessary.

`Commons` represents a collection of classes used by multiple other components. Two of those classes play important roles at the architecture level:

- `EventsCenter` (written using Google's Event Bus library) is used by components to communicate with other components using events (i.e. a form of Event Driven design)
- `LogsCenter` is used by many classes to write log messages to the application's log file.

The rest of the application consists of four components:

- `UI` : Takes charge of the UI of the application.
- `Logic` : Executes commands.
- `Model` : Holds the data of the application in-memory.
- `Storage` : Reads data from, and writes data to, the hard disk.

Each of the four components:

- Defines its *API* in an `interface` with the same name as the Component.
- Exposes its functionality via a `{Component Name}Manager` class.

For example, the `Logic` component (see Figure 2.1.2 for its class diagram) defines its API in the `Logic.java` interface and exposes its functionality via the `LogicManager.java` class.

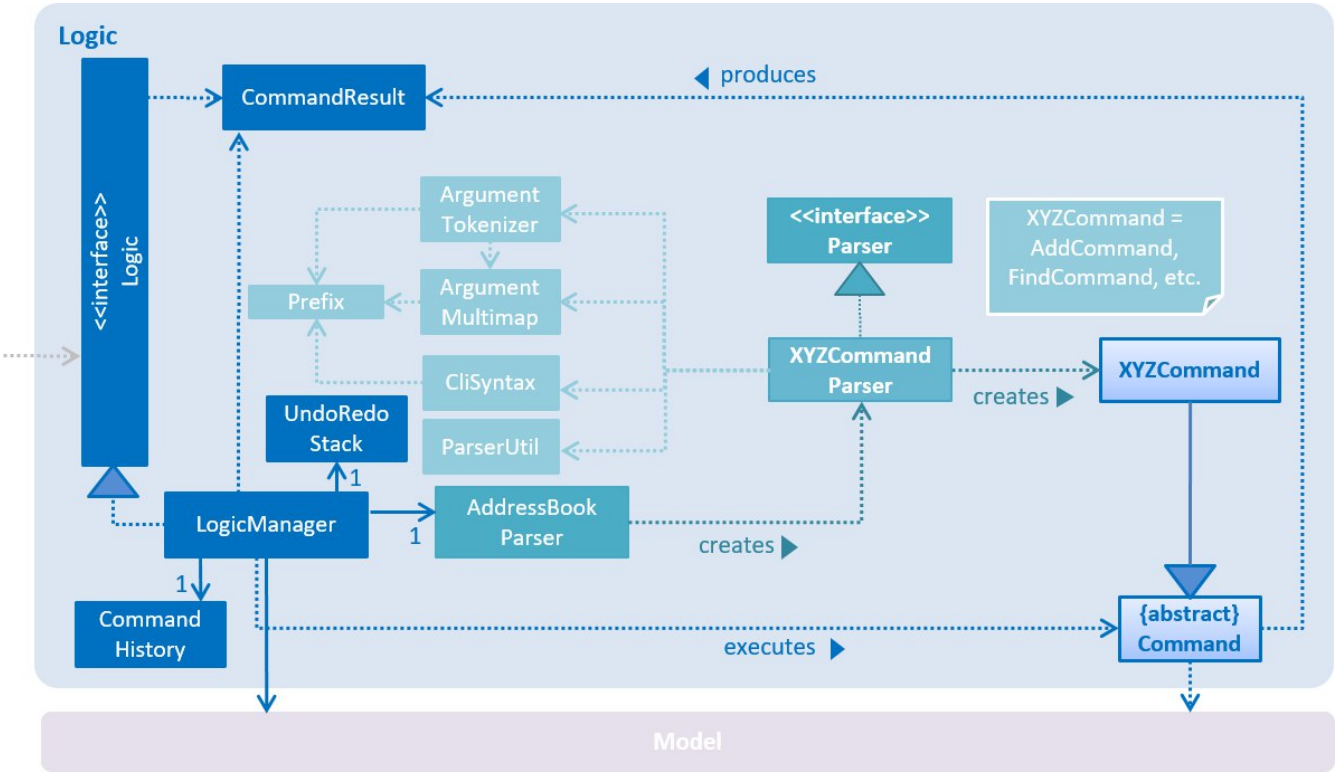

Figure 2.1.2 : Class Diagram of the Logic Component

Events-Driven nature of the design

The *Sequence Diagram* (Figure 2.1.3a) below shows how the components interact in a scenario where the user issues the command `delete 1`.

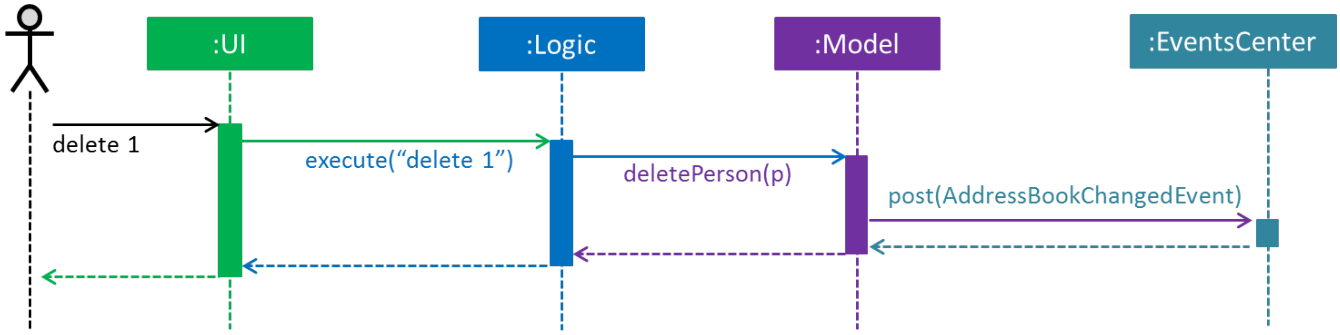

Figure 2.1.3a : Sequence Diagram for `delete 1` command (Part a)

NOTE `Model` simply raises a `AddressBookChangedEvent` when the Address Book data is changed, instead of asking `Storage` to save the updates to the hard disk.

The diagram (Figure 2.1.3b) below shows how `EventsCenter` reacts to that event, which eventually results in the updates being saved to the hard disk and the status bar of the UI being updated to reflect the 'Last Updated' time.

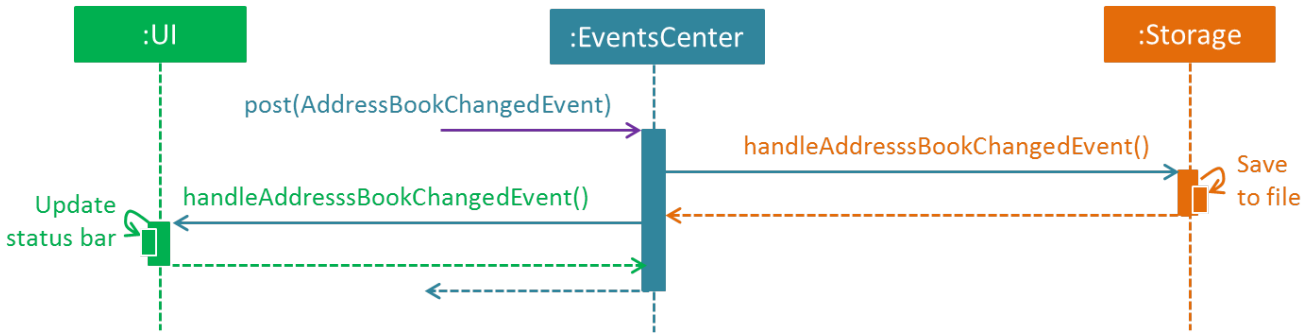

Figure 2.1.3b : Sequence Diagram for `delete 1` command (Part b)

NOTE The event is propagated through `EventsCenter` to `Storage` and `UI` without `Model` having to be coupled to either of them. This is an example of how this Event Driven approach helps us reduce direct coupling between components.

The sections below give more details of the four main components.

