# Planno - Developer Guide

By: Hu Wanqing, Jiang Chunhui, Hou Denghao, Leon Chow WenHao      Since: August 2017

Table of Contents

**Welcome to Planno**  **3**

**1. Setting up**  **3**

1.1. Prerequisites 3

1.2. Setting up the project in your computer 3

1.3. Verifying the setup 3

1.4. Configurations to do before writing code 4

**2. Design**  **5**

2.1. Architecture 5

2.2. Four major components 7

2.3. Common classes 11

**3. Implementation**  **12**

3.1. Undo/Redo mechanism 12

3.2. Find mechanism 21

3.3. Sort mechanism 23

3.4. AddEvent mechanism 24

3.5. ListEvent mechanism 25

3.6. DeleteEvent mechanism 26

3.7. ShowParticipants mechanism 27

3.8. ShowAllJoinedEvent mechanism 29

3.9. Person-Event mechanism 30

3.10. Logging 32

3.11. Configuration 32

**4. Documentation 32**

4.1. Editing documentation 33

4.2. Publishing documentation 33

4.3. Converting documentation to PDF format 33

**5. Testing 34**

5.1. Running tests 34

5.2. Types of tests 35

5.3. Troubleshooting testing 35

**6. Dev ops 36**

6.1. Build automation 36

6.2. Continuous integration 36

6.3. Making a release 36

6.4. Managing dependencies 36

**Appendix A: Suggested programming tasks to get started 36**

A.1. Improving each component 37

A.2. Creating a new command: remark 40

**Appendix B: User stories 43**

**Appendix C: Use cases 46**

**Appendix D: Non-functional requirements 54**

**Appendix E: Glossary 55**

# [Welcome to Planno](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#welcome-to-planno)

Planno is a Java application that helps users to conveniently manage contact as well as event details. To achieve this, Planno provides a large number of easy to use commands for users.

There are many ways to contribute to Planno: coding, testing, improving the build process and tools, or contributing to the documentation. This guide provides information that will not only help you get started as a Planno contributor, but that you’ll find useful to refer to even if you are already an experienced contributor.

This developer guide aims to give future programmers who hope to modify this application a general description of the architecture and design of the Desktop Application Planno. Planno provides users a platform to store and manage people’s contact information, and events related to people in Planno. This developer guide consists of the following sections:

# [1. Setting up](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#setting-up)

## [1.1. Prerequisites](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#prerequisites)

1. **JDK**1.8.0\_60 or later

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

1. **IntelliJ** IDE

|  |  |
| --- | --- |
| 信息 | 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. Setting up the project in your computer](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#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
   1. Click Configure > Project Defaults > Project Structure
   2. 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. Verifying the setup](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#verifying-the-setup)

1. Run the seedu.address.MainApp and try a few commands
2. [Run the tests](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#testing) to ensure they all pass.

## [1.4. Configurations to do before writing code](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#configurations-to-do-before-writing-code)

### [1.4.1. Configuring the coding style](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#configuring-the-coding-style)

This project follows [oss-generic coding standards](https://github.com/oss-generic/process/blob/master/docs/CodingStandards.md). 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](file:///C:\\Users\\Celia\\Documents\\planno\\build\\docs\\html5\\UsingCheckstyle.html) document to configure Intellij to check style-compliance as you write code.

### [1.4.2. Updating documentation to match your fork](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#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](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#setting-up-ci)

Set up Travis to perform Continuous Integration (CI) for your fork. See [UsingTravis.adoc](file:///C:\\Users\\Celia\\Documents\\planno\\build\\docs\\html5\\UsingTravis.html) to learn how to set it up.

Optionally, you can set up AppVeyor as a second CI (see [UsingAppVeyor.adoc](file:///C:\\Users\\Celia\\Documents\\planno\\build\\docs\\html5\\UsingAppVeyor.html)).

|  |  |
| --- | --- |
|  | 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) |

