# NeoXPro Manager - Developer Guide

By: `Team W10-B4` Since: `Jun 2016` Licence: `MIT`

# 1. Introduction

## 1.1. Software Information

NeoXPro is a command line interface(CLI) address book. NeoXPro not only allows users to keep track of the personal information of their friends and acquaintances, it helps users to manage their everyday tasks too. The software consists of four components: UI, Logic, Model, Storage. The language used in implementing this software is JAVA.

## 1.2. Purpose of Developer Guide

Do not worry if you have no idea about how to start. This developer guide will help new team members like you to understand how NeoXPro works. You will definitely find clues about how to contribute to NeoXPro after thorough reading of this document. You can always refer to this document if you have any doubts when implementing new features to NeoXPro.

## 1.3. Content Preview

This developer guide will first bring you through the design and internal structure of NeoXPro. You will be able to understand the how four components are linked and inter-related. There will be explanation on the enhancements implemented too. You can study on those implemented features and at the same time, try to understand the logic behind every implementation. This will help when you are implementing your own new features. After implementing, you will need to update relevant documentations and running various tests to check that your codes work fine. The documentation and testing section will guide you through those processes. Read this document now to understand the best address book, NeoXPro.

# 2. Setting Up

## 2.1. Prerequisites

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

| | |
|---|---|
| **NOTE** | Having any Java 8 version is not enough.<br>This app will not work with earlier versions of Java 8. |

2. **IntelliJ** IDE

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

## 2.2. Setting Up the Project in Your Computer

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.

## 2.3. Verifying the Setup

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

## 2.4. Configurations to do Before Writing Code

### 2.4.1. Configuring the Coding Style

This project follows 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.

### 2.4.2. 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.

### 2.4.3. Setting Up CI

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

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

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

### 2.4.4. Getting Started with Coding

When you are ready to start coding,

1. Get some sense of the overall design by reading the Architecture section.
2. Take a look at the section Suggested Programming Tasks to Get Started.

# 3. Design

## 3.1. Architecture

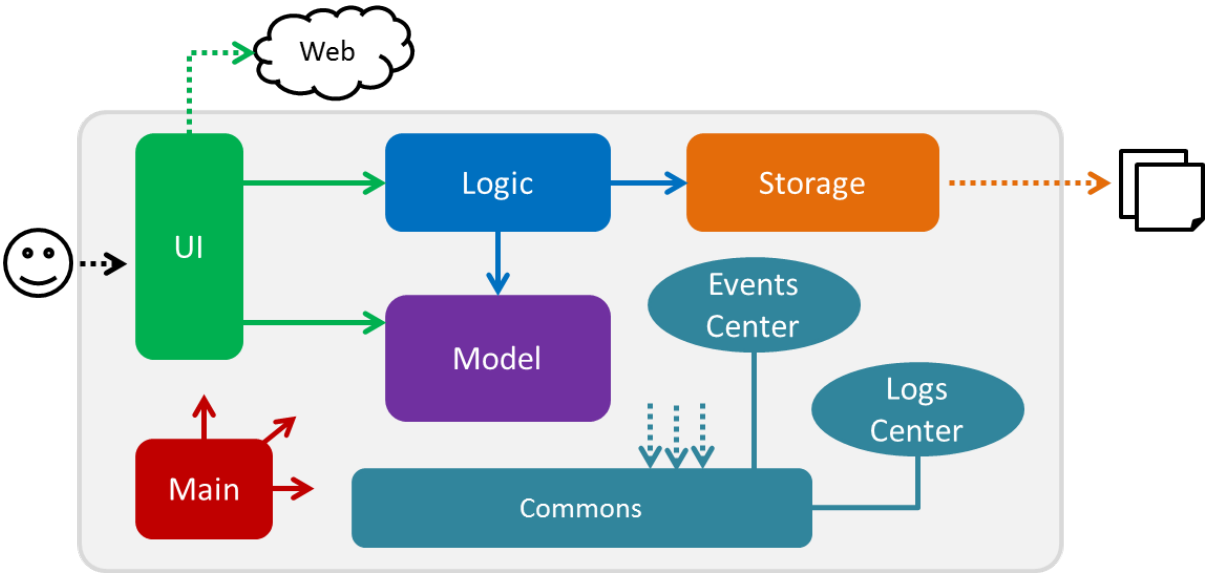



*Figure 2.1.1 : Architecture Diagram*

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: Initializes the components in the correct sequence, and connects them up with each other.
- At shut down: Shuts down the components and invokes cleanup method where necessary.

`Commons` represents a collection of classes used by multiple other components. 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 2.1.2 : 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`.