2.2. UI component

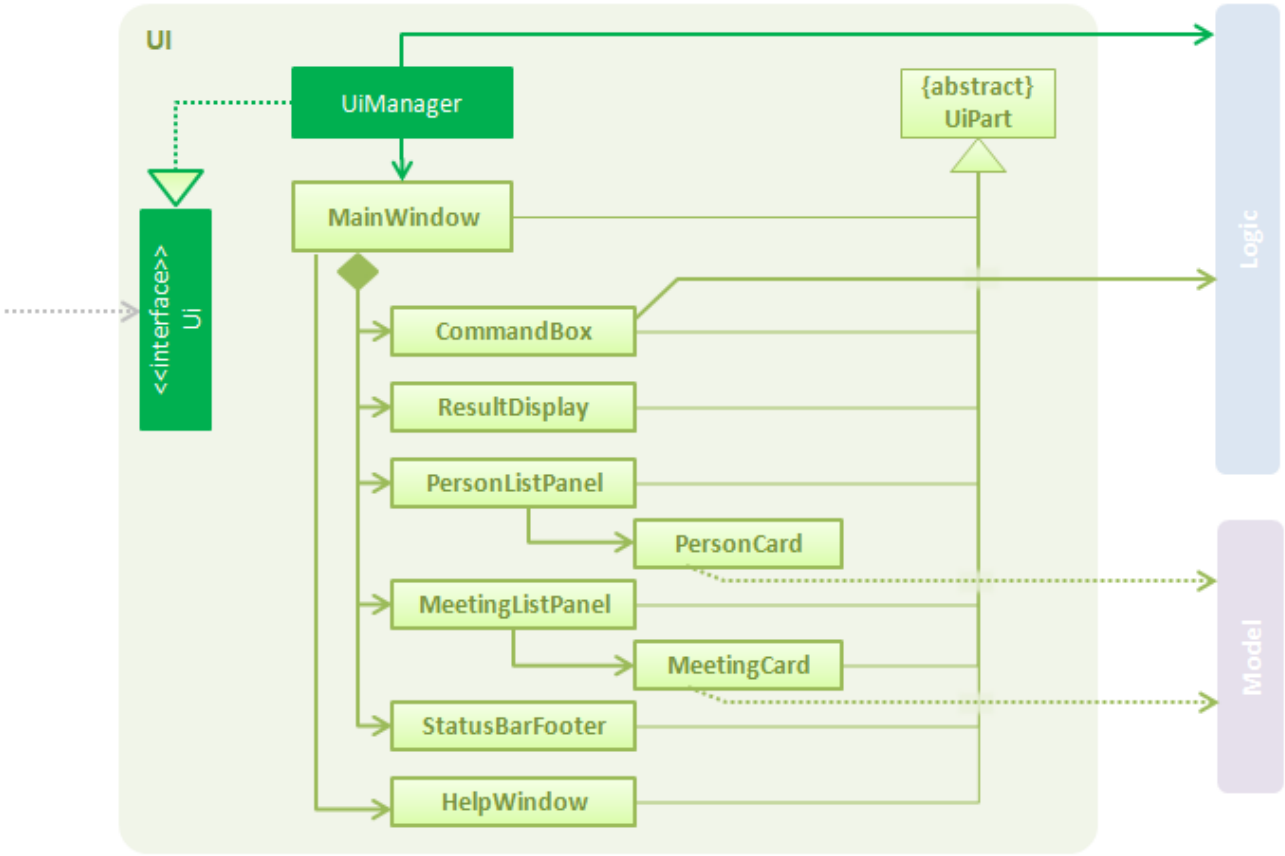

Figure 2.2.1 : Structure of the UI Component

API : `Ui.java`

As seen in *Figure 2.2.1*, the UI consists of a `MainWindow` that is made up of different parts such as `CommandBox`, `ResultDisplay`, `PersonListPanel`, `StatusBarFooter`, `MeetingListPanel`. All these, including the `MainWindow`, inherit from the abstract `UiPart` class.

The `UI` component uses JavaFx UI framework. The layout of these UI parts are defined in their corresponding `.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:

- Executes user commands using the `Logic` component.
- Binds itself to some data in `Model` so that the UI can be updated automatically when data in `Model` changes.
- Responds to events raised from various parts of the application and updates the UI accordingly.

2.3. Logic component

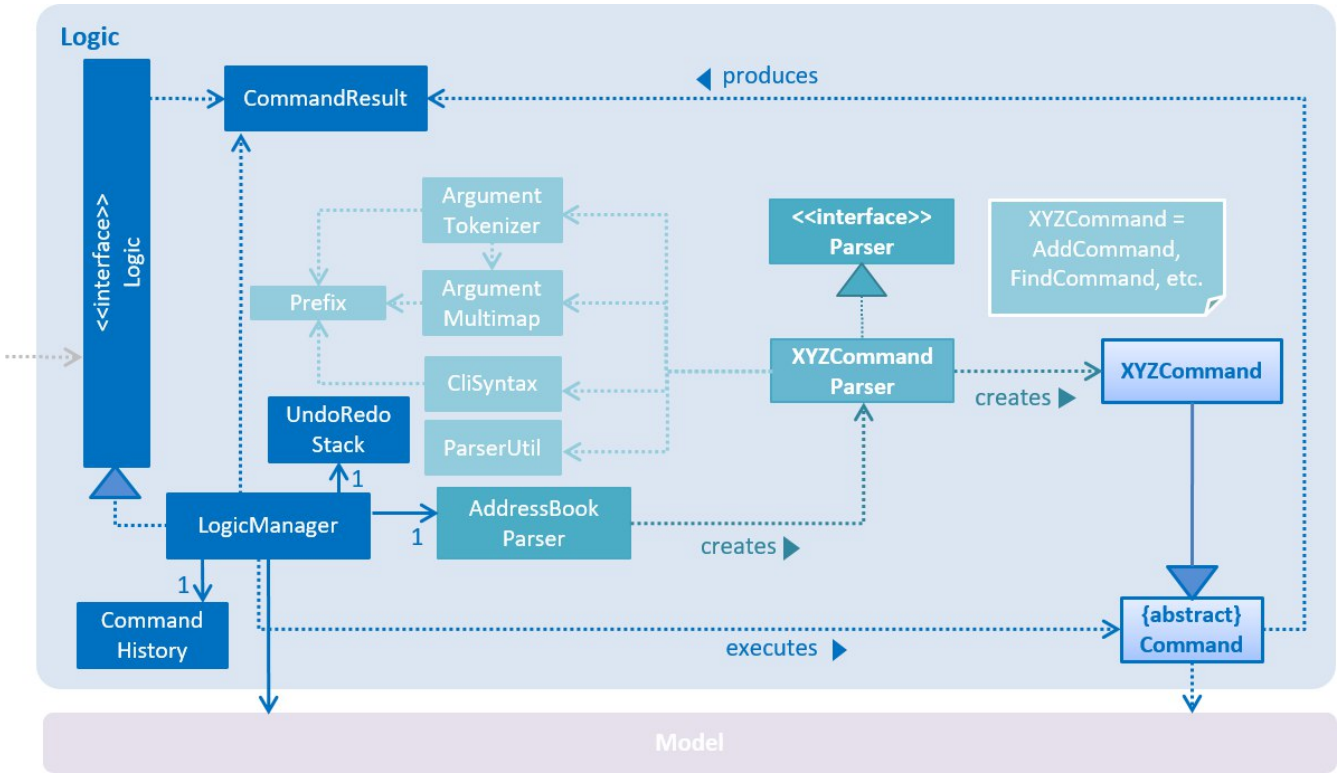

Figure 2.3.1 : Structure of the Logic Component

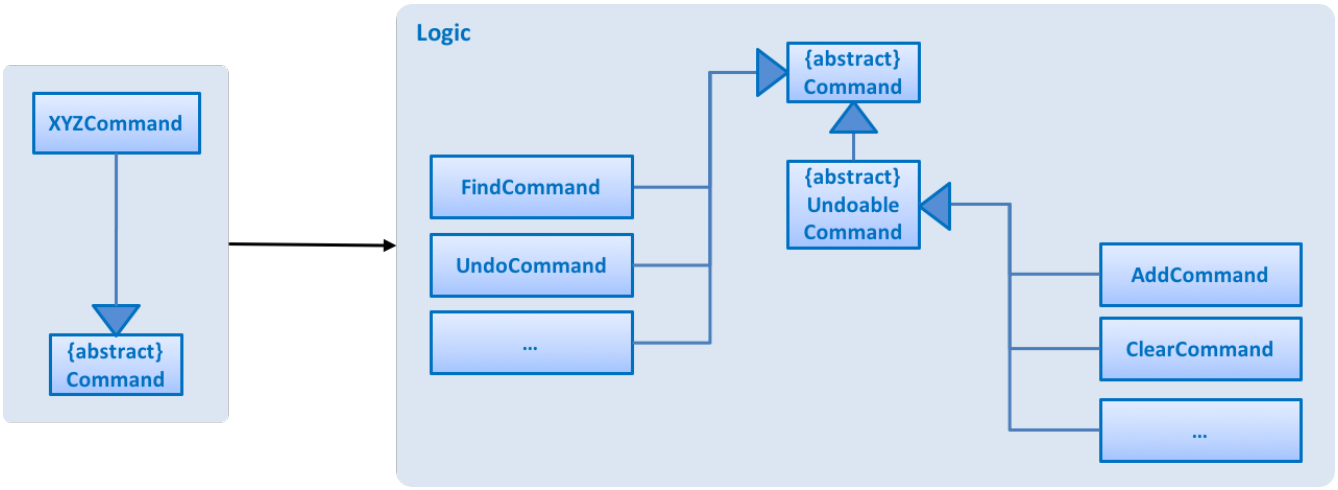

Figure 2.3.2 : Structure of Commands in the Logic Component. This diagram shows finer details concerning `XYZCommand` and `Command` in Figure 2.3.1

API : `Logic.java`

As seen in *Figure 2.3.1*, `Logic` uses the `AddressBookParser` class to parse the user command. This results in a `Command` object which is executed by `LogicManager`. The command execution can affect `Model` (e.g. adding a person) and/or raise events. The result of the command execution is encapsulated as a `CommandResult` object which is passed back to the `Ui`.

Figure 2.3.3 below shows the Sequence Diagram for interactions within the `Logic` component for the `execute("delete 1")` API call.

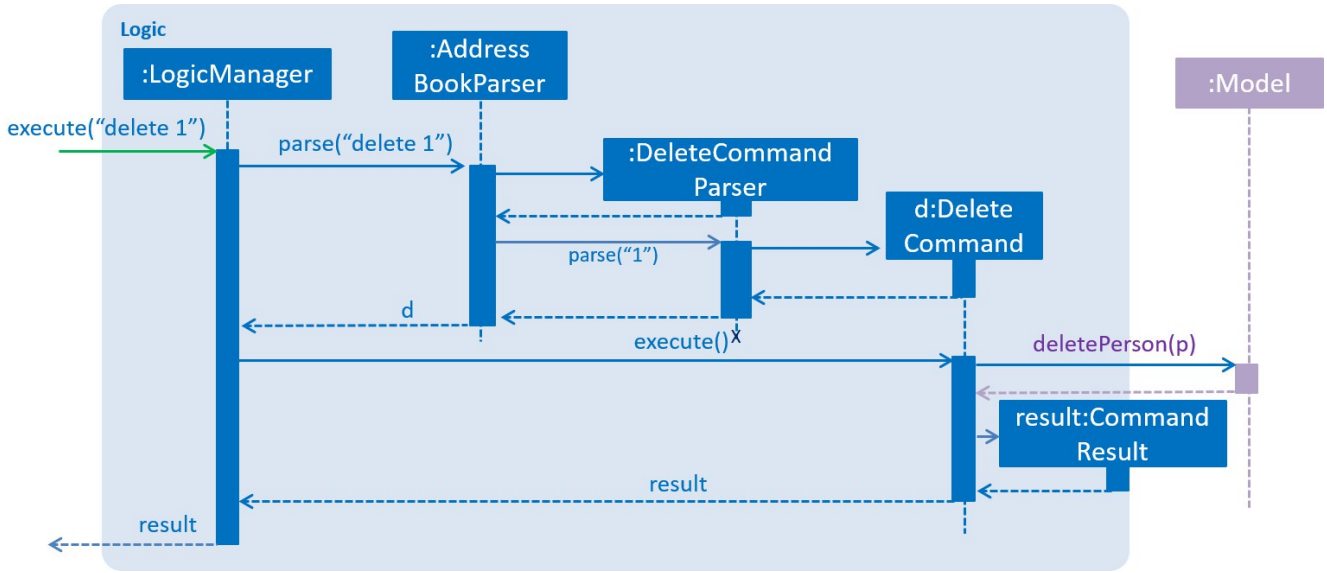

Figure 2.3.3 : Interactions Inside the Logic Component for the `delete 1` Command

