E.T. (Employees Tracker) - Developer Guide

By: `Team W15-B1` Since: `Feb 2018` Licence: `MIT`

1. Setting up

1.1. Prerequisites

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

NOTE	Having any Java 8 version is not enough. This app will not work with earlier versions of Java 8.

2. **IntelliJ** IDE

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

1.2. Instructions for setting up the project in your computer

1. Fork this repo, and clone the fork to your computer
2. Open IntelliJ (if you are not in the welcome screen, click `File` > `Close Project` to close the existing project dialog first)
3. Set up the correct JDK version for Gradle
   a. Click `Configure` > `Project Defaults` > `Project Structure`
   b. Click `New…` and find the directory of the JDK
4. Click `Import Project`
5. Locate the `build.gradle` file and select it. Click `OK`
6. Click `Open as Project`
7. Click `OK` to accept the default settings
8. Open a console and run the command `gradlew processResources` (Mac/Linux: `./gradlew processResources`). It should finish with the `BUILD SUCCESSFUL` message.
   This will generate all resources required by the application and tests.

1.3. Verification of the setup

1. Run the `seedu.address.MainApp` and try a few commands
2. Run the tests to ensure 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 ours but it uses a different import order from ours. To rectify,

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

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

1.4.2. Updating documentation to match your fork

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

1.4.3. Setting up CI

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

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

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

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

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

1.4.4. Getting started with coding

When you are ready to start coding,

1. Get some sense of the overall design by reading Section 2.1, “Architecture”.
2. Take a look at Appendix A, *Suggested Programming Tasks to Get Started*.

2. Design

2.1. Architecture

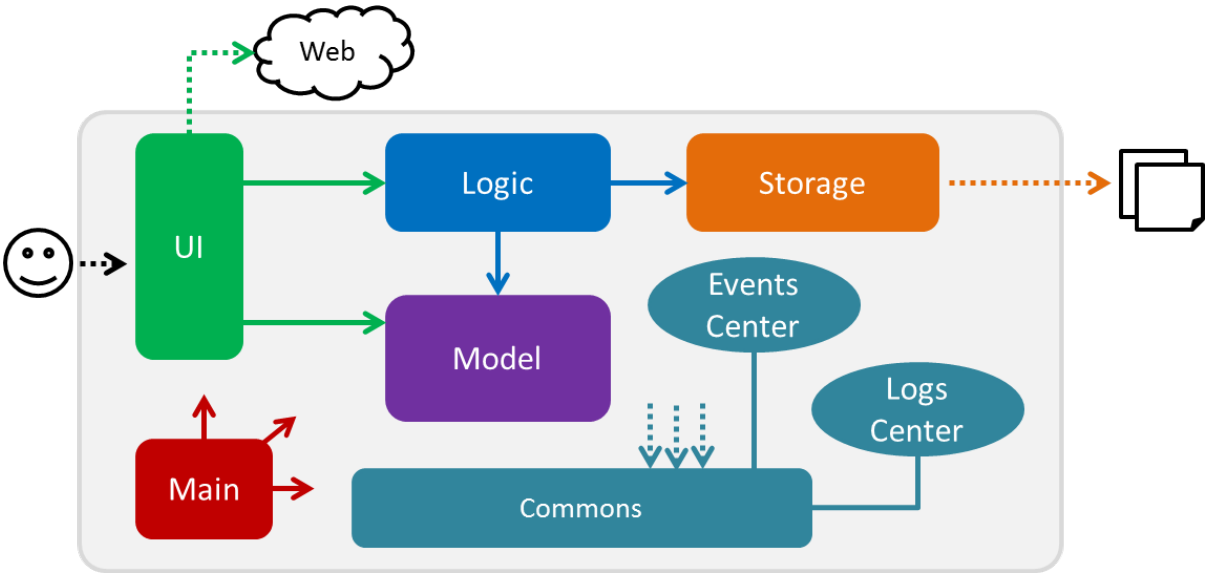


Figure 1. Architecture Diagram

The ***Architecture Diagram*** given above explains the high-level design of the App. Given below is a quick overview of each component.

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

`Main` has only one class called `MainApp`. It is responsible for,

- At app launch: Initializing the components in the correct sequence, and connecting them up with each other.
- At shut down: 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` : This class (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` : Used by many classes to write log messages to the App’s log file.

The rest of the App consists of four components.

- `UI`: The UI of the App.
- `Logic`: The command executor.

- `Model`: Holds the data of the App 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 using a `{Component Name}Manager` class.

For example, the `Logic` component (see the class diagram given below) defines it's API in the `Logic.java` interface and exposes its functionality using the `LogicManager.java` class.

[LogicClassDiagram] | *LogicClassDiagram.png*

Figure 2. Class Diagram of the Logic Component

Events-Driven nature of the design

The *Sequence Diagram* below shows how the components interact for the scenario where the user issues the command `delete 1`.

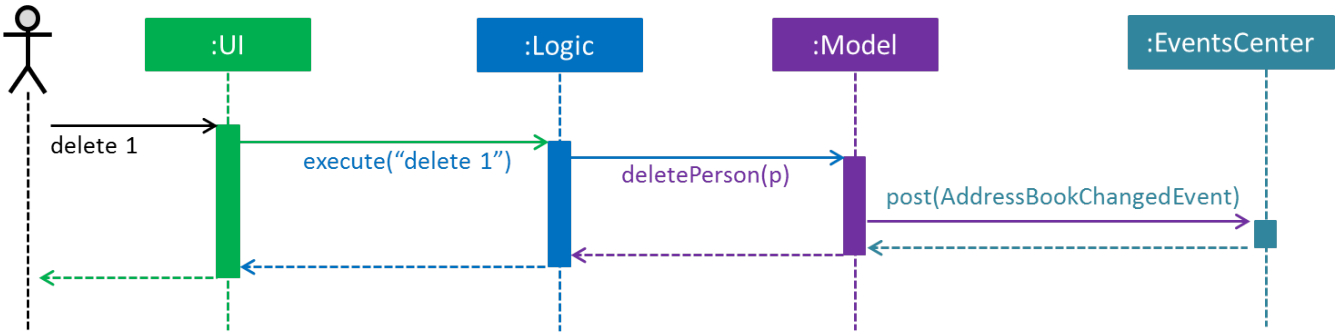


Figure 3. Component interactions for `delete 1` command (part 1)

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

The diagram below shows how the `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.

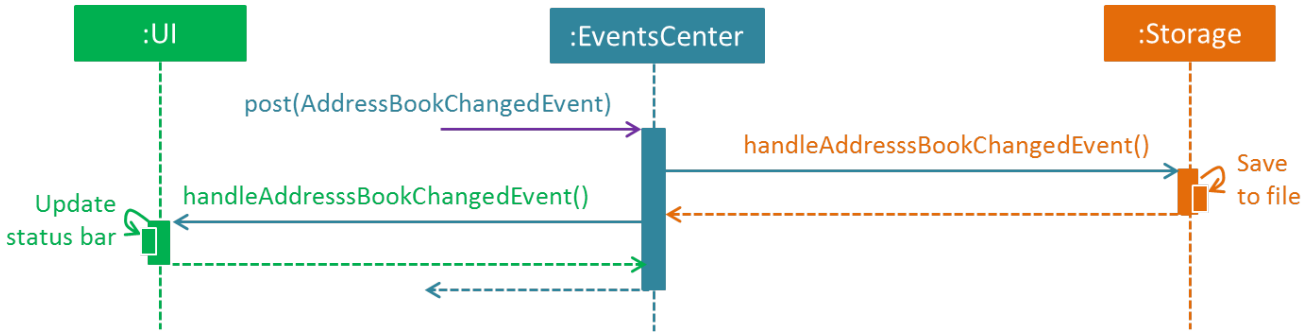


Figure 4. Component interactions for `delete 1` command (part 2)

NOTE	Note how the event is propagated through the `EventsCenter` to the `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 each component.