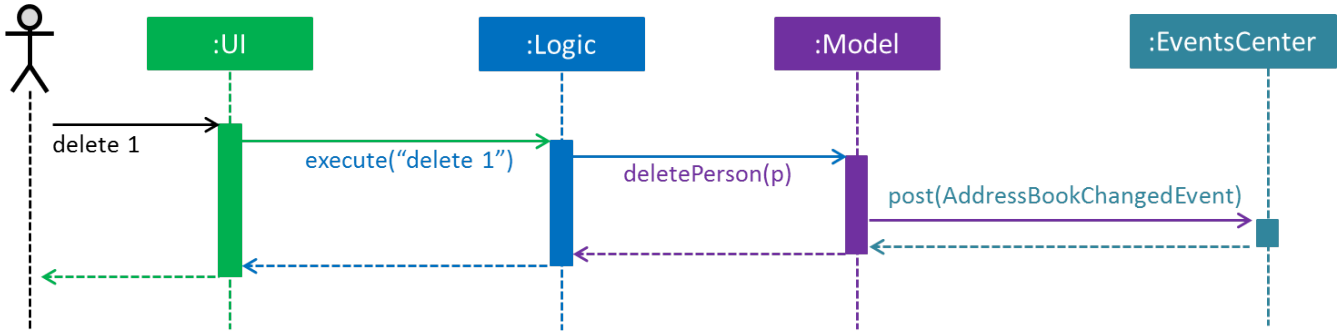



*Figure 2.1.3a : Component interactions for `delete 1` command (part 1)*

| 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.

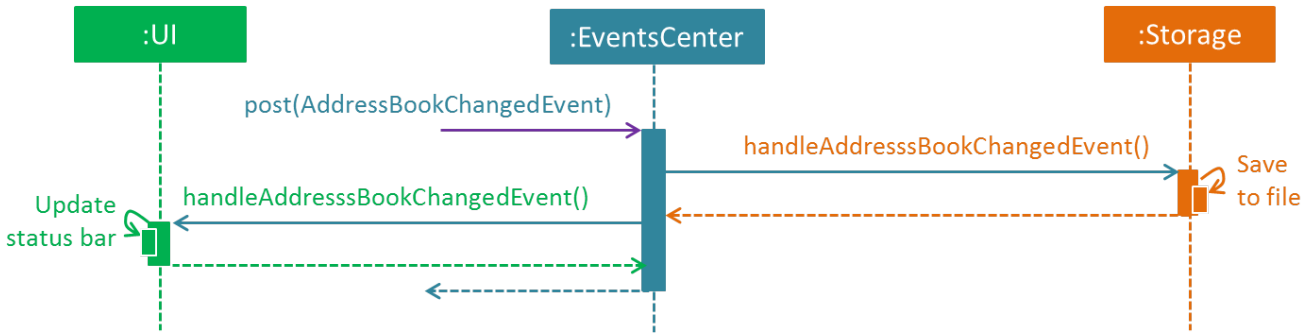



*Figure 2.1.3b : Component interactions for `delete 1` command (part 2)*

| 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.

# 3.2. UI Component

[UiClassDiagram] | *UiClassDiagram.png*

*Figure 2.2.1 : 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.

## 3.3. Logic Component

[LogicClassDiagram] | *LogicClassDiagram.png*

*Figure 2.3.1 : Structure of the Logic Component*

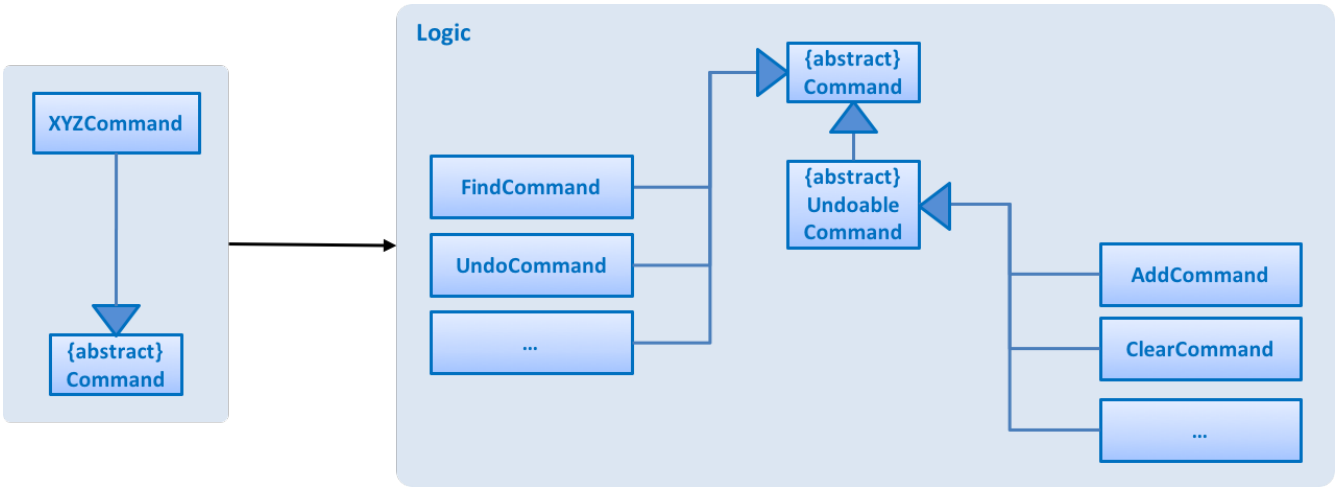



*Figure 2.3.2 : Structure of Commands in the Logic Component. This diagram shows finer details concerning `XYZCommand` and `Command` in Figure 2.3.1*

**API** : `Logic.java`

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 2.3.1 : Interactions Inside the Logic Component for the `delete 1` Command*

## 3.4. Model Component

[ModelClassDiagram] | *ModelClassDiagram.png*

*Figure 2.4.1 : 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<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.