2.4. Model component

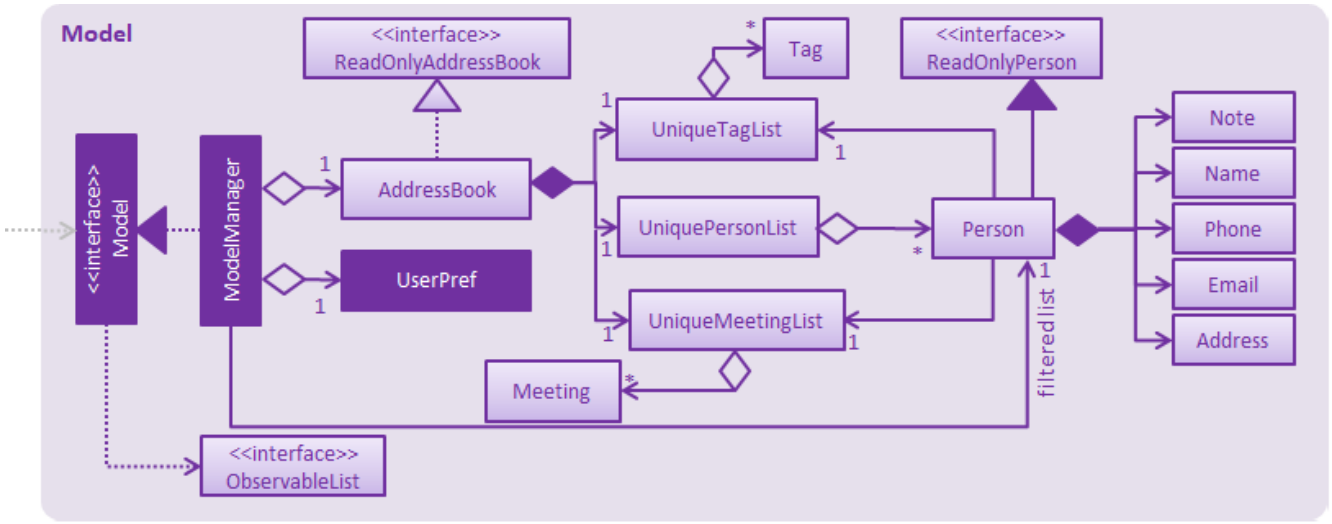

Figure 2.4.1 : Structure of the Model Component

API : `Model.java`

As seen in *Figure 2.4.1*, the `Model` component:

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

2.5. Storage component

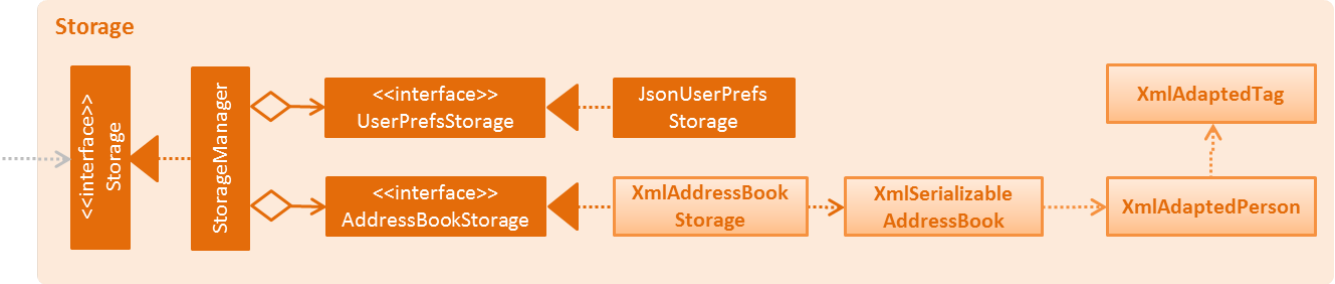

Figure 2.5.1 : Structure of the Storage Component

API : `Storage.java`

As seen in *Figure 2.5.1*, the `Storage` component:

- can save `UserPref` objects in json format and read it back.
- can save the Address Book data in xml format and read it back.

2.6. Common classes

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

3. Implementation

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

3.1. Undo/Redo mechanism

3.1.1. Mechanism

The undo/redo mechanism is facilitated by an `UndoRedoStack`, which resides inside `LogicManager`. It supports undoing and redoing of commands that modify the state of the application (e.g. `add`, `edit`). Such commands will inherit from `UndoableCommand`.

`UndoRedoStack` only deals with `UndoableCommands`. Commands that cannot be undone will inherit from `Command` instead. The following diagram, Figure 3.1.1.1, shows the inheritance diagram for commands:

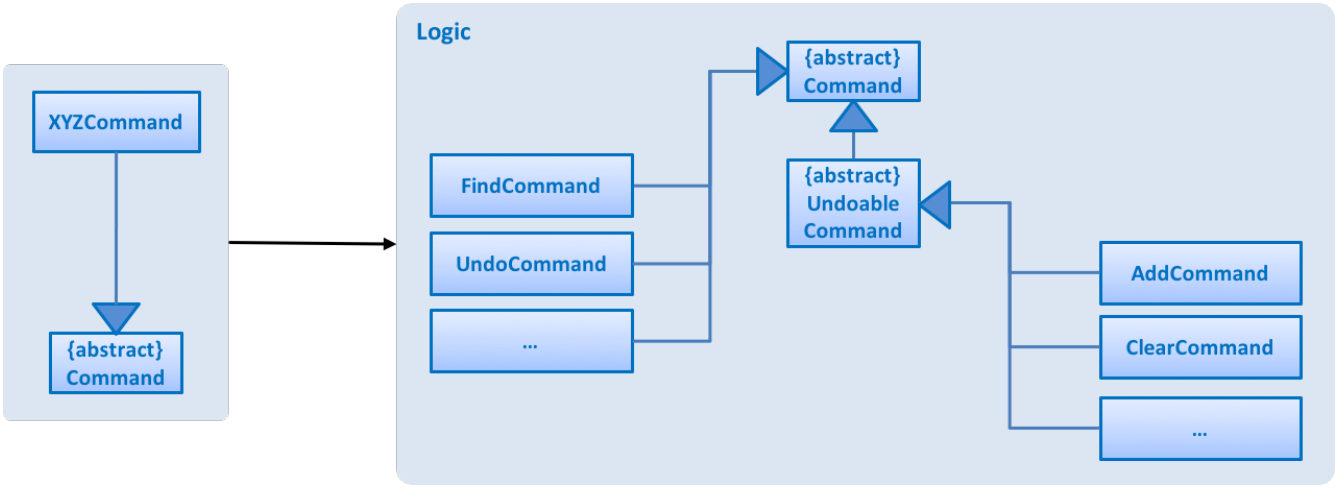

Figure 3.1.1.1 : Logic Command Class Diagram

As seen from Figure 3.1.1.2, `UndoableCommand` adds an extra layer between the abstract `Command` class and concrete commands that can be undone, such as `DeleteCommand`. Note that extra tasks need to be done when executing a command that can be undone, such as saving the state of the application before execution. `UndoableCommand` contains the high-level algorithm for these extra tasks while the child classes implement the details of how to execute the specific command. Note that this technique of putting the high-level algorithm in the parent class and lower-level steps of the algorithm in child classes is also known as the template pattern.

Commands that are not undoable are implemented this way:

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

With the extra layer, the undoable commands are implemented as follows:

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

      executeUndoableCommand();
  }
}

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

Suppose that the user has just launched the application. The `UndoRedoStack` will be empty at the start.

The user executes a new `UndoableCommand`, `delete 5`, to delete the 5th person in the address book. The current state of the address book is saved before the `delete 5` command executes. The `delete 5` command will then be pushed onto the `undoStack` (the current state is saved together with the command). This is illustrated by Figure 3.1.1.2.

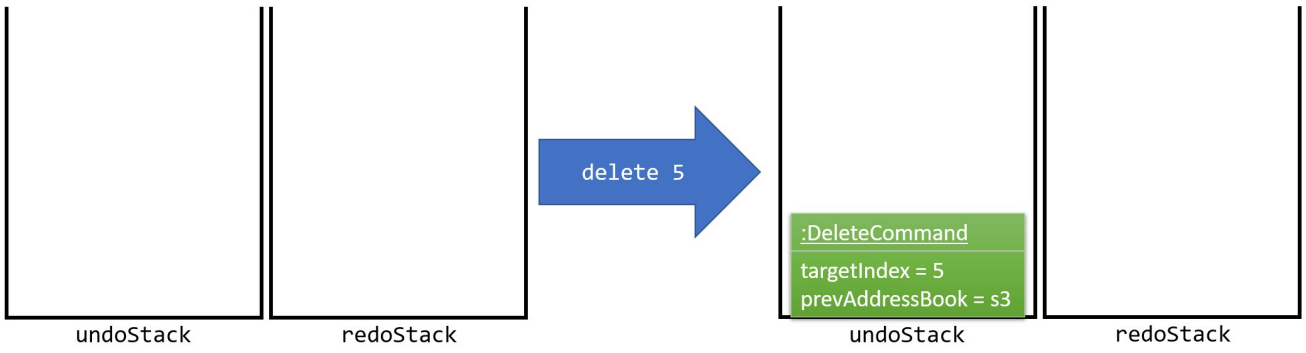

Figure 3.1.1.2 : Execute Delete Command Stack Diagram

As the user continues to use the program, more commands are added into the `undoStack`. For example, the user may execute `add n/David …` to add a new person. This is shown in Figure 3.1.1.3.