2.2. UI component

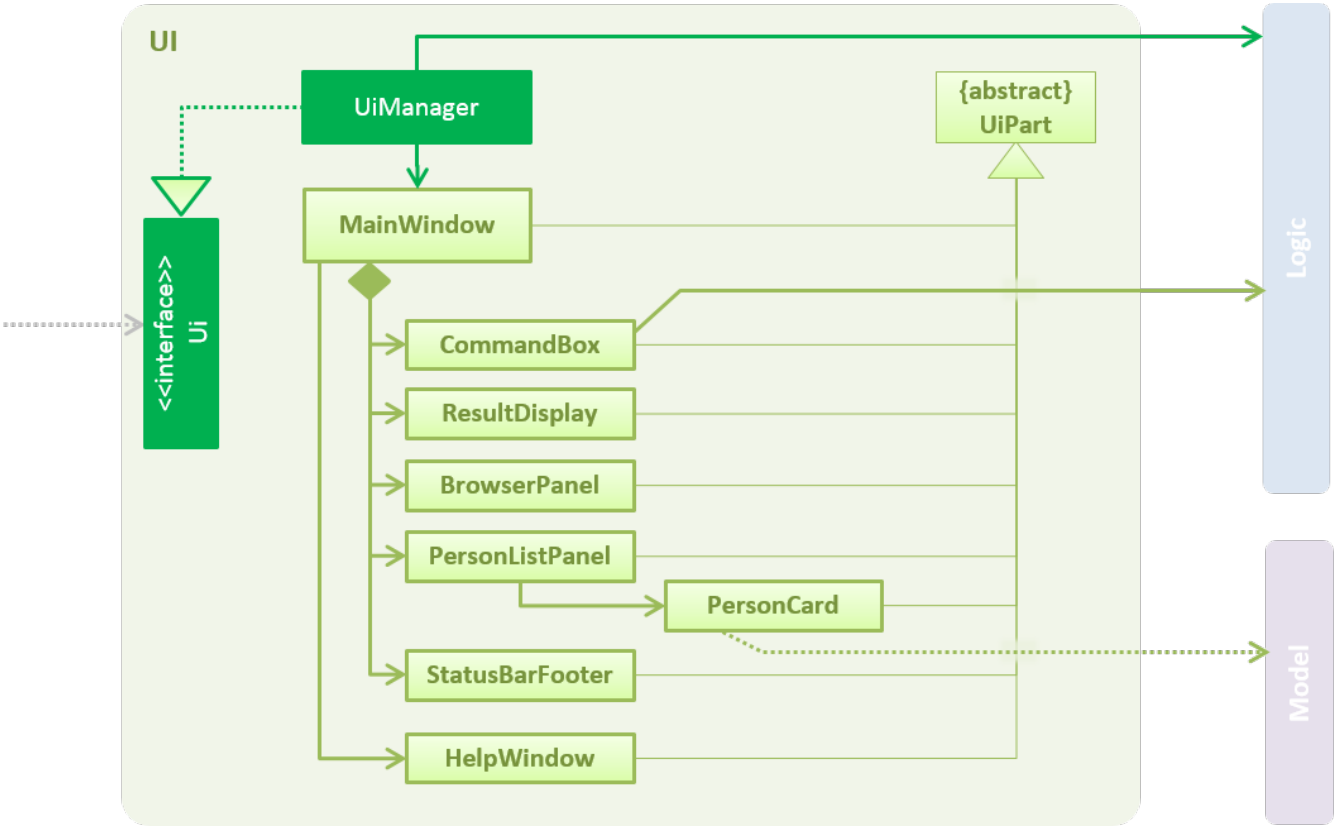


Figure 5. Structure of the UI Component

API : `Ui.java`

The UI consists of a `MainWindow` that is made up of parts e.g.`CommandBox`, `ResultDisplay`, `PersonListPanel`, `StatusBarFooter`, `BrowserPanel` etc. 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 matching `.fxml` files that are in the `src/main/resources/view` folder. For example, the layout of the `MainWindow` is specified in `MainWindow.fxml`

The `UI` component,

- Executes user commands using the `Logic` component.
- Binds itself to some data in the `Model` so that the UI can auto-update when data in the `Model` change.
- Responds to events raised from various parts of the App and updates the UI accordingly.

2.3. Logic component

[LogicClassDiagram] | *LogicClassDiagram.png*

Figure 6. Structure of the Logic Component

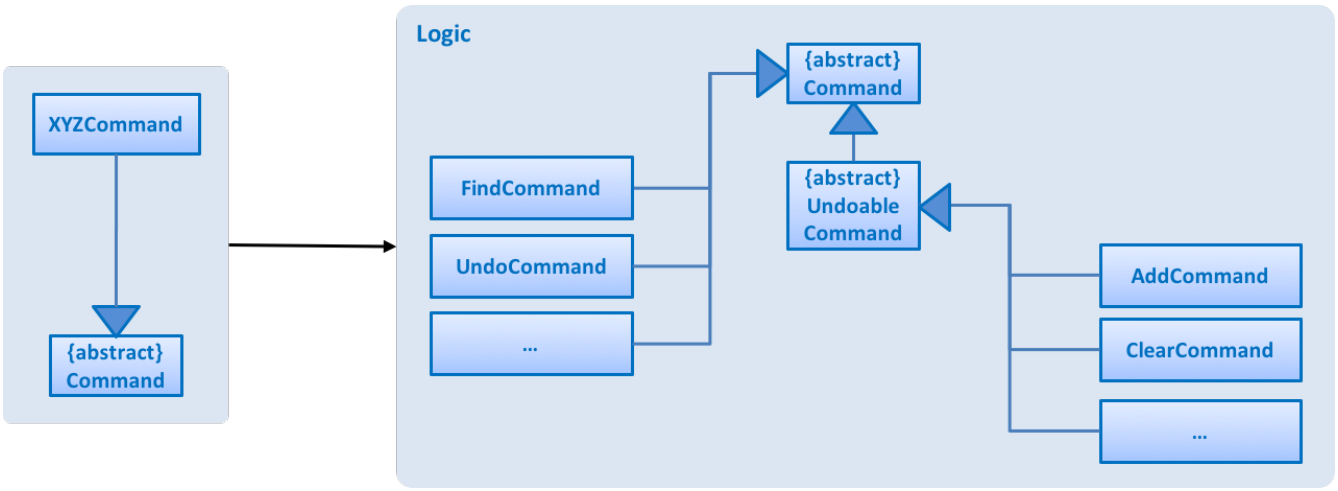


Figure 7. Structure of Commands in the Logic Component. This diagram shows finer details concerning `XYZCommand` *and* `Command` *in Figure 6, "Structure of the Logic Component"*

API : `Logic.java`

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

Given below is the Sequence Diagram for interactions within the `Logic` component for the `execute("delete 1")` API call.

[DeletePersonSdForLogic] | *DeletePersonSdForLogic.png*

Figure 8. Interactions Inside the Logic Component for the `delete 1` *Command*

2.4. Model component

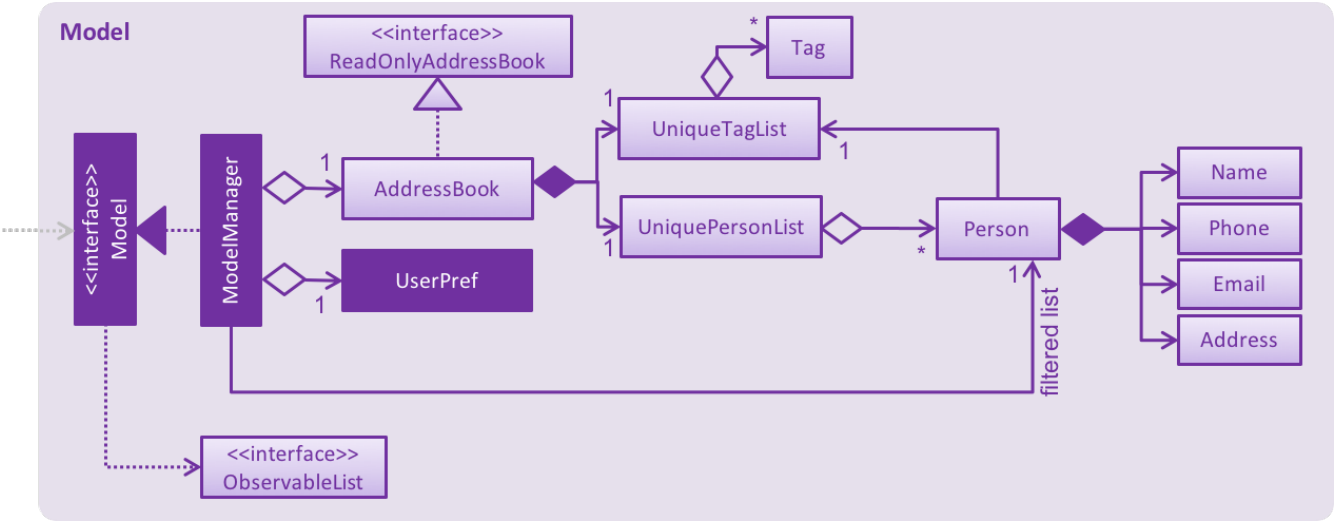


Figure 9. Structure of the Model Component

API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user's preferences.
- stores the Address Book data.
- exposes an unmodifiable `ObservableList<Person>` 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