## 3.5. Storage Component

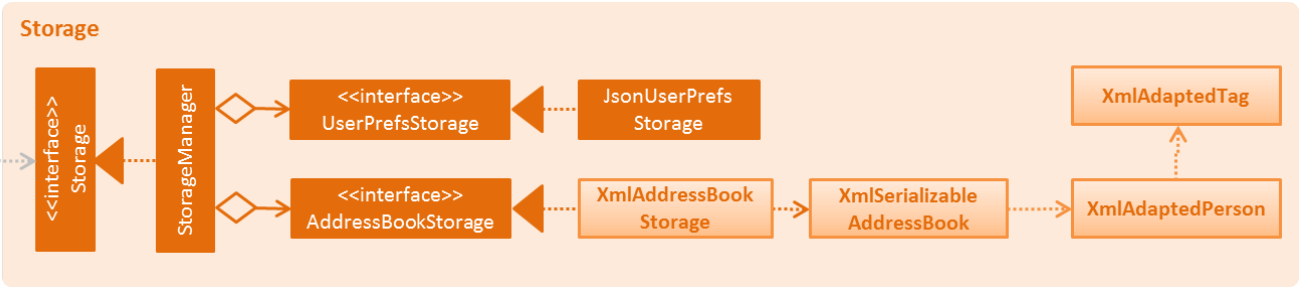



*Figure 2.5.1 : Structure of the Storage Component*

**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.

## 3.6. Common Classes

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

# 4. Implementation

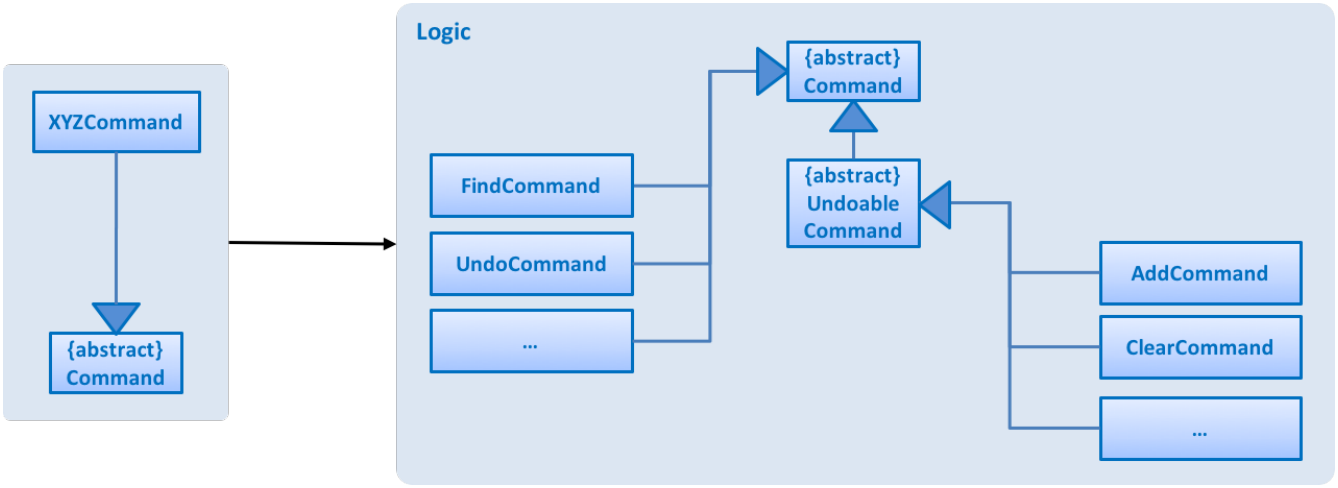
This section describes some noteworthy details on how certain features are implemented.

## 4.1. Undo/Redo Mechanism

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*

### 4.1.1. 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.

---

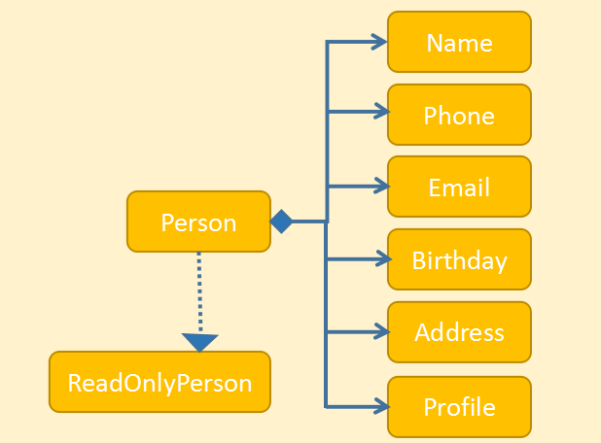
**Aspect:** Data structure to support the undo/redo commands
**Alternative 1 (current choice):** Use separate stack for undo and redo
**Pros:** Easy to understand for new Computer Science student undergraduates to understand, who are likely to be the new incoming developers of our project.
**Cons:** 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.