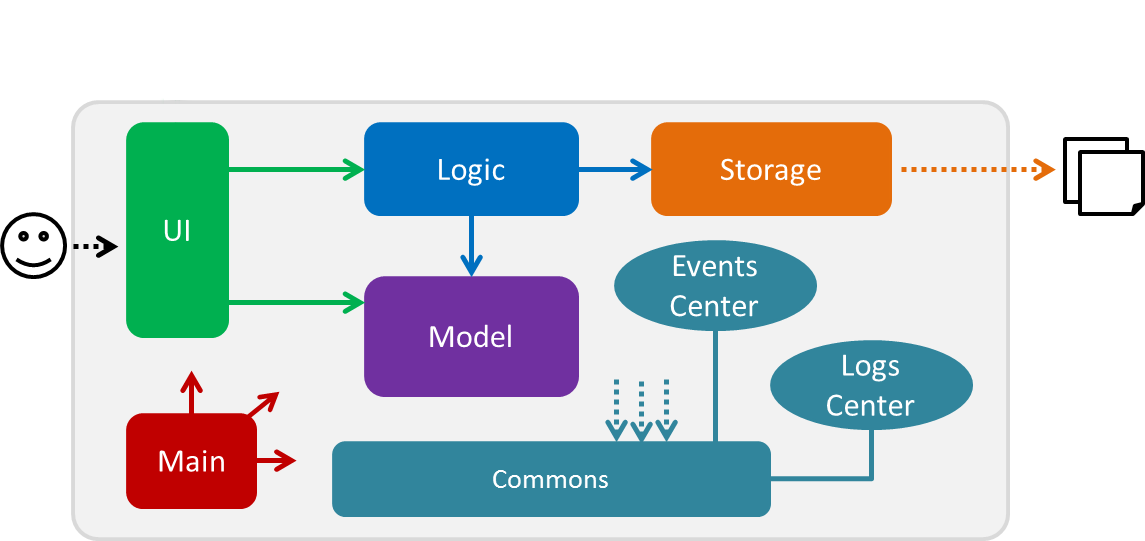
### [1.4.4. Getting started with coding](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#getting-started-with-coding)

When you are ready to start coding,

1. Get some sense of the overall design by reading the [Architecture](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#architecture) section.
2. Take a look at the section [Suggested Programming Tasks to Get Started](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#suggested-programming-tasks-to-get-started).

# [2. Design](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#design)

## [2.1. Architecture](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#architecture)



*Figure 2.1.1: Architecture Diagram*

The ***Architecture Diagram*** *(Figure 2.1.1)* given above explains the high-level design of the App. Given below is a quick overview of each component:

|  |  |
| --- | --- |
| 信息 | The .pptx files used to create diagrams in this document can be found in the [diagrams](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/docs/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](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/MainApp.java). It is responsible for:

* (At app launch) Initializing the components in the correct sequence, and connecting hem up with each other.
* (At app shut down) Shutting down the components and invoking cleanup method where necessary.

[Commons](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#common-classes) represents a collection of classes used by other components. Among them, the following two classes play important roles at the architecture level:

* EventsCenter : This class is written using [Google’s Event Bus library](https://github.com/google/guava/wiki/EventBusExplained). Components communicate with each other by posting event in this class (i.e. a form of event-driven nature of design).
* LogsCenter : Used by many classes to write log messages to the App’s log file.
* Classes used by multiple components are in the seedu.addressbook.commons package.

The rest of the App consists of four major components:

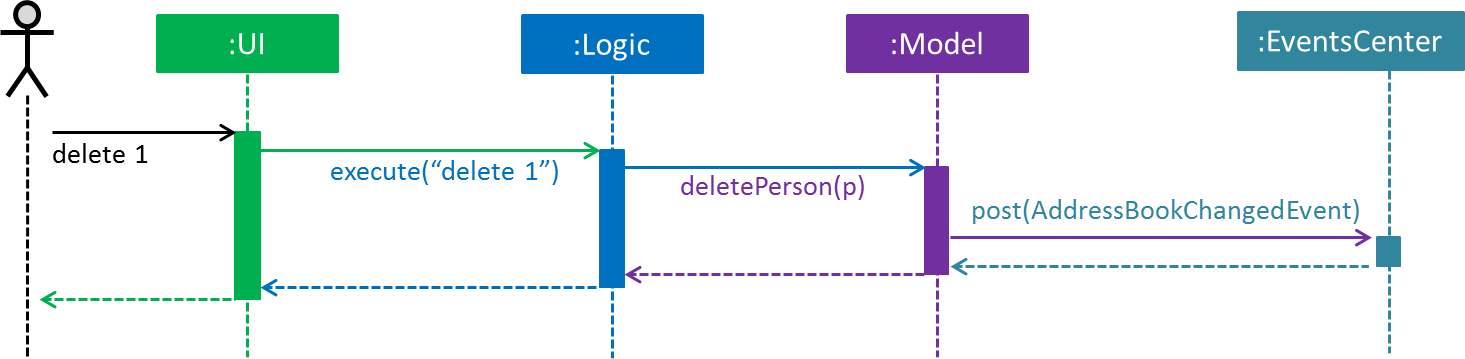
* [UI](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#ui-component) : Displays the user interface.
* [Logic](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#logic-component) : Executes commands.
* [Model](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#model-component) : Holds the data of the App in-memory.
* [Storage](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#storage-component) : Reads data from, and writes data to, the hard disk.