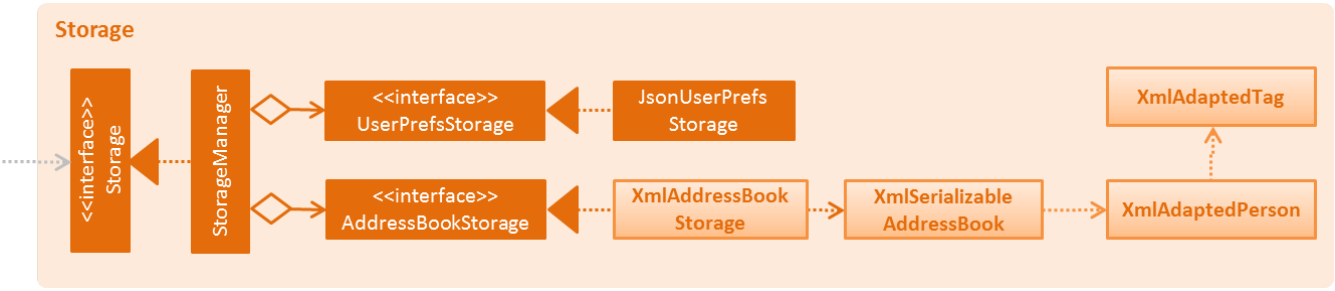


Figure 10. Structure of the Storage Component

API : `Storage.java`

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

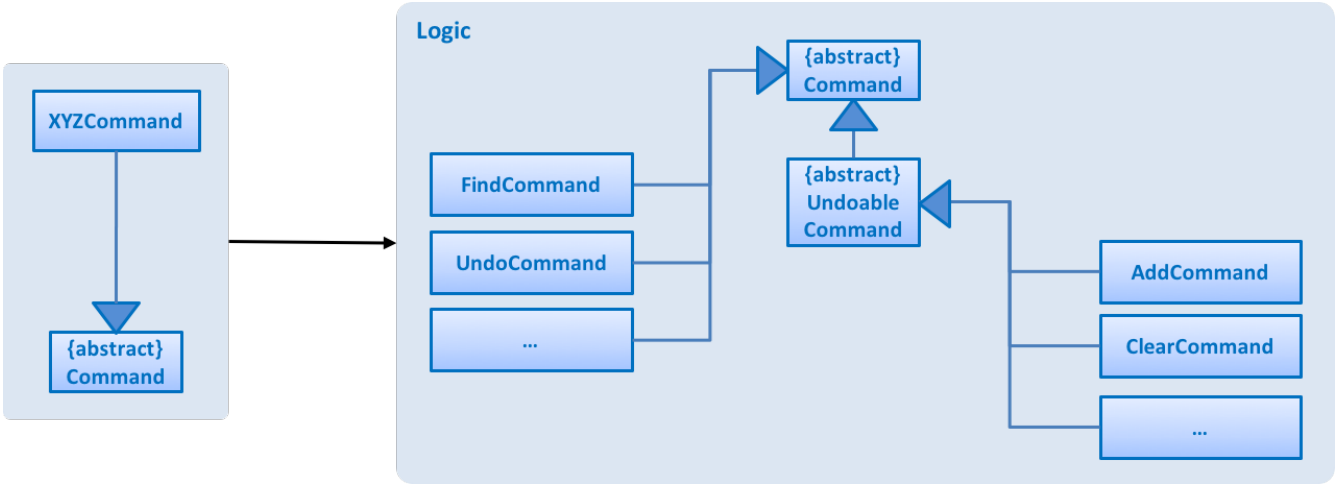
This section describes some noteworthy details on how certain features are implemented.

3.1. Undo/Redo feature

3.1.1. Current Implementation

The undo/redo mechanism is facilitated by an `UndoRedoStack`, which resides inside `LogicManager`. It supports undoing and redoing of commands that modifies the state of the address book (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 shows the inheritance diagram for commands:



As you can see from the diagram, `UndoableCommand` adds an extra layer between the abstract `Command` class and concrete commands that can be undone, such as the `DeleteCommand`. Note that extra tasks need to be done when executing a command in an *undoable* way, such as saving the state of the address book before execution. `UndoableCommand` contains the high-level algorithm for those extra tasks while the child classes implements 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 commands that are undoable are implemented this way:

```
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 beginning.

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).

[UndoRedoStartingStackDiagram] | *UndoRedoStartingStackDiagram.png*

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.

[UndoRedoNewCommand1StackDiagram] | *UndoRedoNewCommand1StackDiagram.png*

> **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 decides to undo that action using `undo`.

We will pop the most recent command out of the `undoStack` and push it back to the `redoStack`. We will restore the address book to the state before the `add` command executed.

[UndoRedoExecuteUndoStackDiagram] | *UndoRedoExecuteUndoStackDiagram.png*

> **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 shows how the undo operation works:

[UndoRedoSequenceDiagram] | *UndoRedoSequenceDiagram.png*

The redo does the exact opposite (pops from `redoStack`, push to `undoStack`, and restores the address

book to the state after the command is executed).

NOTE	If the `redoStack` is empty, then 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`. This time the `redoStack` is no longer empty. It will be purged as it no longer make sense to redo the `add n/David` command (this is the behavior that most modern desktop applications follow).

[UndoRedoNewCommand2StackDiagram] | *UndoRedoNewCommand2StackDiagram.png*

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:

[UndoRedoNewCommand3StackDiagram] | *UndoRedoNewCommand3StackDiagram.png*

The following activity diagram summarize what happens inside the `UndoRedoStack` when a user executes a new command:

[UndoRedoActivityDiagram] | *UndoRedoActivityDiagram.png*

3.1.2. Design Considerations

Aspect: Implementation of `UndoableCommand`

- **Alternative 1 (current choice):** Add a new abstract method `executeUndoableCommand()`
  - Pros: We will not lose any undone/redone functionality as it is now part of the default behaviour. Classes that deal with `Command` do not have to know that `executeUndoableCommand()` exist.
  - Cons: Hard for new developers to understand the template pattern.
- **Alternative 2:** Just 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):** Saves the entire address book.
  - Pros: Easy to implement.
  - Cons: May have performance issues in terms of memory usage.
- **Alternative 2:** Individual command knows how to undo/redo by itself.
  - Pros: Will use less memory (e.g. for `delete`, just save the person being deleted).
  - Cons: We must ensure that the implementation of each individual command are correct.

Aspect: Type of commands that can be undone/redone

- **Alternative 1 (current choice):** Only include commands that modifies the address book (`add`, `clear`, `edit`).
  - Pros: We only revert changes that are hard to change back (the view can easily be re-modified as no data are * lost).
  - Cons: User might think that undo also applies when the list is modified (undoing filtering for example), * only to realize that it does not do that, after executing `undo`.
- **Alternative 2:** Include all commands.
  - Pros: Might be more intuitive for the user.
  - Cons: User have no way of skipping such commands if he or she just want 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 to understand, who are likely to be * the new incoming developers of our project.
  - Cons: Logic is duplicated twice. For example, when a new command is executed, we must remember to update * both `HistoryManager` and `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. Manipulating the Rating field

If a `Person` is instantiated without specifying `Rating` value, he will be assigned a `null rating` (indicated by -1 currently) by default. This will be displayed as `-` to user, indicating that this Person is yet to be rated.

The `Rating` field can be manipulated by user through `edit` or `rate` command. However, the valid inputs for rating are `1, 2, 3, 4, or 5`. That said, the current implementation does not allow a user to assign `null rating` to an existing person.

3.3. Edit and rate a person

`edit` command and `rate` command are implemented in a similar manner. They both involve modifying the field(s) of a person. The only difference is that `rate` can only change the `Rating` field, but `edit` can be used to change any field.

The implementation of `edit` and `rate` command in the Logic component involves 4 objects:

1. Person toEdit
2. Person edited
3. Parser: EditCommandParser or RateCommandParser
4. EditPersonDescriptor

Since all fields of the `Person` class are immutable, we need to use `edited` Person to replace the `toEdit` Person

The details of implementation are as follow:

1. `Parser` extracts the new information for each field from user input.
2. `EditPersonDescriptor` is used to record which field(s) will be changed and the respective new value.
3. `edited` Person will be created, by obtaining the new value for modified fields from `EditPersonDescriptor`. The value of unmodified field(s) are obtained from `toEdit` Person.
4. `edited` Person is used to replace `toEdit` Person in `AddressBook`

3.4. Review system

A `Review` consists of the *review* content itself and the *reviewer*. *Review* and *reviewer* are separated by a single newline character.

A person by default upon creation will have a list of `Review`-s with one null `Review` indicated to be `-` for both *reviewer* and *review* by default. This indicate that the person is yet to be reviewed.

A person can be assigned a `Review` through the command of `review`. `Review` does not have any restriction on the *review* content (alphanumeric and symbols without any length restriction).

3.5. Review a person

Currently `review` command and `rate` command are implemented in a similar manner, and hence `review` command and `edit` command are also implemented in a similar manner.

The implementation basically mirrors what has already been documented in `3.3. Editing and rating a person`, so the implementation below is basically the iteration of the said part.

The implementation of `edit` and `review` command in the Logic component involves 4 objects:

1. Person toEdit
2. Person edited
3. Parser: ReviewCommandParser
4. EditPersonDescriptor

Since all fields of the `Person` class are immutable, we need to use `edited` Person to replace the `toEdit` Person

The details of implementation are as follow:

1. `Parser` extracts the new information for each field from user input.
2. `EditPersonDescriptor` is used to record which field(s) will be changed and the respective new value.
3. `edited` Person will be created, by obtaining the new value for modified fields from `EditPersonDescriptor`. The value of unmodified field(s) are obtained from `toEdit` Person.
4. `edited` Person is used to replace `toEdit` Person in `AddressBook`

`Review` uses JavaFX's `Dialog` to get the review input from the user instead of from command box.

3.6. Lock and unlock the application

`lock` command and `unlock` command are implemented in a similar manner. Both have the same command format.

The implementation of `lock` and `unlock` command in the Logic component involves 5 objects:

1. Command: LockCommand
2. Command: UnlockCommand
3. CommandParser: LockCommandParser or UnlockCommandParser
4. AddressBookParser
5. LogicManager

The details of implementation of `lock` command are as follow:

1. `AddressBookParser` is used to let the application accept `lock` command.
2. `LockCommandParser` extracts the password from user input.
3. `LockCommand` is used to set the password in LogicManager.
4. `LogicManager` is used to decide whether the application is locked or not, and decide the logic flow accordingly.

The details of implementation of `unlock` command are similar to `lock` command and are as follow:

1. `AddressBookParser` is used to let the application accept `unlock` command.
2. `UnlockCommandParser` extracts the password from user input.
3. `UnlockCommand` is used to compare the user input password to the password stored in LogicManager by last `lock` caommand, if they are the same, unlock the application, otherwise, inform incorrect password.
4. `LogicManager` is used to decide whether the application is locked or not, and decide the logic flow accordingly.

3.7. Assign a timetable to every employee

To implement the timetables for employees, we use a third party source which is Google Calendar API. With this API, we can integrate the application with Google Calendar, and achieve things liking creating events as what we usually do on Google Calendar website through the command. The import of Google Calendar API is simple: just add the necessary dependencies in Build.gradle and which will import the external libraries after building.

After we have the API, what we need to do is just automatically creating a new timetable (calendar) for every employee at the time this employee was added to the application. And this is what the CreateNewCalendar class for.