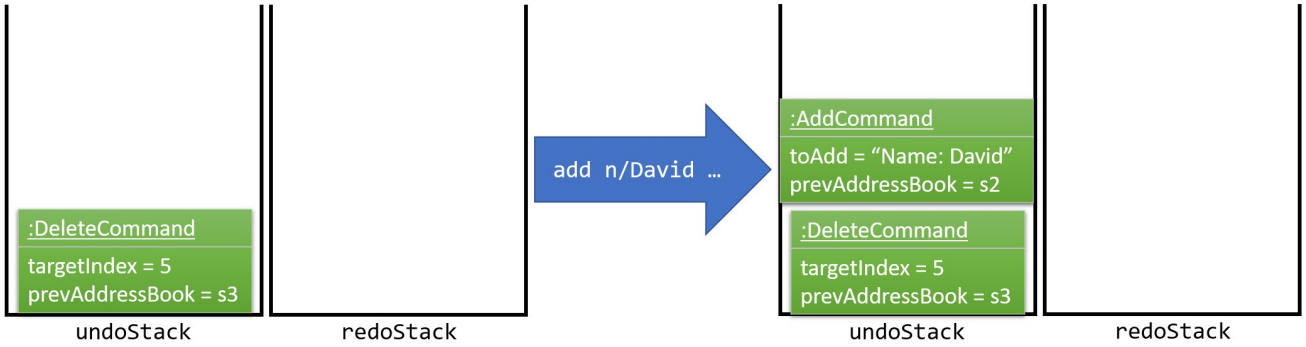

Figure 3.1.1.3 : Execute Add Command Stack Diagram

NOTE If a command fails its execution, it will not be pushed to the `UndoRedoStack` at all.

The user now decides that adding the person was a mistake, and executes `undo` to undo his previous command.

As can be seen from Figure 3.1.1.4, the most recent command is popped out of the `undoStack` and pushed back to the `redoStack`. The address book is then restored to the state before the `add` command executed.

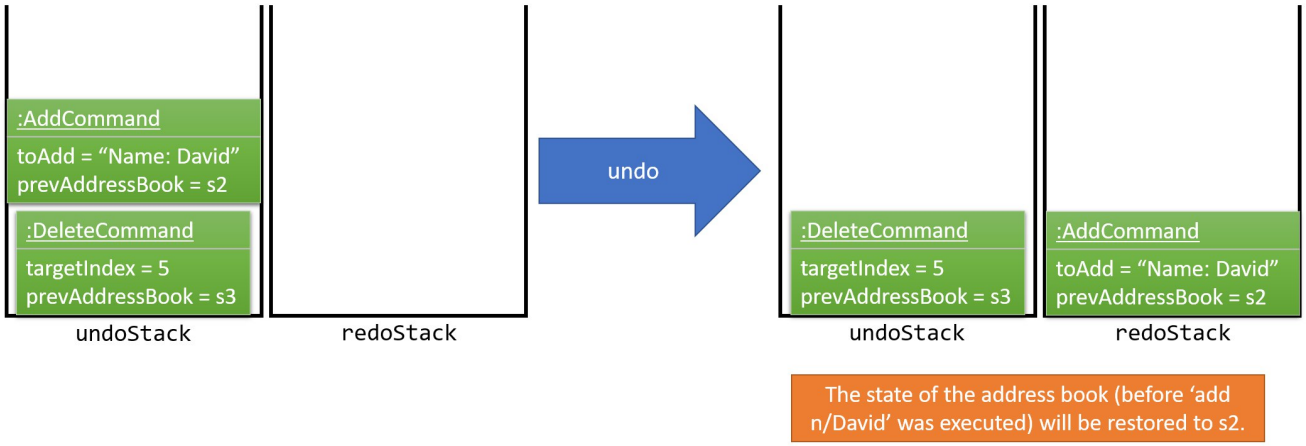

Figure 3.1.1.4 : Execute Undo Command Stack Diagram

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

The following sequence diagram, Figure 3.1.1.5, shows how the undo operation works:

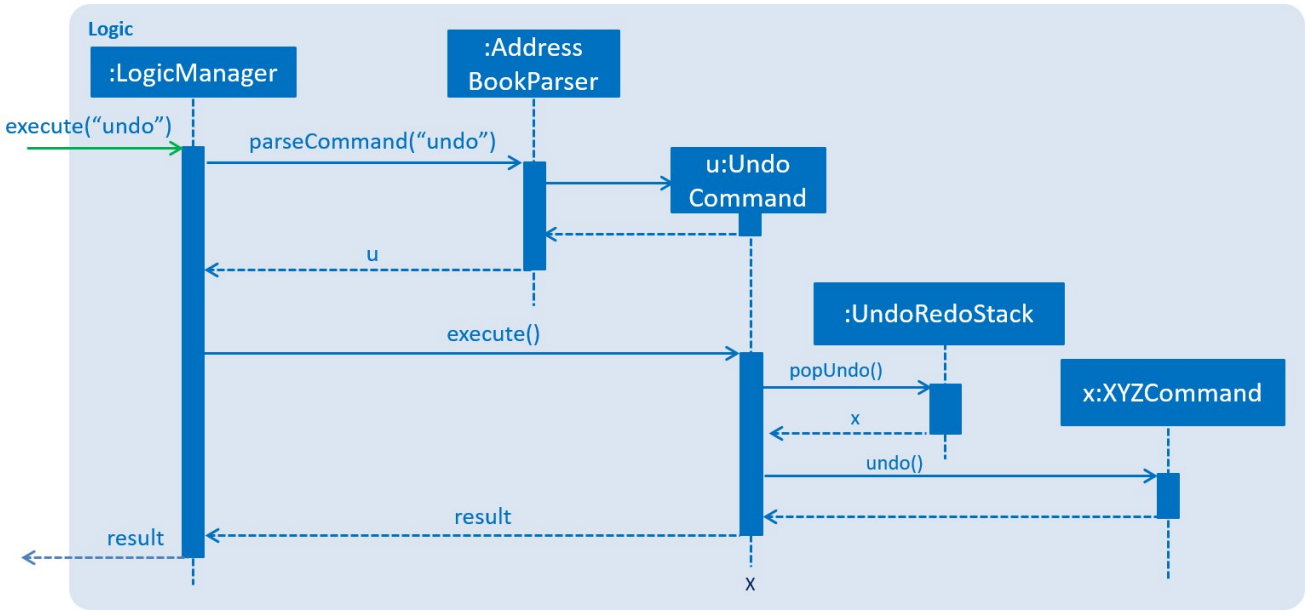

Figure 3.1.1.5 : Undo/Redo Sequence Diagram

The redo does the exact opposite (pops from `redoStack`, pushes to `undoStack`, and restores the address book to the state after the command is executed).

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

The user now decides to execute a new command, `clear`. As before, `clear` will be pushed into the `undoStack`. The `redoStack` is no longer empty. It will be purged as it no longer makes sense to redo the `add n/David` command (this is the behavior that most modern desktop applications follow). This is shown in Figure 3.1.1.6.

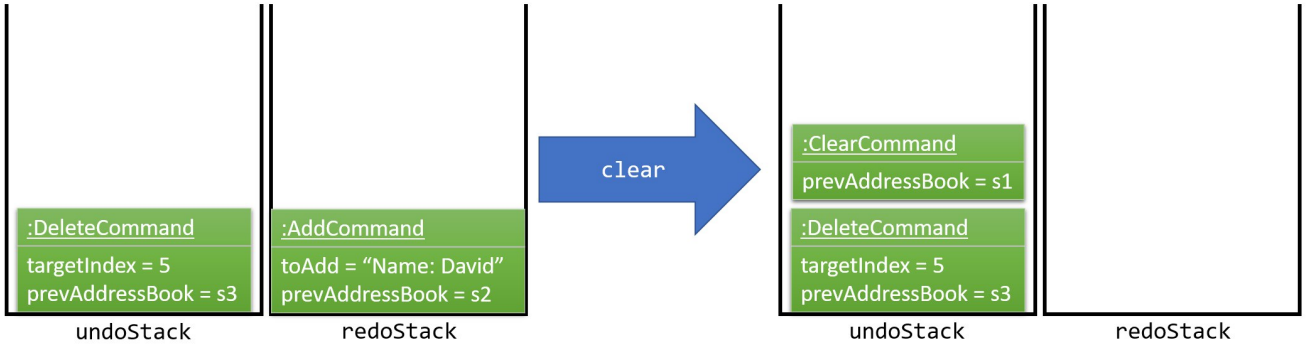

Figure 3.1.1.6 : Execute Clear Command Stack Diagram

Commands that are not undoable are not added into the `undoStack`. For example, `list`, which inherits from `Command` rather than `UndoableCommand`, will not be added after execution, as shown in Figure 3.1.1.7.