# 4.2. Adding a Parameter in Add Command

## 4.2.1. Add Birthday

To add a new birthday in AddCommand, UI, Logic, Model and Storage components are to be modified accordingly.
Be aware of the inheritance between each component when implementing. Refer to Section 3 to observe the relationship between those components.

- `UI Component:`
  PersonCard uses ReadOnlyPerson to create labels of most parameters. New parameter: Birthday, should be added to be included so that it can be shown in the information of person when showing person card in UI.
- `Logic Component:`
  To include birthday parameter, several files in logic components should be modified accordingly. E.g. AddCommand, EditCommand, AddCommandParser, CliSyntax etc. Commands allows users to input the information, provides an interaction between user and the software. Parser is to parse the new parameter information, making sure that information is present before the real add command is created. Information being added should be parse to model to create persons if command format is correct.
- `Model Component:`
  Create a new class for birthday so that it can be used under Person class. Person class uses ReadOnlyPerson interface. Person Class is to collect and set all information about a particular person. If all parameters is parsed correctly, a person with collected information will be created. The following diagram shows the relationship of different classes in model component:




- `Storage Component:`
  To make sure that birthday can be stored in the xml format for future usage, xmlAdaptedPerson is used to save information of a person in xml format.

### 4.2.2. Add Profile Page

The profile page parameter is facilitated by the class `ProfilePage`.
In order to add this parameter profile page, we make modifications to UI, Logic, Model and Storage components.

- `UI Component:`
  `PersonCard`, which resides in UI component, frist creates a label for profile page parameter:

```
@FXML
    private Label profile;
```

Then `PersonCard` binds the label with the value of `ProfilePage` object. If the input value for `ProfilePage` object is null, it make the label dissapear from UI by setting its visibility to FALSE:

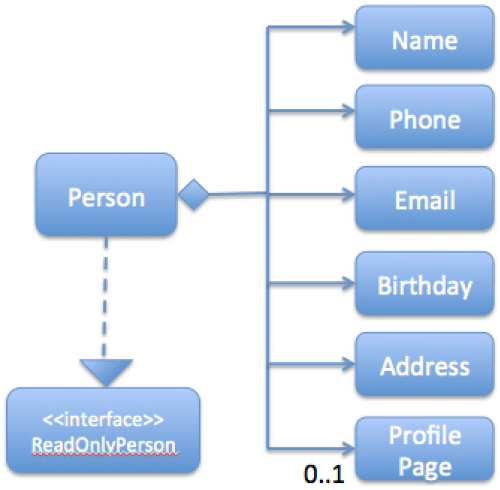
```
private void bindListeners(ReadOnlyPerson person) {
    //... binding other labels ...

    if (!person.profilepageProperty().toString().equals("")) {
        profile.textProperty().bind(Bindings.convert(person.profilepageProperty()));
        profile.setVisible(true);
    } else {
        profile.setVisible(false);
    }
    //... binding other labels ...
}
```

In order to make `select` command load a person's `ProfilePage` object if it exists, we modify method `handlePersonPanelSelectionChangedEvent()` in `BrowserPanel` that is in charge of updating a person panel:

```
 private void handlePersonPanelSelectionChangedEvent(PersonPanelSelectionChangedEvent
event) {
     //...
     if (person.getProfilePage().hasProfilePage()) {
         loadProfilePage(person);
     } else {
         loadPersonPage(person);
     }
 }
```

- `Logic Component:`
  The prefix for profile page `pr/` is added to `CliSyntax`. Then the following files `AddCommand`, `EditCommand`, `AddCommandParser` and `EditCommandParser` are modified so that `add` and `edit` commands accept the new parameter profile page.
  `AddCommandParser` does not check for profile page prefix `pr/` as this is an optional parameter.
- `Model Component:`
  Class `ProfilePage` is used to store profile page property of class `Person`. Class `Person`, which resides in model component and implements `ReadOnlyPerson` interface, form a composition association with `ProfilePage` .
  As profile page is an optional parameter, a `Person` can be linked to 0 or 1 `ProfilePage` object.
  The relationship is illustrated in the following diagram:




- `Storage Component:`
  `xmlAdaptedPerson` file is used to save information of a person in xml format.
  The `required` parameter of `@XmlElement` element which stores profile page information is set to false to make this property optional:

```
@XmlElement(required = false)
    private String profile = "";
```

## 4.3. Logging

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

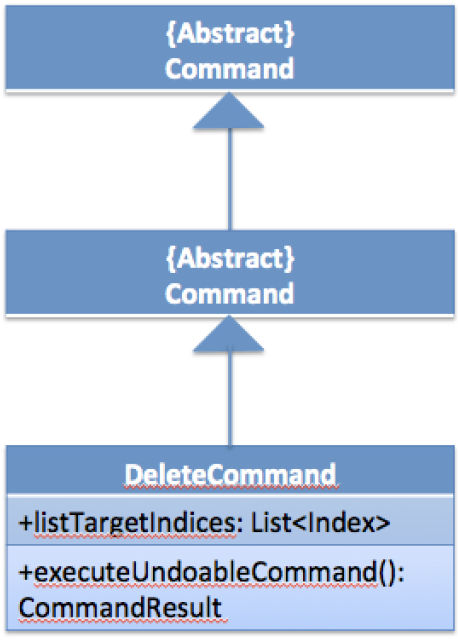
- The logging level can be controlled using the `logLevel` setting in the configuration file (See Configuration)
- The `Logger` for a class can be obtained using `LogsCenter.getLogger(Class)` which will log messages according to the specified logging level
- Currently log messages are output through: `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 App
- `FINE` : Details that is not usually noteworthy but may be useful in debugging e.g. print the actual list instead of just its size