Additionally, as everyone has their own unique timetables, a new field called `CalendarId` will be created for each employee, to indicate the associated timetables.

3.8. Add events on anyone's timetable

`addEvent` command is used to add an event on one employee's timetable. The implementation mainly touches 3 objects in the logic component:

1. Command: TestAddEventCommand
2. CommandParser: TestAddEventCommandParser
3. AddressBookParser

The details of implementation of `addEvent` command are as follow:

1. `AddressBookParser` is used to let the application accept `addEvent` command.
2. `TestAddEventCommandParser` is used to extracts information of the event to be added from user input.
3. `TestAddEventCommand` is used to perform the addition of the event to one's timetable with Google Calendar API.

3.9. Show notifications about expired events

This section discusses about the implementation of the notification feature of Employee Tracker.

NOTE	All usages of the word `Event` in this section refers to the class `com.google.api.services.calendar.model.Event`, not to be confused with Event classes in commons package which will be referred in their full name, e.g. BaseEvent, AddressBookChangedEvent.

3.9.1. Current Implementation

In the current implementation, the `AddressBook` class saves a list of `Notification`.

When adding a new `Notification`:

1. An `Event` is created.
2. `ModelManager` to add the `Notification` into the `Notifications` list in `AddressBook`.
3. `Model Manager` raises a `AddressBookChangedEvent` and `NotificationAddedEvent` after adding the Notification.
4. `AddressBookChangedEvent` and `NotificationAddedEvent` are handled:
    - `Storage Manager` handles the `AddressBookChangedEvent` and saves the new list of Notifications.
    - `Logic Manager` handles the `TimetableEntryAddedEvent` by adding a new `TimerTask` into its HashMap of `scheduledTimerTasks`.

When showing a `Notification Card` in UI:

1. The `TimerTask` associated with the `Notification` expires.
2. `LogicManager` raises a `ShowNotificationEvent` and `RequestToDeleteTimeTableEvent`.
3. UiManager handles the `ShowNotificationEvent` by showing the notification to user.
4. `ModelManager` handles the `RequestToDeleteTimeTableEvent` by removing the corresponding `Notification` from the list of Notifications in `AddressBook`.
5. `Model Manager` raises a `AddressBookChangedEvent` and `NotificationAddedEvent` after adding the Notification.
6. `Storage Manager` handles the `AddressBookChanged` event and saves the new list of Notifications.

NOTE	Each `Event` is assigned (and thus) to an employee. In the following discussions, we will use the phrase `owner` to refer to the employee who is assigned to the `Event`.

In order to support `email` and `whatsapp` command, as well as displaying the `owner` 's name on the `Notification Card`, we need access some information of its `owner`. This is done through the `searchEmployeeById` mechanism:

1. `addressBook` object has a `nextId` integer field.
2. Whenever an employee is added, `addressBook` assigns the `nextId` to him and increment the `nextId` field.
3. The `Person` class has an `id` field to store the id.
4. When a `Notification` is created, it has a `ownerId` field that stores the `id` of its `owner`.
5. When the `TimerTask` associated with the `TimetableEntry` expires, it will extract the name (and other fields) of the owner using the `getNameById()` method in `ModelManager`.

3.9.2. Design Considerations

There are a couple of aspects with regards to `Notification` that can be implemented in alternative designs.

Aspect: extracting a Person's information

- **Alternative1 (current choice)**: `searchEmployeeById` mechanism

    - Pros: provides the latest information even if the `Person` is edited after the `Notification` is created.
    - Cons: requires `id` field to be implemented in `Person` and `AddressBook` class, provides wrong information if `id` is not implemented correctly (e.g. repeated id)
- **Alternative2**: storing the Person's Information in `Notification` class
    - Pros: has a simpler implementation
    - Cons: provides wrong information if `Person` 's information is edited after `Notification` is created, requires to update `Notification` when `Person` is edited, increases coupling

Aspect: storing `Event` locally

- **Alternative1 (current choice)**: stores a list of `Notifications` in `AddressBook`
    - Pros: ensures that notification feature remains working when there's no internet access, saves storage space as only essential information is saved
    - Cons: has a complicated implementation (creation of `Notification` class and edit of `Person` and `AddressBook` class for `searchEmployeeById` mechanism)
- **Alternative2** : stores a list of `Event` in `AddressBook`
    - Pros: has a simpler implementation (doesn't need to implement `Notification` class)
    - Cons: requires another mechanism for extracting `ownerName` because `Event` class does not store `ownerName`, waste of storage space because `Event` class contains a lot of information unnecessary for notification feature
- **Alternative3** : stores a list of `Notifications` in `Person`
    - Pros: has a simpler implementation (doesn't need to implement `searchEmployeeById` mechanism)
    - Cons: compromises performance, especially when there are a lot of employees and only a few `Notification`
- **Alternative4** : does not store `Event` locally (relies on Google calendar service)
    - Pros: has the simplest implementation (only need to pull data from Google, does not need to create local class), saves storage space as nothing is stored locally
    - Cons: compromises notification feature if there's no internet access, requires another mechanism for extracting `ownerName` because `Event` class does not store `ownerName`, compromises performance if internet speed is slow)

3.10. Sort existing employees

The implementation of `sort` command involves following objects:

1. Command: SortCommand
2. Parser: SortCommandParser, AddressBookParser
3. AddressBook, UniqueEmployeeList

The details of implementation of `sort` command are as follow:

1. SortCommandParser guarantees Employees Tracker to accept sort command.
2. SortCommandParser extracts the sorting field from user input.
3. ModelManager and UniqueEmployeeList provide sorting algorithm, called by AddressBook.
4. UniqueEmployeeList will sort the units by giving field

3.11. Change theme of Employees Tracker

changeTheme command is used to change the theme to dark or bright

- changeTheme is achieved by handling changeThemeEvent in MainWindow
- a new "theme" field is added in GuiSettings

3.12. Logging

We are using java.util.logging package for logging. The LogsCenter class is used to manage the logging levels and logging destinations.

- The logging level can be controlled using the logLevel setting in the configuration file (See Section 3.13, “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 App
- FINE : Details that is not usually noteworthy but may be useful in debugging e.g. print the actual list instead of just its size

3.13. Configuration

Certain properties of the application can be controlled (e.g App 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.

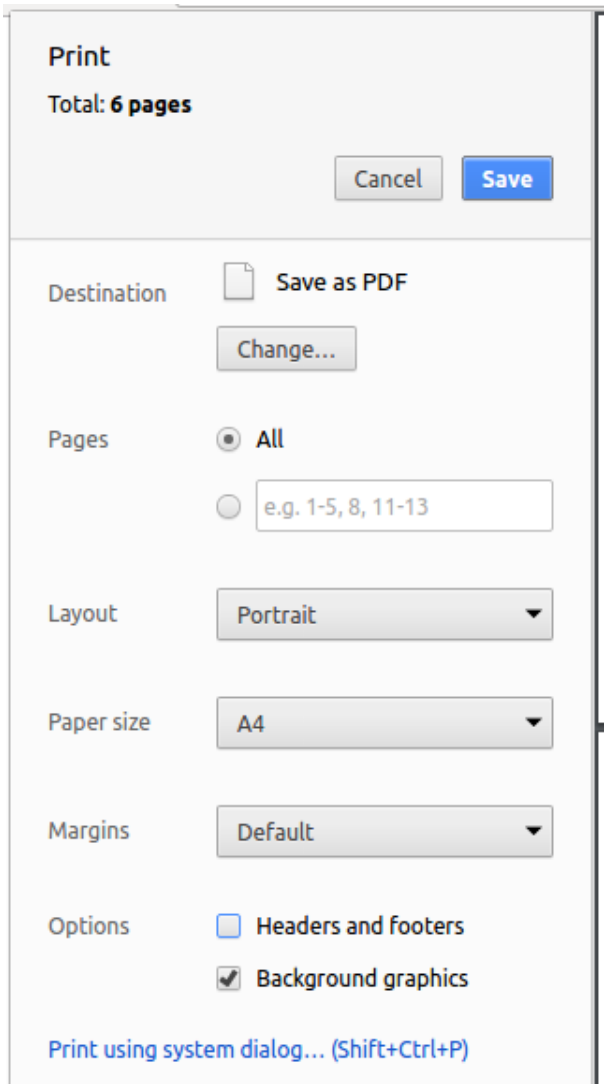


Figure 11. Saving documentation as PDF files in Chrome

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 App by simulating user actions on the GUI. These are in the `systemtests` package.
    b. *Unit tests* that test the individual components. These are in `seedu.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. Coverage Reporting

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

6.4. Documentation Previews

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

6.5. Making a Release

Here are the steps to create a new release.

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

6.6. 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: Suggested Programming Tasks to Get Started

Suggested path for new programmers:

1. First, add small local-impact (i.e. the impact of the change does not go beyond the component) enhancements to one component at a time. Some suggestions are given in Section A.1, “Improving each component”.
2. Next, add a feature that touches multiple components to learn how to implement an end-to-end feature across all components. Section A.2, “Creating a new command: `remark`” explains how to go about adding such a feature.

A.1. Improving each component

Each individual exercise in this section is component-based (i.e. you would not need to modify the other components to get it to work).

`Logic` component

Scenario: You are in charge of `logic`. During dog-fooding, your team realize that it is troublesome for the user to type the whole command in order to execute a command. Your team devise some strategies to help cut down the amount of typing necessary, and one of the suggestions was to implement aliases for the command words. Your job is to implement such aliases.

> **TIP** Do take a look at Section 2.3, "Logic component" before attempting to modify the `Logic` component.

1. Add a shorthand equivalent alias for each of the individual commands. For example, besides typing `clear`, the user can also type `c` to remove all persons in the list.

    - Hints