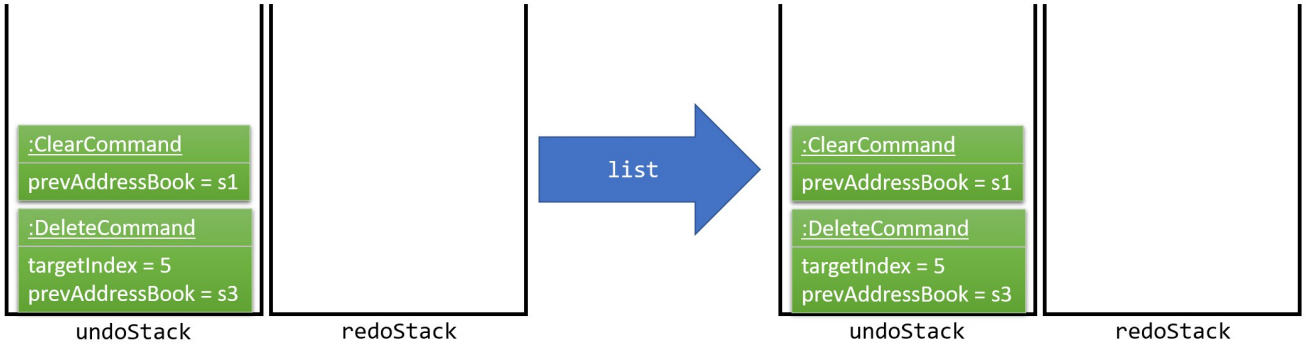

Figure 3.1.1.7 : Execute List Command Stack Diagram

The following activity diagram, Figure 3.1.1.8, summarizes what happens inside the `UndoRedoStack` when a user executes a new command:

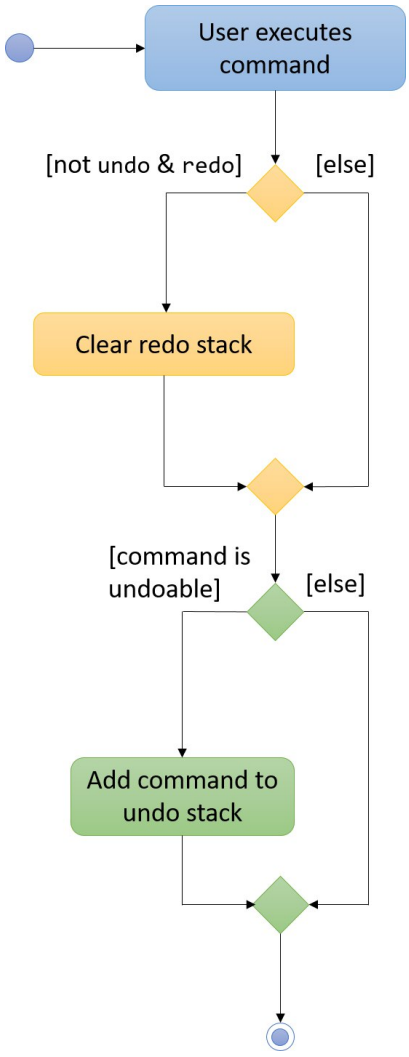

Figure 3.1.1.8 : Undo/Redo Activity Diagram

3.1.2. Design Considerations

Aspect: Implementation of `UndoableCommand`
Alternative 1 (current choice): Add a new abstract method `executeUndoableCommand()`
Pros: No undone/redone functionality is lost as it is now part of the default behaviour. Classes that deal with `Command` do not have to know that `executeUndoableCommand()` exists.
Cons: Difficult for new developers to understand the template pattern.
Alternative 2: Override `execute()`
Pros: Does not involve the template pattern, easier for new developers to understand.
Cons: Classes that inherit from `UndoableCommand` must remember to call `super.execute()`, or lose the ability to undo/redo.

Aspect: How undo & redo executes
Alternative 1 (current choice): Save the entire address book
Pros: Easy to implement.
Cons: May have performance issues in terms of memory usage.
Alternative 2: Individual command is able to undo/redo itself
Pros: Will use less memory (e.g. for `delete`, just save the person being deleted).
Cons: Must ensure that the implementation of each individual command is correct.

Aspect: Type of commands that can be undone/redone
Alternative 1 (current choice): Only include commands that modify the address book (`add`, `clear`, `edit`)
Pros: Only need to revert changes that are hard to change back (the view can easily be re-modified as no data is lost).
Cons: User might think that list modifying operations are also undoable (undoing filtering, for example), only to realize that it does not do that.
Alternative 2: Include all commands
Pros: More intuitive for the user.
Cons: User has no way of skipping such commands if he wants to reset the state of the address book and not the view.
Additional Info: See our discussion here.

Aspect: Data structure to support the undo/redo commands
Alternative 1 (current choice): Use separate stack for undo and redo
Pros: Easy to understand for new Computer Science student undergraduates, 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 `UndoRedoStack`.
Alternative 2: Use `HistoryManager` for undo/redo
Pros: We do not need to maintain a separate stack, 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.

3.2. Backup/Restoring Backup

3.2.1. Mechanism

The backing up of ABC is done by `BackupCommand` and the restoring of data from a backup file is done by `RestoreBackupCommand`. `BackupCommand` inherits from `Command` as it does not support the undoing and redoing of user actions, whereas `RestoreBackupCommand` inherits from `UndoableCommand`. These commands require access to `Storage` from `Logic` and this is accomplished by posting an event to `EventsCenter`. `Subscribers` in `StorageManager` will handle these events and respond correspondingly. The sequence diagram below (Figure 3.2.1.1) shows how the `BackupCommand` is carried out.

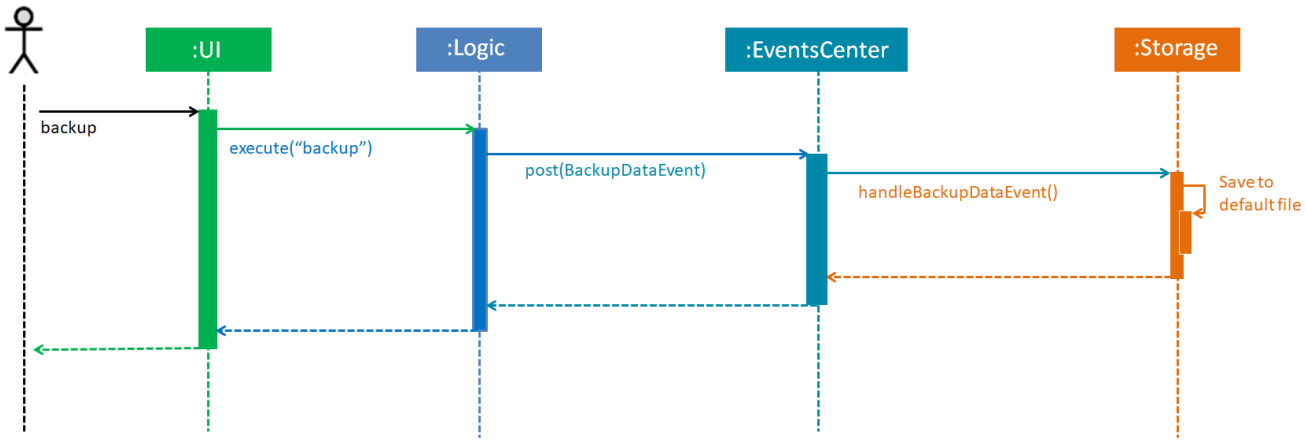

Figure 3.2.1.1 : Backup Command Sequence Diagram

NOTE `RestoreBackupCommand` shares a similar flow for its sequence diagram.

The `BackupCommand` is executed when the command `backup` is entered. The data that is in `Model` or the active address book is first passed as a parameter to `BackupDataEvent`. The event will be handled by `StorageManager` and is saved into the default file path "data/addressbook-backup.xml". The following is the implementation of `BackupCommand`:

```
public class BackupCommand extends Command {
  //... variables, constructor, other methods...

  @Override
  public CommandResult execute() throws CommandException {
      // reading data from model
      ReadOnlyAddressBook backupAddressBookData = model.getAddressBook();

      // posting event to backup data
      EventsCenter.getInstance().post(new BackupDataEvent(backupAddressBookData));
      return new CommandResult(String.format(MESSAGE_SUCCESS));
  }
}
```

The `RestoreBackupCommand` is executed when the command `restore` is entered. `RestoreBackupDataEvent` is posted and `StorageManager` handles it. The data from default file path

"data/addressbook-backup.xml" will be retrieved and it will replace the active address book. The following is the implementation of `RestoreBackupCommand`:

```
public class RestoreBackupCommand extends UndoableCommand {
  //... variables, constructor, other methods...

  @Override
  public CommandResult execute() throws CommandException {
      //... other codes and checks...

      RestoreBackupDataEvent event = new RestoreBackupDataEvent();

      // posting event to help with restoring backup data
      EventsCenter.getInstance().post(event);

      // overwriting the data in active address book
      ReadOnlyAddressBook backupAddressBookData = event.getAddressBookData();
      model.resetData(backupAddressBookData);
      return new CommandResult(String.format(MESSAGE_SUCCESS));

      //... other codes and checks...
  }
}
```

If the backup file does not exist in the default file path, an error message will be shown to the user. This check is done before `RestoreBackupDataEvent` is posted. Once again, this requires `Logic` to access `Storage`. Therefore, a `BackupFilePresentEvent` will be posted and the `Subscriber` in `StorageManager` would handle this event to check if the backup file exists.

NOTE	A backup of the data is automatically created when **ABC** is closed.

3.2.2. Design Considerations

Aspect: Accessing `Storage` from `Logic`
Alternative 1 (current choice): Make use of `EventBus` to post events and have `StorageManager` handle the backing up or retrieval of data
Pros: Follow the architecture closely without introducing dependencies between components.
Cons: New `Event` classes have to be created every time a command requires access to data in the storage.
Alternative 2: Allow `Logic` to access `Storage` and its functions
Pros: Easier implementation for current and future functions or commands related to `Storage`.
Cons: Increases coupling between the components.

3.3. Adding/Removing a tag

3.3.1. Mechanism

Adding or removing a tag is facilitated by `AddTagCommand` and `DeleteTagCommand`, which are subclasses of `UndoableCommand`. These commands work by changing the value of the `Tag` objects associated with the contact.

These commands take in an integer and a string as arguments. The command is first parsed in `AddressBookParser` to identify it as the appropriate command. It will then be parsed by `AddTagCommandParser` or `DeleteTagCommandParser`, to parse the index, which was the integer argument, and the `Tag`, which was represented by the string argument. Invalid indexes and tags will be handled by throwing an exception. This is how `AddTagCommandParser` is implemented:

```
public class AddTagCommandParser implements Parser<AddTagCommand> {
  public AddTagCommand parse(String args) throws ParseException {
      try {
          // ... parse `Index` and `Tag` and pass it to `AddTagCommand` ...
      } catch (IllegalValueException ive) {
          // ... throw an exception ...
      }
  }
}
```

To update the `Tag` objects associated with a `Person`, the set of `Tag` objects belonging to that `Person` is copied to a new set. The new data is then modified, then copied into a newly created `Person` instance. This is implemented as follows:

```
public class AddTagCommand extends UndoableCommand {
  // ... variables, constructor, other methods ...
  private final Tag newTag;

  @Override
  public CommandResult executeUndoableCommand() throws CommandException {
      // ... fetch personToEdit ...

      Set<Tag> oldTags = new HashSet<Tag>(personToEdit.getTags());
      // ... check if tag is duplicated ...
      Person editedPerson = new Person(personToEdit);
      oldTags.add(newTag);
      editedPerson.setTags(oldTags);

      // ... try to replace personToEdit with editedPerson ...
  }
}
```

The diagram below (Figure 3.3.1.1) shows how `AddTagCommand` works.