## 4.4. Delete command

`delete` command supports modifying the state of address book by deleting all persons whose indices are specified in the input. It inherits from `UndoableCommand`.




The implementation of `delete` contains 2 classes: `DeleteCommand` and `DeleteCommandParser` inside the logic component.
`DeleteCommandParser`, the parser of `delete`, parses user's input into the variable `input: List<Index>` that store a list of `Index`. `DeleteCommand`, which handles the logic of `delete` command, then iteratively remove any `Person` object with `Index` specified in `input`.

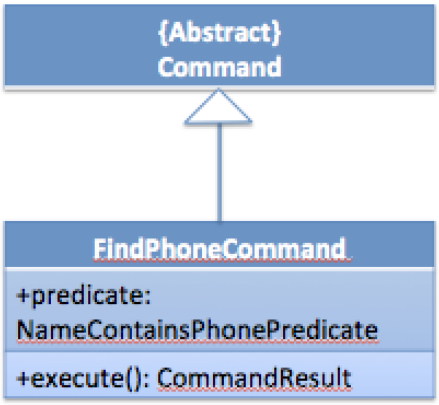
```
public class DeleteCommandParser implements Parser<DeleteCommand> {
    public DeleteCommand parse(String args) throws ParseException {
        try {
            List<Index> input= new ArrayList<Index>();
            // ... Parser logic ...
            return new DeleteCommand(input);
        } catch (IllegalValueException ive) {
            // throw exception here
        }
    }
}
```

And finally, we add the `delete` command to the class 'AddressBookParser' so that `delete` command is recognized whenever invoked.

## 4.5. Phone command

The `phone` command utilize the same implementation as the `Find` command for name. Instead of logic execute search via `Name` attribute of `Person`, the command search for the `phone` attribute.

The `phone` command is handled by the class `FindPhoneCommand` that inherits from `Command` class.
`Name` and `Phone` API structure is roughly similar that they allow to extract value of the object. The search algorithm utilizes a class `NameContainsPhonesPredicate implements Predicate<ReadOnlyPerson>` which allows the algorithm to use Java `Predicate` class method.
The diagram demonstrating `phone` command structure is illustrated here:




## 4.6. FindTag Command

The findTag command utilize the same implementation as the Find command for name. Instead of logic execute search via `Name` attribute of Person, the command search in the `TagList` attribute.

Tag component in model already had API methods for searching, which is similar to `Name` which allows extracting value of the object. The search algorithm utilize a class `TagContainsKeywordsPredicate implements Predicate<ReadOnlyPerson>` which allows the algorithm to use Java `Predicate` class method.

## 4.7. (Task) event management

The model for task event is implemented similar to `person` class.

[EventModelClassDiagram] | *EventModelClassDiagram.png*

The `EventList` is then linked up to the `AddressBook` which is monitored by `ModelManager` to reflect any change to the `Addressbook` model. The Ui components `EventCard` and `EventListPanel` are also hooked up to the model and reflect any changes made to `filterEvents`.

## 4.8. Event auto-reminder

When the `UiManager` component got initialized at the start of the application, a new `ShowReminderRequestEvent` gets posted. This is then handle by the `MainWindow` which initialized one or more `ReminderWindow` from the `upcomingEvents` in the model.

## 4.9. Coming Birthday List

Basically, adding a list of upcoming birthday is only related to UI component. A new class should be implemented to hold the list. In this case, ComingBirthdayListPanel is implemented. ComingBirthdayListPanel should process all contacts in the person list and decide which persons to include in the coming birthday list.
To get the list of upcoming birthdays, the ComingBirthdayListPanel should have a method comingBirthdayListGetter which is implemented in this way:

```
private ObservableList<ReadOnlyPerson>
comingBirthdayListGetter(ObservableList<ReadOnlyPerson> personList) {
        List<ReadOnlyPerson> comingBirthdayList =
personList.stream().collect(Collectors.toList());
        boolean isRemoved = false;
        Calendar cal = Calendar.getInstance();
        int month = cal.get(Calendar.MONTH)+1;
        int date = cal.get(Calendar.DATE);
        //...
        for (int i = 0; i < comingBirthdayList.size(); i++) {
            if (!(Integer.parseInt(comingBirthdayList.get(i).getBirthday().toString()
                    .substring(5, 7)) == month)) {
                comingBirthdayList.remove(i);
                isRemoved = true;
            }
            else
if((Integer.parseInt(comingBirthdayList.get(i).getBirthday().toString()
                    .substring(5, 7)) == month) &&

Integer.parseInt(comingBirthdayList.get(i).getBirthday().toString()
                    .substring(8)) < date) {
                comingBirthdayList.remove(i);
                isRemoved = true;
            }
            //...
        }
        return FXCollections.observableArrayList(comingBirthdayList);
}
```

Main window should include the newly added ComingBirthdayListPanel class so that the panel will be shown in the main window. A fxml file, in this case is ComingBirthdayListPanel.fxml for coming birthday list should be present to reserve a section for the list in the main window UI. MainWindow.fxml should modified accordingly to include the ComingBirthdayListPanel.fxml.