        - Just like we store each individual command word constant `COMMAND_WORD` inside `*Command.java` (e.g. `FindCommand#COMMAND_WORD`, `DeleteCommand#COMMAND_WORD`), you need a new constant for aliases as well (e.g. `FindCommand#COMMAND_ALIAS`).
        - `AddressBookParser` is responsible for analyzing command words.
    - Solution
        - Modify the switch statement in `AddressBookParser#parseCommand(String)` such that both the proper command word and alias can be used to execute the same intended command.
        - Add new tests for each of the aliases that you have added.
        - Update the user guide to document the new aliases.
        - See this PR for the full solution.

`Model` component

Scenario: You are in charge of `model`. One day, the `logic`-in-charge approaches you for help. He wants to implement a command such that the user is able to remove a particular tag from everyone in the address book, but the model API does not support such a functionality at the moment. Your job is to implement an API method, so that your teammate can use your API to implement his command.

> **TIP** Do take a look at Section 2.4, "Model component" before attempting to modify the `Model` component.

1. Add a `removeTag(Tag)` method. The specified tag will be removed from everyone in the address

book.

- Hints
  - The `Model` and the `AddressBook` API need to be updated.
  - Think about how you can use SLAP to design the method. Where should we place the main logic of deleting tags?
  - Find out which of the existing API methods in `AddressBook` and `Person` classes can be used to implement the tag removal logic. `AddressBook` allows you to update a person, and `Person` allows you to update the tags.
- Solution
  - Implement a `removeTag(Tag)` method in `AddressBook`. Loop through each person, and remove the `tag` from each person.
  - Add a new API method `deleteTag(Tag)` in `ModelManager`. Your `ModelManager` should call `AddressBook#removeTag(Tag)`.
  - Add new tests for each of the new public methods that you have added.
  - See this PR for the full solution.
    - The current codebase has a flaw in tags management. Tags no longer in use by anyone may still exist on the `AddressBook`. This may cause some tests to fail. See issue `#753` for more information about this flaw.
    - The solution PR has a temporary fix for the flaw mentioned above in its first commit.

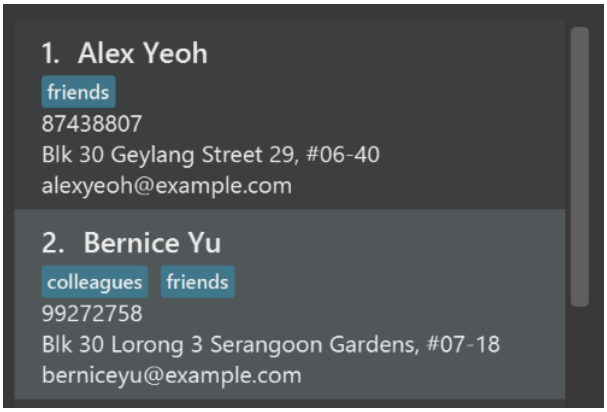
`Ui` component

Scenario: You are in charge of `ui`. During a beta testing session, your team is observing how the users use your address book application. You realize that one of the users occasionally tries to delete non-existent tags from a contact, because the tags all look the same visually, and the user got confused. Another user made a typing mistake in his command, but did not realize he had done so because the error message wasn't prominent enough. A third user keeps scrolling down the list, because he keeps forgetting the index of the last person in the list. Your job is to implement improvements to the UI to solve all these problems.

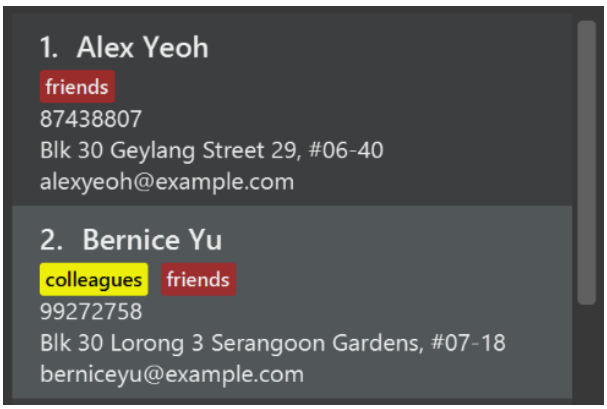
> **TIP** Do take a look at Section 2.2, "UI component" before attempting to modify the `UI` component.

1. Use different colors for different tags inside person cards. For example, `friends` tags can be all in brown, and `colleagues` tags can be all in yellow.

   Before



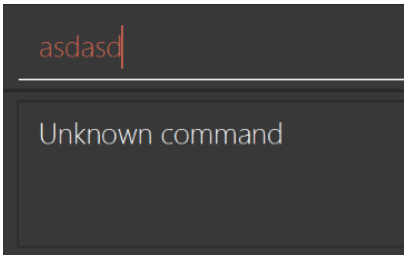
After



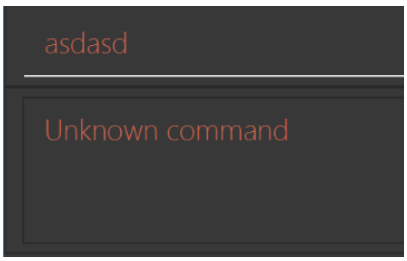
- Hints
  - The tag labels are created inside the `PersonCard` constructor (`new Label(tag.tagName)`). JavaFX's `Label` class allows you to modify the style of each Label, such as changing its color.
  - Use the .css attribute `-fx-background-color` to add a color.
  - You may wish to modify `DarkTheme.css` to include some pre-defined colors using css, especially if you have experience with web-based css.
- Solution
  - You can modify the existing test methods for `PersonCard` 's to include testing the tag's color as well.
  - See this PR for the full solution.
    - The PR uses the hash code of the tag names to generate a color. This is deliberately designed to ensure consistent colors each time the application runs. You may wish to expand on this design to include additional features, such as allowing users to set their own tag colors, and directly saving the colors to storage, so that tags retain their colors even if the hash code algorithm changes.

2. Modify `NewResultAvailableEvent` such that `ResultDisplay` can show a different style on error (currently it shows the same regardless of errors).

   Before



After



- Hints
  - `NewResultAvailableEvent` is raised by `CommandBox` which also knows whether the result is a success or failure, and is caught by `ResultDisplay` which is where we want to change the style to.
  - Refer to `CommandBox` for an example on how to display an error.
- Solution
  - Modify `NewResultAvailableEvent` 's constructor so that users of the event can indicate whether an error has occurred.
  - Modify `ResultDisplay#handleNewResultAvailableEvent(NewResultAvailableEvent)` to react to this event appropriately.
  - You can write two different kinds of tests to ensure that the functionality works:
    - The unit tests for `ResultDisplay` can be modified to include verification of the color.
    - The system tests `AddressBookSystemTest#assertCommandBoxShowsDefaultStyle()` and `AddressBookSystemTest#assertCommandBoxShowsErrorStyle()` to include verification for `ResultDisplay` as well.
  - See this PR for the full solution.
    - Do read the commits one at a time if you feel overwhelmed.

3. Modify the `StatusBarFooter` to show the total number of people in the address book.

Before

After

- Hints
  - `StatusBarFooter.fxml` will need a new `StatusBar`. Be sure to set the `GridPane.columnIndex` properly for each `StatusBar` to avoid misalignment!
  - `StatusBarFooter` needs to initialize the status bar on application start, and to update it accordingly whenever the address book is updated.
- Solution
  - Modify the constructor of `StatusBarFooter` to take in the number of persons when the application just started.
  - Use `StatusBarFooter#handleAddressBookChangedEvent(AddressBookChangedEvent)` to update the number of persons whenever there are new changes to the addressbook.
  - For tests, modify `StatusBarFooterHandle` by adding a state-saving functionality for the total number of people status, just like what we did for save location and sync status.
  - For system tests, modify `AddressBookSystemTest` to also verify the new total number of persons status bar.
  - See this PR for the full solution.

`Storage` component

Scenario: You are in charge of `storage`. For your next project milestone, your team plans to implement a new feature of saving the address book to the cloud. However, the current implementation of the application constantly saves the address book after the execution of each command, which is not ideal if the user is working on limited internet connection. Your team decided that the application should instead save the changes to a temporary local backup file first, and only upload to the cloud after the user closes the application. Your job is to implement a backup API for the address book storage.

TIP Do take a look at Section 2.5, "Storage component" before attempting to modify the `Storage` component.

1. Add a new method `backupAddressBook(ReadOnlyAddressBook)`, so that the address book can be saved in a fixed temporary location.

- Hint
  - Add the API method in `AddressBookStorage` interface.
  - Implement the logic in `StorageManager` and `XmlAddressBookStorage` class.
- Solution
  - See this PR for the full solution.

A.2. Creating a new command: `remark`

By creating this command, you will get a chance to learn how to implement a feature end-to-end, touching all major components of the app.

Scenario: You are a software maintainer for `addressbook`, as the former developer team has moved on to new projects. The current users of your application have a list of new feature requests that they hope the software will eventually have. The most popular request is to allow adding additional comments/notes about a particular contact, by providing a flexible `remark` field for each contact, rather than relying on tags alone. After designing the specification for the `remark` command, you are convinced that this feature is worth implementing. Your job is to implement the `remark` command.

A.2.1. Description

Edits the remark for a person specified in the `INDEX`.
Format: `remark INDEX r/[REMARK]`

Examples:

- `remark 1 r/Likes to drink coffee.`
  Edits the remark for the first person to `Likes to drink coffee.`
- `remark 1 r/`
  Removes the remark for the first person.

A.2.2. Step-by-step Instructions

[Step 1] Logic: Teach the app to accept 'remark' which does nothing

Let's start by teaching the application how to parse a `remark` command. We will add the logic of `remark` later.

Main:

1. Add a `RemarkCommand` that extends `UndoableCommand`. Upon execution, it should just throw an `Exception`.
2. Modify `AddressBookParser` to accept a `RemarkCommand`.

Tests:

1. Add `RemarkCommandTest` that tests that `executeUndoableCommand()` throws an Exception.
2. Add new test method to `AddressBookParserTest`, which tests that typing "remark" returns an instance of `RemarkCommand`.

[Step 2] Logic: Teach the app to accept 'remark' arguments