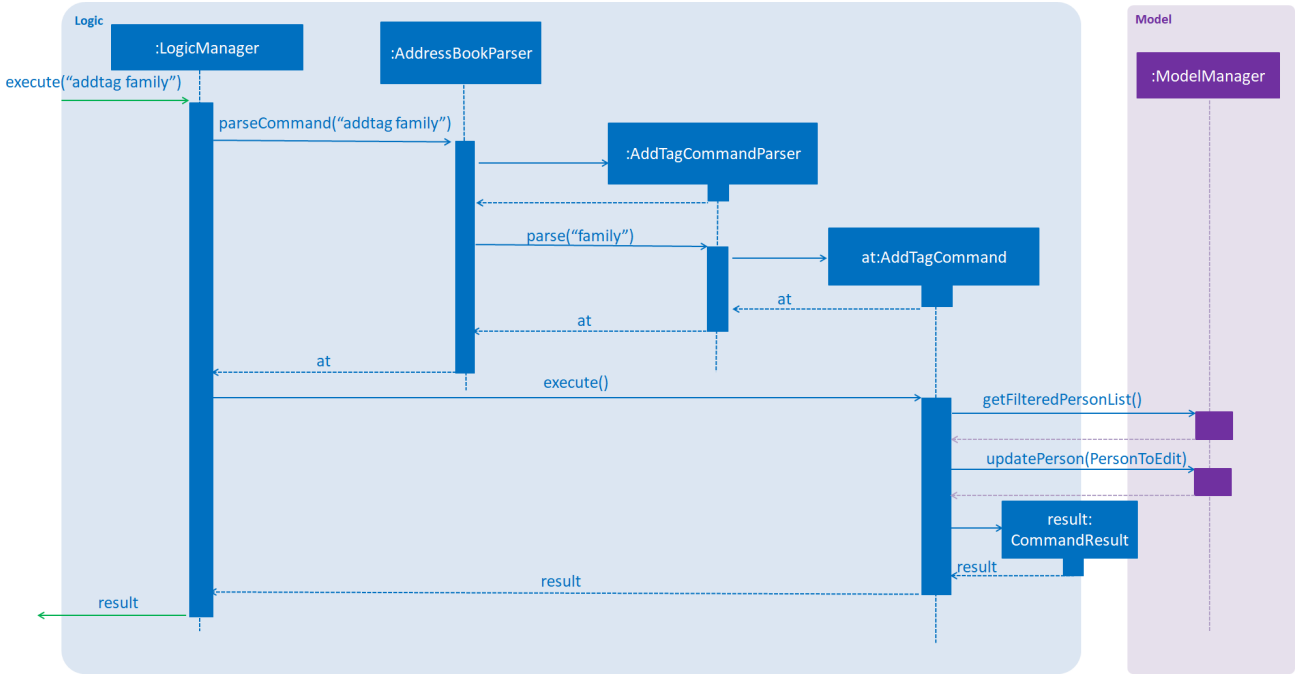

Figure 3.3.1.1 : AddTag Command Sequence Diagram

`RemoveTagCommand` works in a similar way. Note that `AddTagCommand` will throw an exception if the `Tag` already exists for the `Person` selected. `DeleteTagCommand` throws an exception if the `Tag` is not found on the `Person`.

3.3.2. Design Considerations

Aspect: Changing the `Tag` objects of the selected `Person`
Alternative 1 (current choice): Copy set of `Tag` objects to a newly created set and modify the newly created set, then create a copy of the selected `Person` instance and replace its set of `Tag` objects
Pros: Ensures that the original value will be unchanged, which is important in the event that updating the `Person` instance fails in a later stage.
Cons: Additional memory required to create a new `Person` instance.
Alternative 2: Edit the `Tag` set directly
Pros: No need to instantiate new `Person` instance. Easy to implement.
Cons: Problematic implementation and bad coding practice. Modifying the original values directly can cause problems if updating the `Person` instance fails in a later stage.

3.4. Filtering mechanism in find

3.4.1. Basic mechanism

The list of persons displayed is filtered by a [Predicate] when the method `updateFilteredPersonList(predicate)` from the `Model` interface is invoked.

The relevant methods in the Model interface are as follows:

```
public interface Model {

  ...

  /** Returns the predicate of the current filtered person list */
  Predicate<? super ReadOnlyPerson> getPersonListPredicate();

  /** Updates the filter of the filtered person list to filter by the given {@code
predicate}.*/
  void updateFilteredPersonList(Predicate<ReadOnlyPerson> predicate);

}
```

When `updateFilteredPersonList(predicate)` is invoked, every `Person` in **ABC** is evaluated against the `predicate`. A `Person` is added to the displayed list if `predicate.test(person)` is evaluated to be TRUE. Therefore, all `Person` instances that fulfill the conditions specified in `predicate` are displayed.

3.4.2. Filtering the displayed list

Note that all `Person` instances in the displayed list satisfy a Predicate `currentPredicate`. Given a new Predicate `newPredicate`, filtering the displayed list of contacts is equivalent to selecting `Person` instances that satisfy both `currentPredicate` and `newPredicate`. From Figure 3.4.2.1, it can also be viewed as the intersection of two lists of `Person` objects, each satisfying one of the two predicates respectively.

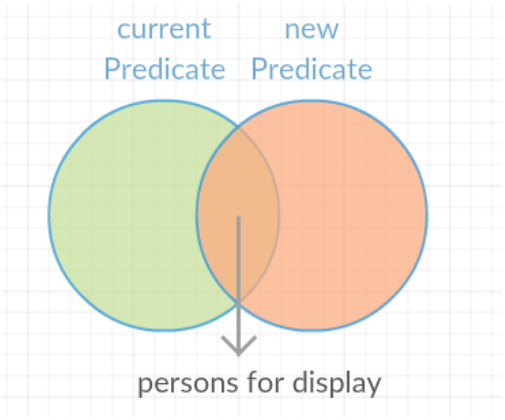

Figure 3.4.2.1 : Venn Diagram for Filtering

3.4.3. Implementation

The actual implementation of filtering the displayed list involves three steps.

1. Invoke `getPersonListPredicate()` provided in the Model interface to get the `currentPredicate`.
2. Use [Predicate.and()] to generated the logical AND of the two predicates.
3. Update the list using the predicate generated in step 2.

For more details, refer to the sequence diagram(Figure 3.4.3.1) below.

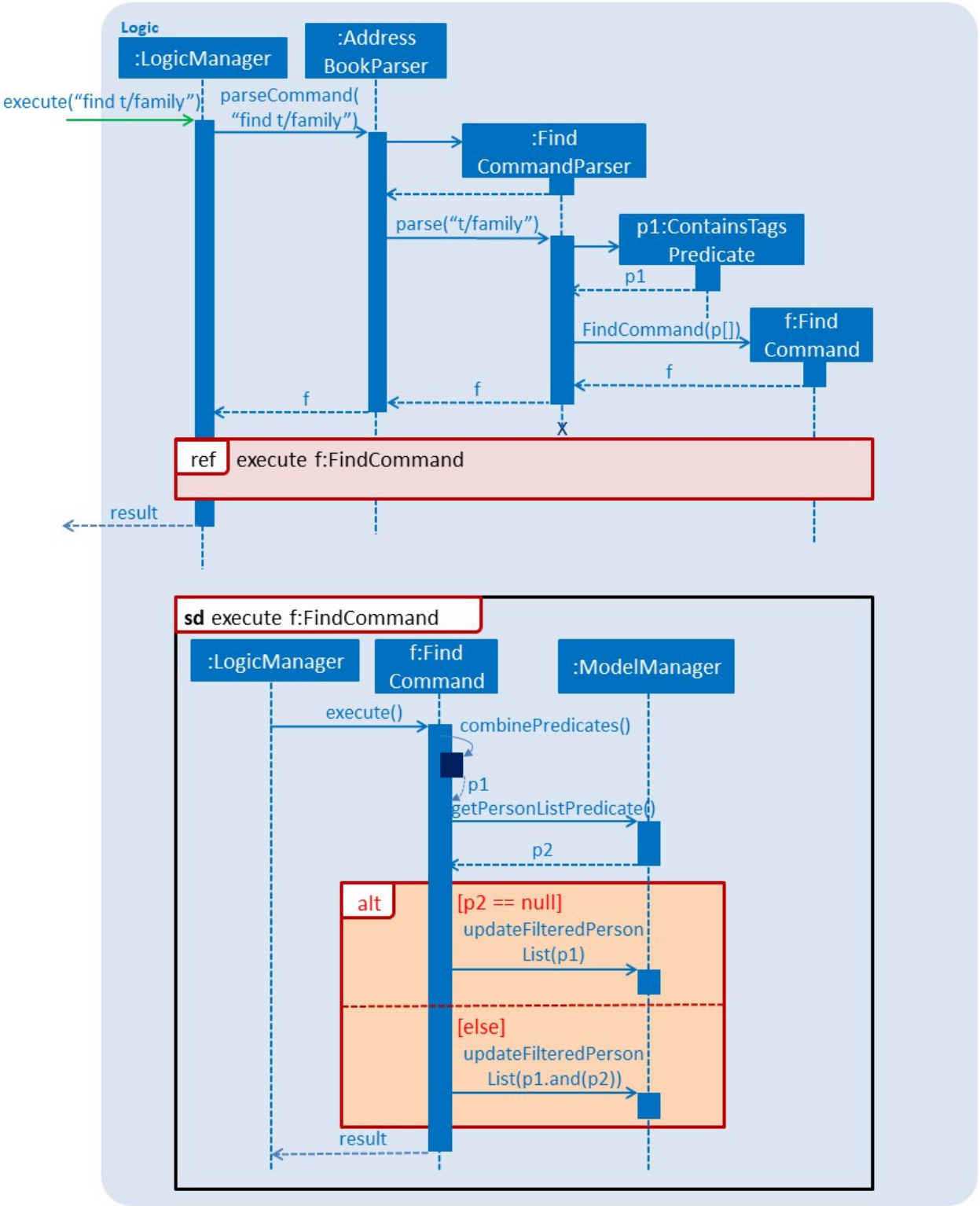

Figure 3.4.3.1 : Sequence Diagram for Find

3.4.4. Design consideration

The design for filtering the displayed list applies the [Open/Close Principle].