# 4.10. Favorite/Unfavorite a person using FavoriteCommand.

## 4.10.1. FavoriteCommand/Unfavorite Command

To Favorite/Unfavorite a person, UI, Logic, Model and Storage components are to be modified accordingly.
Be aware of the inheritance between each component when implementing. Refer to Section 3 to observe the relationship between those components.

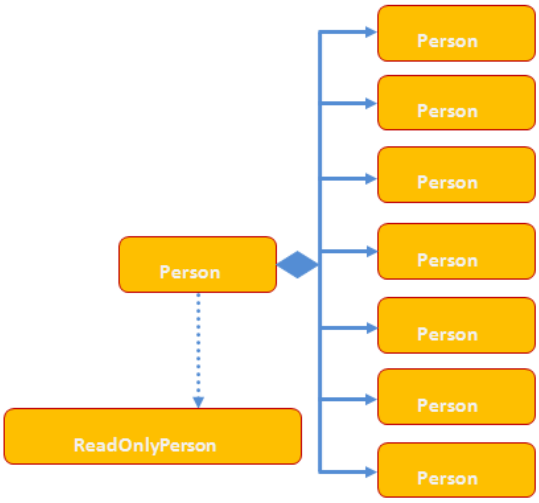
- `UI Component:`
  PersonCard uses ReadOnlyPerson to create labels of most parameters, and it uses an imageView to record the "Favorite" parameter.
  When a person is favorited, a star would appear next to the person name, and it would

disappear if he/she is unfavorited.
Different from other parameters, since 'favorite' is based on a boolean value, instead of using bindListener, A new 'star' image is set inside the ImageView if it is meant to appear and it is set to null when it is meant to disappear.

This uniqueness is demonstrated as below:

```
        if (person.getFavorite().value == true) {
            Image image = new Image("/images/star.png");
            favorite.setImage(image);
        } else {
            favorite = null;
        }
```

- `Logic Component:`
  To include the Favorite parameter, several files in logic components should be modified accordingly. E.g. AddCommand, EditCommand, AddCommandParser, etc. Parser is to parse the new parameter information, making sure that information is present before the real add command is created. Information being added should be parse to model to create persons if command format is correct.
- `Model Component:`
  Create a new class for Favorite so that it can be used under Person class. Person class uses ReadOnlyPerson interface. Person Class is to collect and set all information about a particular person. If all parameters is parsed correctly, a person with collected information will be created. The following diagram shows the relationship of different classes in model component:




- `Storage Component:`
  To make sure that favorite value can be stored in the xml format for future usage, xmlAdaptedPerson is used to save information of a person in xml format.

List all your favorite persons using ListFavoriteCommand.

### 4.10.2. ListFavoriteCommand

To list all your favorite persons in AddressBook, UI, Model, and Logic Components are involved. Logic Component includes `ListFavoriteCommand` and `ListFavoriteCommandParser`, and they call `FavoritePredicate` in Model Component to check whether a person is favorited or not, UI component then decide which persons to include in the PersonListPanel.

## 4.11. Export Command

`export` access the file `addressbook.xml` to retrieve each person and his/her properties and write it on the file at the specified input. Its implementation contains 2 classes: `ExportCommand` and `ExportCommandParser`.
The parser `ExportCommandParser` of `export` parses the file path input as a `String` to `ExportCommand`.
`ExportCommand`, which handles the logic of `export` command, then writes the exported file on the specified input file path.

## 4.12. Configuration

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

# 5. Documentation

We use asciidoc for writing documentation.

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

## 5.1. Editing Documentation

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

## 5.2. Publishing Documentation

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

## 5.3. Converting Documentation to PDF Format

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

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

1. Follow the instructions in UsingGradle.adoc to convert the AsciiDoc files in the `docs/` directory

to HTML format.