Each of the four components above

* 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 defines its API in the Logic.java interface and exposes its functionality using the LogicManager.java class.

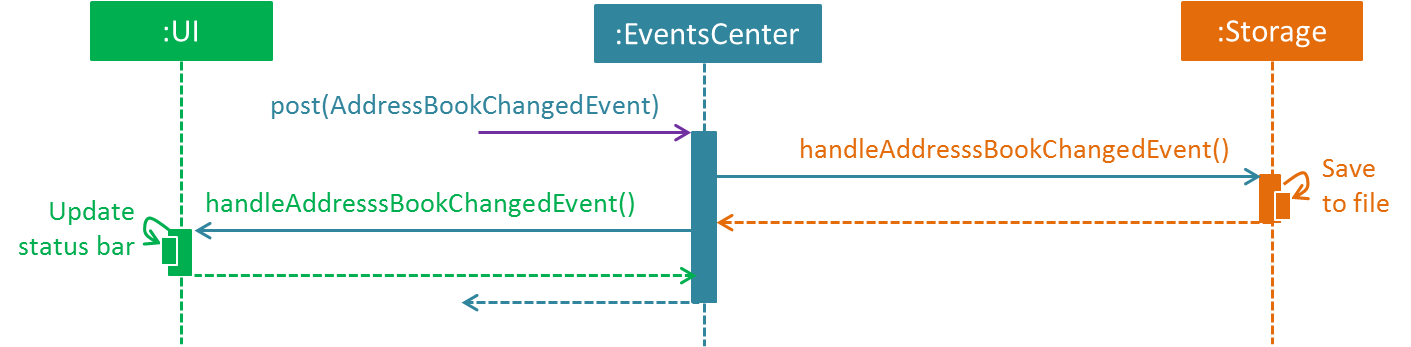
## Events-Driven nature of the design

The ***Sequence Diagram*** *(Figure 2.1.2)* below shows how the components interact using EventsCenter for the scenario where the user issues the command delete 1.

*Figure 2.1.2: Component interactions for delete 1 command (part 1)*

|  |  |
| --- | --- |
| 信息 | 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 *(Figure 2.1.3)* 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 2.1.3: Component interactions for delete 1 command (part 2)*

|  |  |
| --- | --- |
| 信息 | Note how the event is propagated through the EventsCenter to the Storage and UI without Modelhaving 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. Four major components](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#four-major-components)