- By providing a new extension of `getPersonListPredicate()` in the `Model` interface, the new feature is enabled.
- By making use of the logical AND of two predicates, the list can be filtered without modification of the fundamental filtering mechanism.

3.5. Synchronisation with Google Contacts

3.5.1. Mechanism

Authentication and bi-directional synchronisation of data with a user's Google Contacts is done via the `sync` command, which is a subclass of `Command`. This command works in conjunction with the Google Client and People API.

A `PeopleService` instance is required and obtained via the `LoginCommand` before synchronisation is possible. `PeopleService` is then used to perform Create, Read, Update, and Delete (CRUD) operations on the user's Google Contacts, which is used in `SyncCommand`. The four primary methods in `SyncCommand` are `checkContacts`, `updateContacts`, `importContacts` and `exportContacts`.

The sequence diagram for the command can be seen below, in Figure 3.5.1.1:

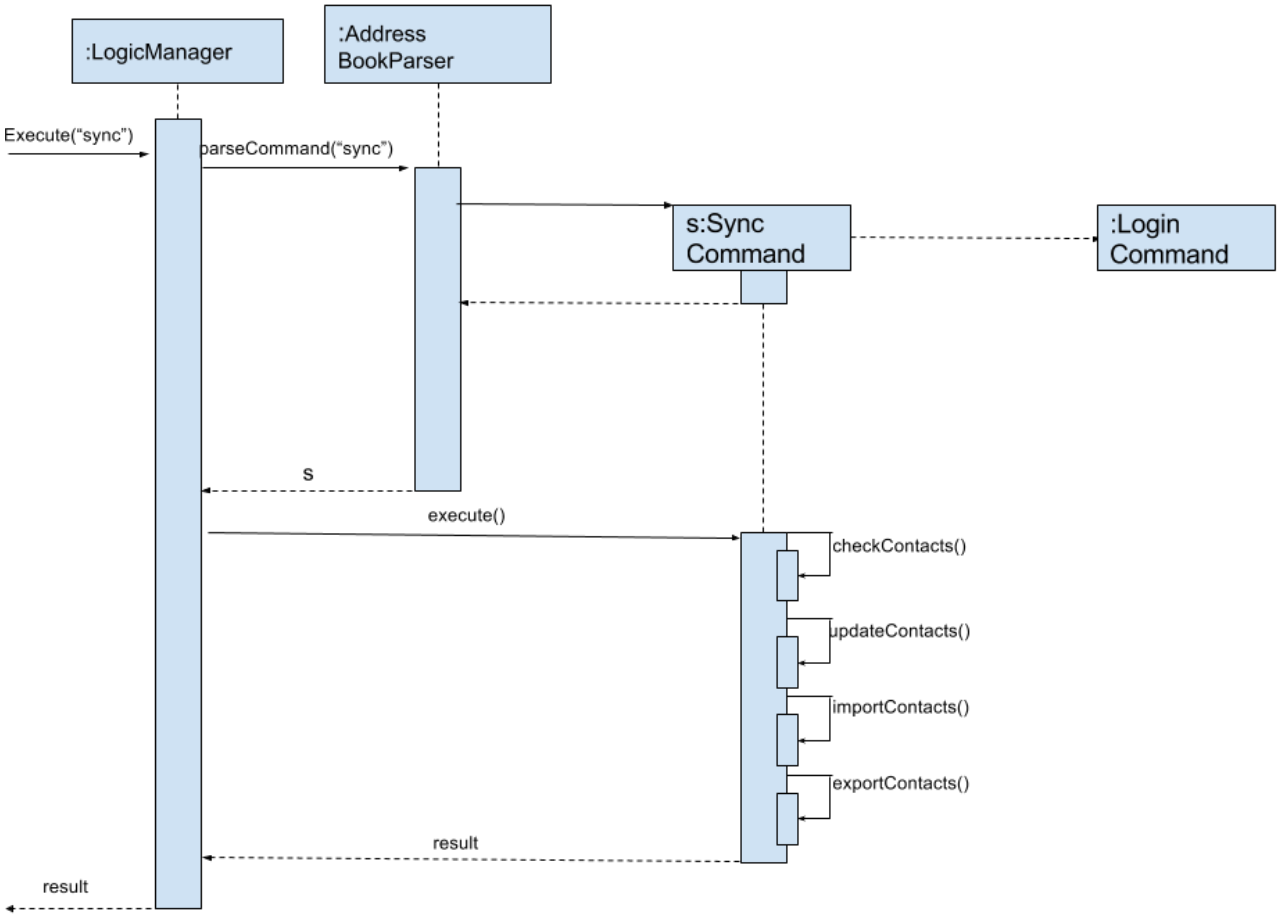

Figure 3.5.1.1: Sync Command Sequence Diagram

3.5.2. Methods

SyncCommand

Below is the implementation of SyncCommand. Upon execution, the command checks if a PeopleService instance has been instantiated, and throws a `CommandException` if it has not. It then runs the `initialise` method, which preprocesses the **ABC** and Google Contacts data, before

performing the 4 main functions, `checkContacts`, `updateContacts`, `importContacts` and `exportContacts`.

```
public class SyncCommand extends Command {
  //...variables, constructor, other methods

    @Override
    public CommandResult execute() throws CommandException {

        if (clientFuture == null || !clientFuture.isDone()) {
            throw new CommandException(MESSAGE_FAILURE);
        } else {

            syncedIDs =  (loadStatus() == null) ? new HashSet<String>() : (HashSet)
loadStatus();

            try {
                List<ReadOnlyPerson> personList = model.getFilteredPersonList();
                initialise();
                checkContacts();
                updateContacts();
                exportContacts(personList);
                if (connections != null) {
                    importContacts();
                }

                saveStatus(syncedIDs);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        return new CommandResult(String.format(MESSAGE_SUCCESS));
    }
}
```

Checking and updating of contacts

A `syncedIDs.dat` file is maintained to keep track of the contacts that have been synchronised currently, as each link between an **ABC** and a Google contact consists of an unique ID. `checkContacts` ensures that contacts on both ends still exist, and removes the link if either of them no longer exist. `updateContacts`, on the other hand, compares linked contacts, and if there is a difference, it updates the contact that has an older timestamp. Below is the implementation of the 2 functions respectively.

```
public class SyncCommand extends Command {
  // variables, constructor and other methods
  private void checkContacts() throws Exception {
        List<ReadOnlyPerson> personList = model.getFilteredPersonList();
        for (ReadOnlyPerson person : personList) {
            String id = person.getId().getValue();

            if (!hashGoogleId.containsKey(id)) {
                logger.info("Deleting local contact");
                model.deletePerson(person);
                syncedIDs.remove(id);
                continue;
            }

        }
  }

      private void updateContacts() throws Exception {
        List<String> toRemove = new ArrayList<String>();
        for (String id : syncedIDs) {
            seedu.address.model.person.ReadOnlyPerson aPerson;
            Person person;
            // Checks whether person and aPerson still exists, mainly for defensive
programming
            // We check the last updated times for both contacts
            if (compare < 0) {
                // We update the remote contact
            } else if (compare > 0) {
                // We update the local contact
            }
        }
    }
}
```

Importing and exporting of contacts

We then move on to importing of new Google Contacts, and exporting of new **ABC** contacts to Google servers. To achieve this, we iterate through all Google Contacts and **ABC** contacts respectively, and import or export them accordingly if they are not linked with an ID yet.

```
public class SyncCommand extends Command {
  // variables, constructor and other methods
    private void importContacts () throws IOException {

        for (Person person : connections) {
                String id = person.getResourceName();
                String gName = retrieveFullGName(person);
                if (!syncedIDs.contains(id)) {
                    if (!hashName.containsKey(gName)) {
                        // We import the contact if there is no contact of a similar
name
                    } else if (hashName.containsKey((gName))) {
                        seedu.address.model.person.ReadOnlyPerson aPerson =
hashName.get(gName);
                        if (equalPerson(aPerson, person)) {
                            //We link the 2 contacts if they have the same details
                        } else {
                            // We can safely import the contacts as they have
different details
                        }
                    }
                }
        }
    }

    private void exportContacts (List<ReadOnlyPerson> personList) throws Exception {
        for (ReadOnlyPerson person : personList) {
            if (person.getId().getValue().equals("")) {
                if (hashGoogleName.containsKey(person.getName().fullName)) {
                    // We can safely export the contact as there is no one with a
similar name
                } else if (hashGoogleName.containsKey(person.getName().fullName)) {
                    // We check if the person is identical, and link them if they are
                    Person gPerson = hashGoogleName.get(person.getName().fullName);
                    if (equalPerson(person, gPerson)) {
                        // We link the similar contacts
                    } else {
                        // We can safely export the contact as their other details are
not similar
                    }
                }
            }
        }
    }
}
```

3.5.3. Design Considerations

Aspect: It is difficult to keep track of which contacts have been synchronised
Alternative 1 (current choice): An unique ID is assigned to each contact, and a global syncedIDS HashTable is stored
Pros: It is easy to link/unlink synchronised contacts and keep track of the IDs that are in use.
Cons: Requires a persistent data file to be stored for synchronised IDs.

Alternative 2: Use an ID field for each contact, but keep no global data file
Pros: Less resources and room for error
Cons: Requires more time for synchronisation, and it will be difficult to remove Google Contacts that have been deleted locally.

Aspect: Authorisation cannot be performed synchronously due to the Google People library
Alternative 1 (current choice): Introduce a new command `login` which is asynchronous, while `sync` remains synchronous **Pros:** `Model` can be updated as we remain on the application thread
Cons: It is difficult to control the number of open threads, which can impact system resources, and we have to run `login` before `sync`

Alternative 2: We implement synchronous usage of the Google Peopl library
Pros: No threading and hence complication is required.
Cons: It is extremely difficult to achieve this.