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

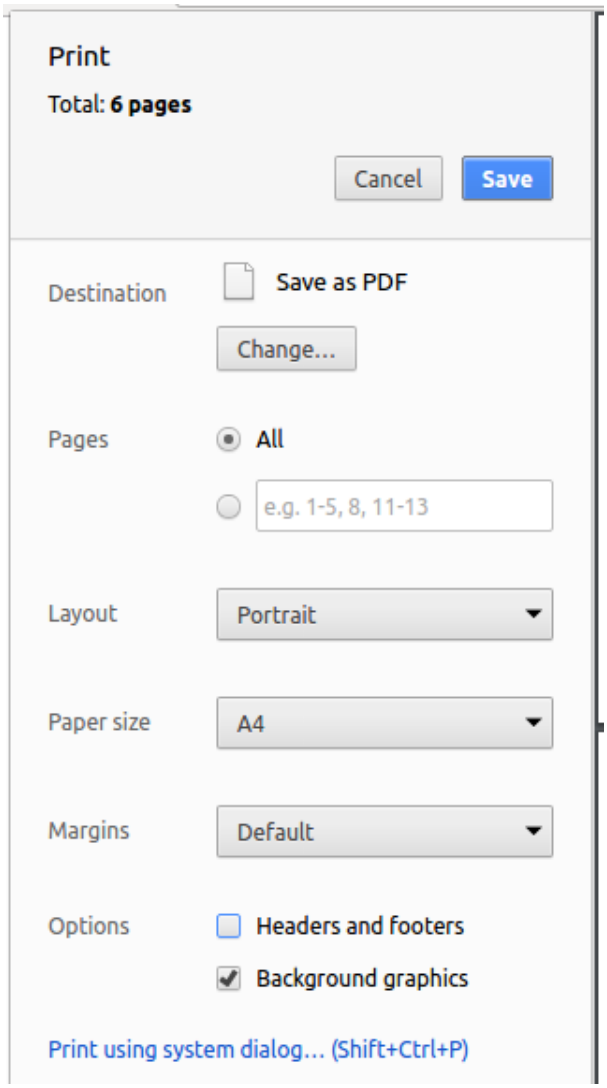



*Figure 5.6.1 : Saving documentation as PDF files in Chrome*

# 6. Testing

## 6.1. Running Tests

There are three ways to run tests.

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

**Method 1: Using IntelliJ JUnit test runner**

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

**Method 2: Using Gradle**

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

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

**Method 3: Using Gradle (headless)**

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

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

## 6.2. Types of Tests

We have two types of tests:

1. **GUI Tests** - These are tests involving the GUI. They include,
   a. *System Tests* that test the entire App by simulating user actions on the GUI. These are in the `systemtests` package.
   b. *Unit tests* that test the individual components. These are in `seedu.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`

## 6.3. 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`.

# 7. Dev Ops

## 7.1. Build Automation

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

## 7.2. Continuous Integration

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

## 7.3. Making a Release

Here are the steps to create a new release.

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

## 7.4. 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: 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 this section Improving a Component.
2. Next, add a feature that touches multiple components to learn how to implement an end-to-end feature across all components. The section 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

**TIP** Do take a look at the Design: Logic Component section 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.
        - See this PR for the full solution.

## `Model` Component

**TIP** Do take a look at the Design: Model Component section 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.

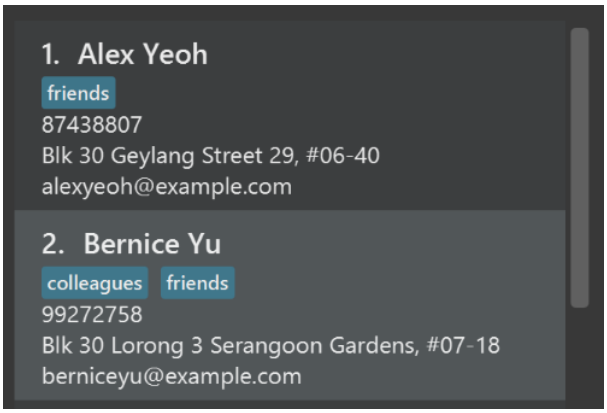
- Hints
  - The `Model` API needs to be updated.
  - Find out which of the existing API methods in `AddressBook` and `Person` classes can be used to implement the tag removal logic. `AddressBook` allows you to update a person, and `Person` allows you to update the tags.
- Solution
  - Add the implementation of `deleteTag(Tag)` method in `ModelManager`. Loop through each person, and remove the `tag` from each person.
  - See this PR for the full solution.

## `Ui` Component

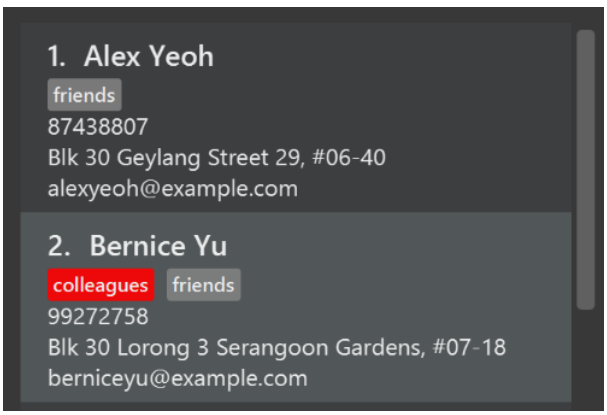
**TIP** Do take a look at the Design: UI Component section before attempting to modify the `UI` component.