Let's teach the application to parse arguments that our `remark` command will accept. E.g. `1 r/Likes to drink coffee.`

Main:

1. Modify `RemarkCommand` to take in an `Index` and `String` and print those two parameters as the error message.
2. Add `RemarkCommandParser` that knows how to parse two arguments, one index and one with prefix 'r/'.
3. Modify `AddressBookParser` to use the newly implemented `RemarkCommandParser`.

Tests:

1. Modify `RemarkCommandTest` to test the `RemarkCommand#equals()` method.
2. Add `RemarkCommandParserTest` that tests different boundary values for `RemarkCommandParser`.
3. Modify `AddressBookParserTest` to test that the correct command is generated according to the user input.

[Step 3] Ui: Add a placeholder for remark in `PersonCard`

Let's add a placeholder on all our `PersonCard` s to display a remark for each person later.

Main:

1. Add a `Label` with any random text inside `PersonListCard.fxml`.
2. Add FXML annotation in `PersonCard` to tie the variable to the actual label.

Tests:

1. Modify `PersonCardHandle` so that future tests can read the contents of the remark label.

[Step 4] Model: Add `Remark` class

We have to properly encapsulate the remark in our `Person` class. Instead of just using a `String`, let's follow the conventional class structure that the codebase already uses by adding a `Remark` class.

Main:

1. Add `Remark` to model component (you can copy from `Address`, remove the regex and change the names accordingly).
2. Modify `RemarkCommand` to now take in a `Remark` instead of a `String`.

Tests:

1. Add test for `Remark`, to test the `Remark#equals()` method.

[Step 5] Model: Modify `Person` to support a `Remark` field

Now we have the `Remark` class, we need to actually use it inside `Person`.

Main:

1. Add `getRemark()` in `Person`.
2. You may assume that the user will not be able to use the `add` and `edit` commands to modify the remarks field (i.e. the person will be created without a remark).
3. Modify `SampleDataUtil` to add remarks for the sample data (delete your `addressBook.xml` so that the application will load the sample data when you launch it.)

[Step 6] Storage: Add `Remark` field to `XmlAdaptedPerson` class

We now have `Remark` s for `Person` s, but they will be gone when we exit the application. Let's modify `XmlAdaptedPerson` to include a `Remark` field so that it will be saved.

Main:

1. Add a new Xml field for `Remark`.

Tests:

1. Fix `invalidAndValidPersonAddressBook.xml`, `typicalPersonsAddressBook.xml`, `validAddressBook.xml` etc., such that the XML tests will not fail due to a missing `<remark>` element.

[Step 6b] Test: Add withRemark() for `PersonBuilder`

Since `Person` can now have a `Remark`, we should add a helper method to `PersonBuilder`, so that users are able to create remarks when building a `Person`.

Tests:

1. Add a new method `withRemark()` for `PersonBuilder`. This method will create a new `Remark` for the person that it is currently building.
2. Try and use the method on any sample `Person` in `TypicalPersons`.

[Step 7] Ui: Connect `Remark` field to `PersonCard`

Our remark label in `PersonCard` is still a placeholder. Let's bring it to life by binding it with the actual `remark` field.

Main:

1. Modify `PersonCard`'s constructor to bind the `Remark` field to the `Person` 's remark.

Tests:

1. Modify `GuiTestAssert#assertCardDisplaysPerson(…)` so that it will compare the now-functioning

remark label.

[Step 8] Logic: Implement `RemarkCommand#execute()` logic

We now have everything set up… but we still can't modify the remarks. Let's finish it up by adding in actual logic for our `remark` command.

Main:

1. Replace the logic in `RemarkCommand#execute()` (that currently just throws an `Exception`), with the actual logic to modify the remarks of a person.

Tests:

1. Update `RemarkCommandTest` to test that the `execute()` logic works.

A.2.3. Full Solution

See this PR for the step-by-step solution.

Appendix B: Product Scope

Target user profile:

- is a manger
- has a need to keep track of a significant number of subordinate employees
- need to assign work to employees and keep track of it
- prefer desktop apps over mobile apps
- can type fast, i.e. >45 words per minute
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition: keep track of employees and their work easily through a single app

Feature Contribution：

1. Li Yufei
    - (minor) be able to lock the application and unlock it and must use the same password
    - (major) each employee has his own timetable and manager can add events on anyone's timetable
2. Yang Yuqing
    - (minor) be able to sort the employees by existing field (ie. name, phone, email, address, rate)
    - (major) UI optimization
3. Ho Bing Xuan

- (minor) add Rating features
- (major) notification feature for timetable entry

4. Gilbert Emerson
    - (minor) enhance on the Find feature, able to find by multiple keyphrases and also in multiple fields
    - (major) add Review feature

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	include employee that have just entered the company
* * *	user	have a dedicated timetable for each employee	trace progress of that person
* * *	user	add a timetable entry on anyone’s timetable	assign jobs to employee
* * *	user	edit a timetable entry	update timetable entry information
* * *	user	delete a timetable entry	remove a job or event that is canceled
* * *	user	give employee a rating	evaluate their performance

Priority	As a …	I want to …	So that I can…
* * *	user	change employee’s rating	update my rating record when the performance of an employee changes
* * *	user	add notes on a timetable entry or on a person	include more details about the entry
* * *	user	search employees by specific criteria (e.g. name and tags)	search employees more easily
* * *	user	identify a person with name when carrying out commands	save time by not having to browse through a long list
* *	user	sort employees by name	locate an employee easily
* *	user	sort employees by their rating	give them bonus salaries accordingly
* *	user	sort employees by their salaries	see their salary conditions
* *	user	hide private contact details by default	minimize chance of someone else seeing them by accident
* *	user	have my own timetable	manage my own time
* *	user	start composing an email with a command	send an email to a specific person faster

Priority	As a …	I want to …	So that I can…
* *	user	mass adding timetable entry to many employees' timetable at once	save time by not having to add the event to person by person
* *	user	be notified for any deadline for the timetable entry of my employees	be aware of employees who are late in submitting their work
* *	user	export my employees tracker	share information with another user
* *	user	import my employees tracker	obtain information from another user
*	user	see timetable entries happening in other departments	be aware of the progress of other departments
*	user	view to-do-list	see my own upcoming jobs/events
*	user	export a list of people into excel sheet	do collective operations easily on other platform
*	user	login	have personalized privileges/window scheme
*	user	change the window scheme/theme/skin	have personalised experience in the app
*	user	know employee’s location	search their location on the map

Priority	As a …	I want to …	So that I can…
*	user	lock the employees tracker	leave my app open while making unauthorized people cannot access it
*	user	unlock the employees tracker	continue to use the app after leaving it locked
*	user	write a review to an employee that has ever worked below me	information his/her current manager of his/her performance

Appendix D: Use Cases

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

Use case: Add a timetable entry on anyone's timetable

MSS

1. User requests to list employees
2. shows a list of employees
3. User requests to add a entry to an employee's timetable
4. Employees Tracker adds the entry

   Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index is invalid.

3a1. Employees Tracker shows an error message.

Use case resumes at step 2.

Use case: Give employee a rating

MSS

1. User requests to list employees
2. Employees Tracker shows a list of employees
3. User requests to give an employee a rating
4. Employees Tracker add the rating

   Use case ends.

Extensions

2a. The list is empty.

Use case ends.

3a. The given index is invalid.

3a1. Employees Tracker shows an error message.

Use case resumes at step 2.

3b. The given rating is not an integer.

3b1. Employees Tracker shows an error message.

Use case resumes at step 2.

4a. The person has already been rated.

4a1. Employees Tracker updates the rating for the person with the new rating.

Use case ends.

Use case: Sort the employees by their rating

MSS

1. User requests to list employees sorted by their ratings
2. Employees Tracker accesses the list of all employees
3. Employees Tracker sorts the employees in the list by their ratings
4. Employees Tracker shows the sorted list

   Use case ends.

Extensions

2a. The list is empty.

Use case ends.

Use Case: Mass adding timetable entry to many employees' timetable at once

MSS

1. User requests to list all employees
2. Employees Tracker shows a list of all employees
3. User requests to add timetable entry to many employees
4. Employees Tracker add the entry to many employees

   Use case ends.

Extensions