3.6. 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 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` : Can continue, but with caution
- `INFO` : Information showing the noteworthy actions by the application
- `FINE` : Details that is not usually noteworthy but may be useful in debugging e.g. print the actual list instead of just its size

3.7. Configuration

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

4. 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.

4.1. 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.

4.2. Publishing documentation

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

4.3. 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 4.3.1)

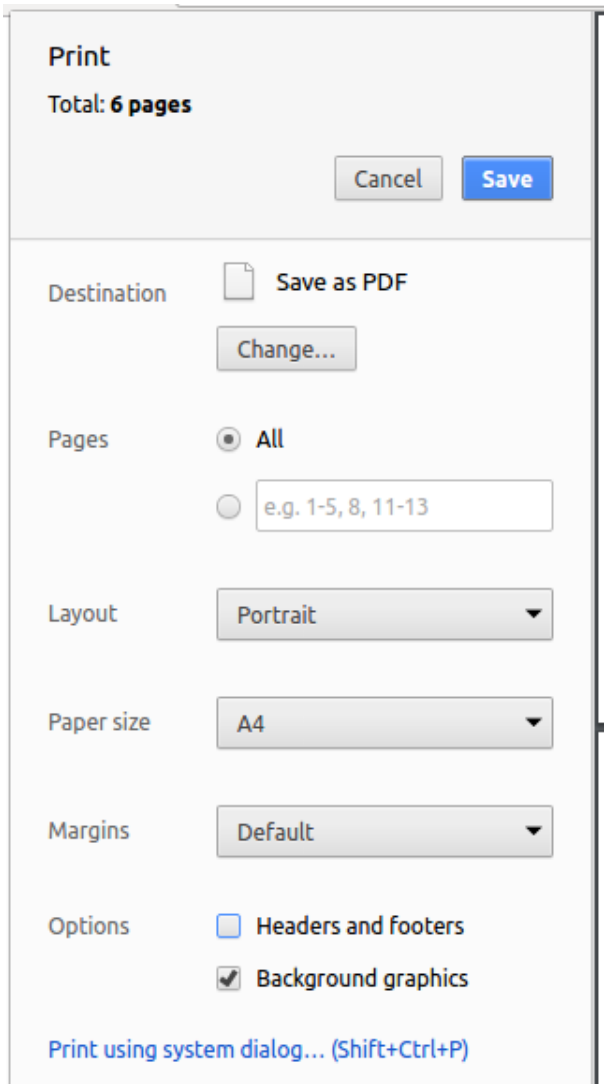

Figure 4.3.1 : Saving documentation as PDF files in Chrome

5. Testing

5.1. 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`)

5.2. 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 application 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.address.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.address.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.address.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.address.logic.LogicManagerTest`

5.3. Troubleshooting testing

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

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

6. Dev Ops

6.1. Build automation

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

6.2. Continuous Integration

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

6.3. 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.

6.4. Managing dependencies

A project often depends on third-party libraries. For example, Address Book depends on the Jackson library for XML 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: 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 application
* * *	user	add a new person	
* * *	user	delete a person	remove entries that I no longer need
* * *	user	find a person by name	locate details of persons without having to go through the entire list
* * *	user	use shorter aliases for commands	enter commands quickly and not have to type in the full command
* * *	user	find people by their tags	locate a specific group of persons
* * *	user	have a responsive inbuilt browser with similar response times to external browsers	use the inbuilt browser smoothly
* * *	user	add tags cumulatively	edit tags conveniently
* * *	user	add a person with fewer parameters	add someone I don't know all the details of
* * *	user	edit contact details	modify contacts without having to delete the contact
* * *	user	view in-line help via the help command	view the help without having to navigate the user guide(which is not CLI friendly)
* * *	user	add contacts with multiple phone numbers	have contact entries with multiple phone numbers without the need for multiple entries
* * *	user	revert to a previous version of my AddressBook	restore from a backup if my contact data is accidentally lost
* * *	user	navigate the navigation using only my keyboard (using preset keybindings)	use the application solely with my keyboard, as with CLI-focused apps

Priority	As a …	I want to …	So that I can…
* *	user	hide private contact details by default	minimize chance of someone else seeing them by accident
* *	user	have Google Contacts integration Google Contacts API	view and modify my contacts on other platforms than my computer
* *	user	have a reminder system tag to names	remember my appointments with other people
* *	user	access a person's Facebook account via in the in-built browser	use Facebook features from the AddressBook
* *	user	find a subset of contacts using specified parameters	filter through my contacts
* *	user	locate a person's address on Google Maps	easily navigate to my contact's location
* *	user	resize the dimensions of the command and output bar	customise the application to the desired layout
* *	user	upload pictures of my contacts	identify my contacts with similar names
* *	user	change the layout and enable/disable certain components e.g. the inbuilt browser	change the layout as desired and customise my AddressBook
* *	user	clear the screen to the default view	reset my AddressBook and start from a clean slate
* *	user	have a plugin manager to download and use plugins I want	only use resources I want to
* *	user	have a theme manager	change the colours to fit my desires
* *	user	modify private information	conveniently modify private information
* *	user	encrypt private information with a passphrase	secure my private information and hide it from others
* *	user	have a Favourites section where popular contacts are shown	access my frequently viewed contacts quickly
* *	user	have a settings manager/config file	customise the application and preferences

Priority	As a …	I want to …	So that I can…
* *	user	send an email via the inbuilt browser by clicking on a contact's email	easily and quickly send an email to an existing contact
* *	user	have a Notes section to add notes that attaches to a person	jot down certain events and details
* *	user who is privacy focused	encrypt my contacts	hide and secure my contacts from others
* *	user	tab-complete my commands	quickly complete my commands and do inline searching for contacts
* *	user	add aliases for contacts	label my contacts with a different name
* *	user	use regex for find command	type less and perform a wider variety of searches
*	user with many persons in the address book	sort persons by name	locate a person easily
*	user	be able to sort the contacts	look for people easily
*	user	send a message to my contacts in the AddressBook	contact people directly from the application
*	user	use the application on my phone	access contact details directly on my phone
*	user	store/see the relationship between our contacts in a graph	see our mutual friends

Appendix B: Use Cases

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

Use case: Delete person

MSS

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

   Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index is invalid.

3a1. ABC shows an error message.

Use case resumes at step 2.

Use case: Delete tag

MSS

1. User requests to delete a specific tag by name
2. ABC deletes the tag from every person in the contact list

   Use case ends.

Extensions

1a. The tag does not exist.

1a1. ABC shows an error message. Use case ends.

1b. The tag is not a valid tag.

1b1. ABC shows an error message.

Use case ends.

Use case: Edit contact details

MSS

1. User requests to edit contact
2. ABC shows a list of persons
3. User requests to edit a specific index in the list with required tags on new information
4. ABC confirms that user wishes to change data
5. User confirms the change
6. ABC changes the information in the field

   Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index is invalid.

3a1. ABC shows an error message.

Use case resumes at step 2.

3b. The user does not provide fields for new data.

3b1. ABC shows an error message.

Use case resumes at step 2.

3c. The user does not change any field.

3c1. ABC shows an error message.

Use case resumes at step 2.

5a. User inputs no.

Use case ends.

5b. User inputs something other than yes or no.

5b1. ABC shows an error message.

Use case resumes at step 4.

Use case: Add tag to contact

MSS

1. User requests to add tag to contact
2. ABC shows a list of persons
3. User requests to add tag to the person at a specific index in the list
4. ABC changes the information in the field

   Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index is invalid.

3a1. ABC shows an error message.

Use case resumes at step 2.

3b. The user does not provide a new tag.

3b1. ABC shows an error message.

Use case resumes at step 2.

3c. The user provides an invalid tag.

3c1. ABC shows an error message.

Use case resumes at step 2.

3d. The user provides a tag that already exists on the specified contact.

3d1. ABC shows an error message.

Use case resumes at step 2.

Use case: Backup data

MSS

1. User requests to backup data
2. ABC backs up the data to the hard drive

   Use case ends.

Extensions

2a. ABC fails to save the data.

2a1. ABC shows an error message.

Use case ends.

Use case: Restore backup

MSS

1. User requests to restore backup
2. ABC shows a list of backups available
3. User selects index of specific backup in the list
4. ABC confirms that user wishes to restore backup and will lose current data
5. User confirms the change
6. ABC restores to backup specified by user

Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index is invalid.

3a1. ABC shows an error message.

Use case resumes at step 2.

5a. User inputs no.

Use case ends.

5b. User inputs something other than yes or no.

5b1. ABC shows an error message.

Use case resumes at step 4.

Use case: Upload pictures

MSS

1. User requests to list persons
2. ABC shows a list of persons

3. User requests to upload a picture in a directory for a specific person in the list
4. ABC saves the picture for the person in the contact list

   Use case ends.

Extensions

   2a. The list is empty.

   Use case ends.

   3a. The given index is invalid.

      3a1. ABC shows an error message.

      Use case resumes at step 2.

   3b. The specified picture is invalid

      3b1. ABC shows an error message.

      Use case resumes at step 2.

Use case: Add remark to a person

MSS

1. User requests to list persons
2. ABC shows a list of persons
3. User requests to add remark to a person in the list
4. ABC adds remark to the person in the contact list

   Use case ends.

Extensions

   2a. The list is empty.

   Use case ends.

   3a. The given index is invalid.

      3a1. ABC shows an error message.

      Use case resumes at step 2.

Appendix C: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java `1.8.0_60` or higher installed.
2. Should be able to hold up to 1000 contacts without a noticeable sluggishness in performance for typical usage.
3. A user with above average typing speed for regular English text (i.e. not code, not system admin commands) should be able to accomplish most of the tasks faster using commands than using the mouse.
4. Should be able to respond to a command within 500ms.
5. Data should only be accessible to the user himself.
6. Should be compatible with earlier versions.
7. Should be able to handle all possible exceptions.

Appendix D: Glossary

Mainstream OS

```
Windows, Linux, Unix, OS-X
```

Private contact detail

```
A contact detail that is not meant to be shared with others
```

Google Contacts API

```
An API provided by Google for client applications to perform basic CRUD functions on a
user's contacts.
```