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

   **Before**

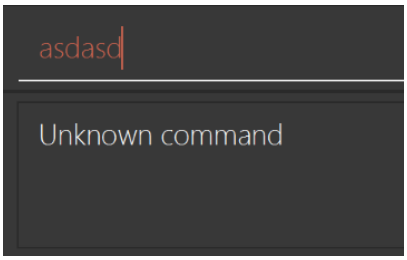
   **After**

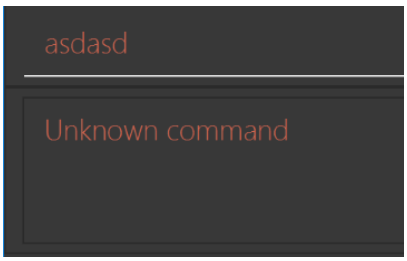
- Hints
  - The tag labels are created inside `PersonCard#initTags(ReadOnlyPerson)` (`new Label(tag.tagName)`). JavaFX's `Label` class allows you to modify the style of each Label, such as changing its color.
  - Use the .css attribute `-fx-background-color` to add a color.
- Solution
  - See this PR for the full solution.

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(event)` to react to this event appropriately.
     - See this PR for the full solution.

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.
  - See this PR for the full solution.

### `Storage` Component

**TIP** Do take a look at the Design: Storage Component section 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` 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.

### 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 `ReadOnlyPerson` 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 `ReadOnlyPerson` to Support a `Remark` Field

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

**Main:**

1. Add three methods `setRemark(Remark)`, `getRemark()` and `remarkProperty()`. Be sure to implement these newly created methods in `Person`, which implements the `ReadOnlyPerson` interface.
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`.
2. Be sure to modify the logic of the constructor and `toModelType()`, which handles the conversion to/from `ReadOnlyPerson`.

**Tests:**

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

#### [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#bindListeners()` to add the binding for `remark`.

**Tests:**

1. Modify `GuiTestAssert#assertCardDisplaysPerson(…)` so that it will compare the remark label.
2. In `PersonCardTest`, call `personWithTags.setRemark(ALICE.getRemark())` to test that changes in the `Person` 's remark correctly updates the corresponding `PersonCard`.

#### [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: User Stories

**User Profile:** John is an undergraduate student in a local university. He has an active social life with many friend circles. John wants a way to manage his contacts and a way to remind him tasks and responsibilities in related to these contacts.

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 | |
| * * * | 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 | check upcoming birthdays | keep track of all people who are having their birthday soon |
| * * * | user | edit contact information | make changes if needed |
| * * * | user | add birthday information | remember and keep track of my family members/friends birthday |
| * * * | user | favorite a person | keep track of the person I'm interested in recently. |
| * * * | user | find persons whose names contain a partial string | locate person whose name I forgot how to spell |
| * * * | user | find persons by tags | identify a group of people with common attributes |
| * * * | undergraduate student user | add events related to one or multiple contacts such as birthday or meetings | keep track tasks I need to do |

| Priority | As a … | I want to … | So that I can… |
|---|---|---|---|
| * * | user | add or remove some tags of a contact | quickly update contact information |
| * * | user | hide private contact details by default | minimize chance of someone else seeing them by accident |
| * * | user | add profile photo | can double check the identity if there are persons with the same name |
| * * | user | add multiple phone number to one person | store different contact number as one person may have more than one phone number |
| * * | user | have some important contacts | reach to my close friends easily at a click |
| * * | undergraduate student user | find persons by their address | identify which friends stay in the same campus |
| * * | undergraduate student user | have a reminder pop up when an event is coming up | remember to prepare and fulfill my responsibility |
| * | user with many persons in the address book | sort persons by name | locate a person easily |

# Appendix C: Use Cases

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

## Use Case: Delete Person

**MSS**

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

   Use case ends.

**Extensions**

   2a. The list is empty.

   Use case ends.

   3a. The given index is invalid.

      3a1. AddressBook shows an error message.

      Use case resumes at step 2.

# Use Case: Edit Person

**MSS**

1. User request to edit contact information
2. AddressBook shows old and new information stored

**Extensions**

   2a. No such person.

      2a1. AddressBook shows an error message.

      Use case ends.

   2b. Incorrect format of new information.

      2b1. AddressBook shows an error message and request for new information again.

      Use case resumes at step 1.

# Use Case: List All Persons

**MSS**

1. User request to have a list of all persons
2. AddressBook shows a list of all persons in the contact list

**Extensions**

2a. The list is empty.

Use case ends.

# Use Case: Find Persons by Tags

**MSS**

1. User request to list keywords
2. Application shows a list of persons who has those keywords as tags

**Extensions**

2a. The list is empty.

Use case ends.

# Use Case: Find Persons by Name String

**MSS**

1. User request to key in a name string
2. Application shows a list of persons whose names contain the full or partial string

Use case ends.

**Extensions**

2a. The list is empty.

Use case ends.

# Use Case: Find Persons by Phone Number

**MSS**

1. User request key in a phone number
2. Application shows a list of persons associated with the phone number

Use case ends.

**Extensions**

2a. The list is empty.

Use case ends.

## Use Case: Add an Event Reminder

**MSS**

1. User requests to add an event associated with a future date.
2. Application adds the event

   Use case ends.

**Extensions**

   3. User start the application on the date associated with the added event.

   4. Application shows the user popup reminder for the event upon the start.

   Use case ends.

# Appendix D: 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 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.

# Appendix E: Glossary

Mainstream OS

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

Private contact detail

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