   2a. The list is empty.

   Use case ends.

   3a. The index range given is invalid.

      3a1. Employees Tracker shows an error message.

      Use case resumes at step 2.

Use Case: Write a review to an employee that has ever worked below me

MSS

1. User requests to list all employees
2. Employees Tracker shows a list of all employees
3. User requests to add review to an employee
4. Employees Tracker add the review to the employee

   Use case ends.

Extensions

   2a. The list is empty.

   Use case ends.

   3a. The index given is invalid.

      3a1. Employees Tracker shows an error message.

      Use case resumes at step 2.

Use Case: Export Employees Tracker

MSS

1. User requests export Employees Tracker into a directory
2. Employees Tracker export the Employees Tracker data in a file to the directory specified

   Use case ends.

Extensions

1a. The directory is invalid.

   1a1. Employees Tracker shows an error message.

   Use case ends.

Use Case: Import Employees Tracker

MSS

1. User requests import Employees Tracker file from a directory
2. Employees Tracker import the Employees Tracker data from the file in the directory specified

   Use case ends.

Extensions

1a. The directory is invalid.

   1a1. Employees Tracker shows an error message.

   Use case ends.

Use case: edit timetable entry

MSS

1. User requests to list all persons.
2. User requests to view the timetable of a person.
3. Employees Tracker shows the timetable of that person.
4. User requests to update the information of an entry in the timetable.
5. Employees Tracker updates the new information for that timetable entry.

   Use case ends.

Extensions

1a. The list is empty.

Use case ends.

2a. The given index is invalid.

2a1. Employees Tracker shows an error message.

Use case resumes at step 2.

2b. The person has no timetable entry.

Use case ends.

4a. The event name provided by user doesn’t exist.

4a1. Employees Tracker shows an error message

Use case resumes at step 2.

4b. The new information given by user doesn’t follow command format

4b1. Employees Tracker shows an error message

Use case resumes at step 2.

5a. Employees Tracker couldn’t write to save file.

5a1. Employees Tracker shows error message and requests user to resolve the error.

5a2. User resolves the error.

Use case resumes at step 5.

Use case: delete a timetable entry

MSS

1. User requests to list all persons.
2. User requests to view the timetable of a person.
3. Employees Tracker shows the timetable of that person.
4. User requests to delete an entry in that person’s timetable.
5. Employees Tracker deletes the timetable entry.

Use case ends.

Extensions

1a. The list is empty.

Use case ends.

2a. The given index is invalid.

2a1. Employees Tracker shows an error message.

Use case resumes at step 2.

2b. The person has no timetable entry.

Use case ends.

4a. The event name provided by user doesn't exist.

4a1. Employees Tracker shows an error message

Use case resumes at step 2.

5a. Employees Tracker couldn't write to save file.

5a1. Employees Tracker shows error message and requests user to resolve the error.

5a2. User resolves the error.

Use case resumes at step 5.

Use case: start composing email

MSS

1. User requests to list all persons.
2. User requests to email a person in the list.
3. Employees Tracker opens up a webpage for composing email to that person.

Use case ends.

Extensions

1a. The list is empty.

Use case ends.

2a. The given index is invalid.

2a1. Employees Tracker shows an error message.

Use case resumes at step 2.

3a. The computer has no access to internet.

3a1. Employees Tracker shows error message.

Use case ends.

Use case: export a list of people as Excel spreadsheet

MSS

1. User requests to list all persons, or perform a search.
2. User requests to export the list of persons as excel sheet and save it in a save file path.
3. Employees Tracker exports the list of persons as excel sheet.

   Use case ends.

Extensions

1a. The list is empty.

Use case ends.

2a. The given save file path is invalid.

2a1. Employees Tracker shows an error message.

Use case resumes at step 2.

3a. Employees Tracker couldn’t write to save file.

3a1. Employees Tracker shows error message and requests user to resolve the error.

3a2. User resolves the error.

Use case ends.

Use case: login

MSS

1. User starts the Employees Tracker program.
2. Employees Tracker requests user to enter username and password.
3. User enters his username and password
4. Employees Tracker shows the content.

   Use case ends.

Extensions

3a. The user enters an invalid username or wrong password.

3a1. Employees Tracker shows an error message.

Use case resumes at step 1.

Use case: Lock the employees tracker

MSS

1. User requests to lock the employees tracker by entering "lock" and password
2. Employees Tracker is locked unless user unlocks it.

    2a. When employees tracker is locked, user are required to unlock the employees tracker before any instruction.

    Use case ends.

Use case: Unlock the employees tracker

MSS

1. User requests to unlock the employees tracker by entering "unlock" and the password set earlier
2. Employees Tracker unlocked. Use case ends.

Extensions

1a. The given password is incorrect, which means different from the one set earlier

    1a1. Employees Tracker shows an error message and requires to re-enter password

    Use case resumes at step 1

Use case: Change an employee’s rating

MSS

1. User requests to list employees
2. Employees Tracker shows a list of employees
3. User requests to change the rating of one of the employees from the list
4. Employees Tracker changes the rating of the employee.

Use case ends.

Extensions

3a. The employee does not exist in employees tracker.

    3a1. Employees Tracker shows an error message.

    Use case resumes at step 2.

3b. The rate is out of bound.

    3b1. Employees Tracker shows an error message

Use case resumes at step 2.

Use case: Change the window scheme/theme/skin

MSS

1. User requests to change the scheme/theme/skin on the User Interface
2. Employees Tracker changes the scheme/theme/skin

   Use case ends.

Appendix E: 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 persons without a noticeable sluggishness (i.e. response time > 500ms) in performance for typical usage.
3. A user with above average typing speed (i.e. ≥ 45 words per minute) 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. The app should be used only by one user.
5. Only the owner can view the data in the app
6. The app should be able to apply to people who are generally managing ≤ 200 other people
7. Should not consume memory more than 2GB
8. Should be able to be run easily by non-technical user
9. The save file of the app should be cross-compatible
10. The app should be used only by one user.
11. Only the owner can view the data in the app
12. The app should be able to response the command within 500ms
13. The app should be able to apply to people who are generally managing ≤ 200 other people
14. Users should prefer typing over mouse input or other input methods.
15. Users should be comfortable using CLI apps.
16. The app should not have flow flaws when running.
17. The app may utilise third party libraries, API and plug-ins.
18. The app should be able to access the Internet.
19. The app should be able to perform basic commands without internet access.
20. This application should work well both on 32-bit and 64-bit environments.
21. Should be easy to use by new users
22. This application should be stable and maintainable

23. This application is open source

Appendix F: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

Private contact detail

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

Appendix G: Product Survey

Product Name

Author: …

Pros:

- …
- …

Cons:

- …
- …

Appendix H: Instructions for Manual Testing

Given below are instructions to test the app manually.

NOTE	These instructions only provide a starting point for testers to work on; testers are expected to do more *exploratory* testing.

H.1. Launch and Shutdown

1. Initial launch
   a. Download the jar file and copy into an empty folder
   b. Double-click the jar file
      Expected: Shows the GUI with a set of sample contacts. The window size may not be optimum.
2. Saving window preferences
   a. Resize the window to an optimum size. Move the window to a different location. Close the window.

b. Re-launch the app by double-clicking the jar file.
   Expected: The most recent window size and location is retained.

{ more test cases … }

H.2. Deleting a person

1. Deleting a person while all persons are listed
   a. Prerequisites: List all persons using the `list` command. Multiple persons in the list.
   b. Test case: `delete 1`
      Expected: First contact is deleted from the list. Details of the deleted contact shown in the status message. Timestamp in the status bar is updated.
   c. Test case: `delete 0`
      Expected: No person is deleted. Error details shown in the status message. Status bar remains the same.
   d. Other incorrect delete commands to try: `delete`, `delete x` (where x is larger than the list size) *{give more}*
      Expected: Similar to previous.

{ more test cases … }

H.3. Saving data

1. Dealing with missing/corrupted data files
   a. *{explain how to simulate a missing/corrupted file and the expected behavior}*

{ more test cases … }