### [2.2.1. UI component](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#ui-component)UiClassDiagram

*Figure 2.2.1: Structure of the UI Component*

**API** : [Ui.java](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/ui/Ui.java)

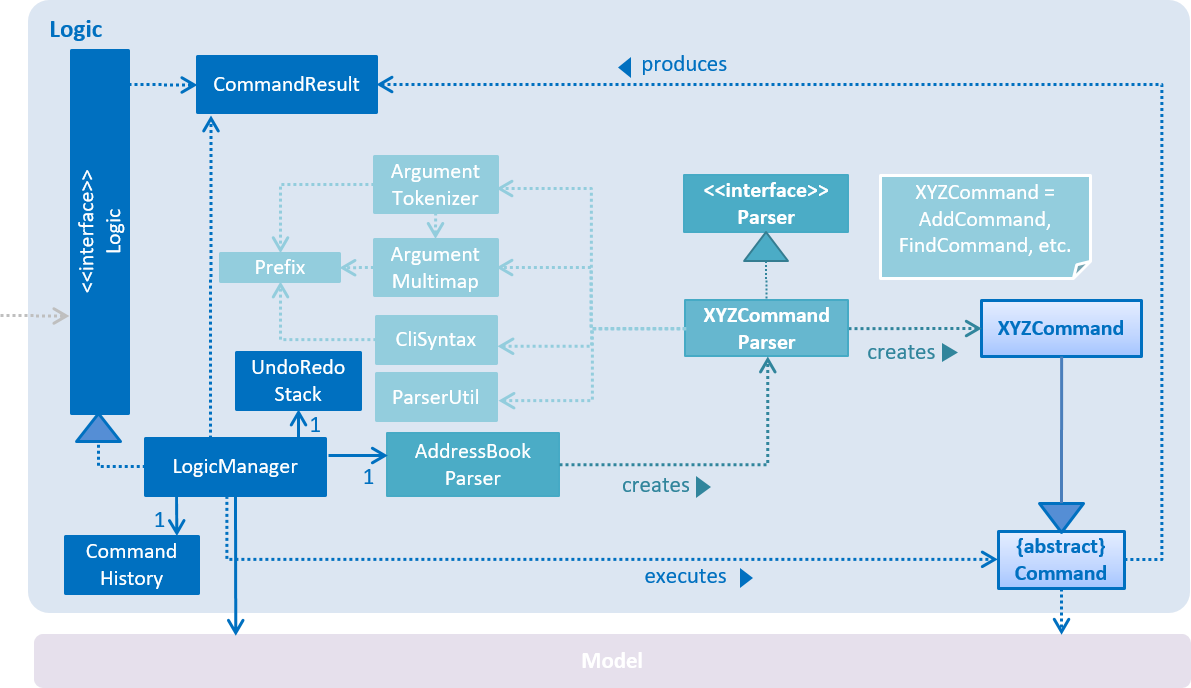
The structure of the UI Component is shown in the ***Class Diagram*** *(Figure 2.2.1)* above. The UI consists of a MainWindow that is made up of parts like CommandBox, ResultDisplay, PersonListPanel, StatusBarFooter, BrowserPaneletc. All these, including the MainWindow, inherit from the abstract UiPart class.

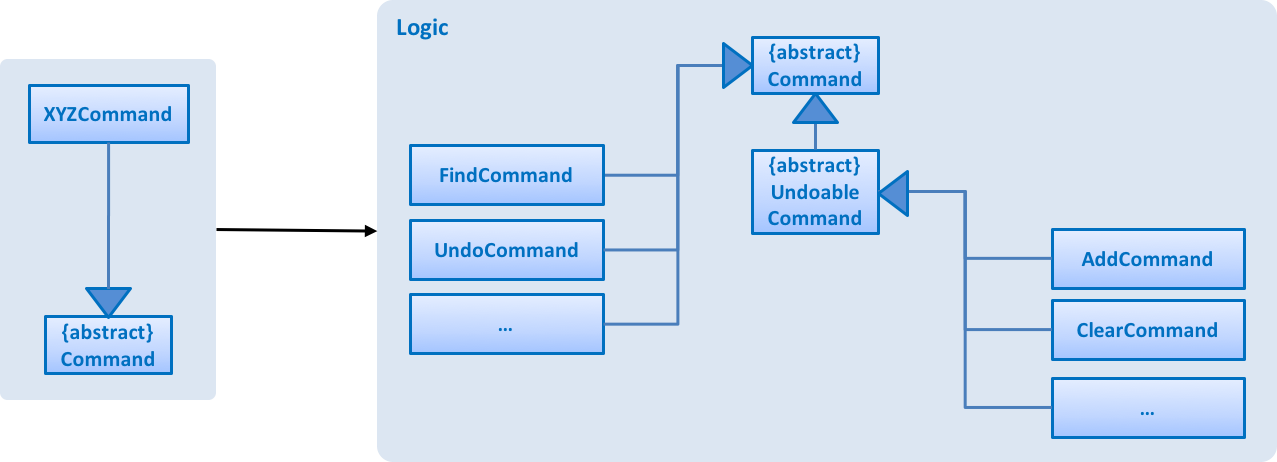
The UI component uses JavaFx UI framework. The layout of these UI parts are defined in matching .fxml files that are in the src/main/resources/view folder. For example, the layout of the [MainWindow](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/ui/MainWindow.java) is specified in [MainWindow.fxml](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/resources/view/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 changes.
* Responds to events raised from various parts of the App and updates the UI accordingly.

### [2.2.2. Logic component](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#logic-component)

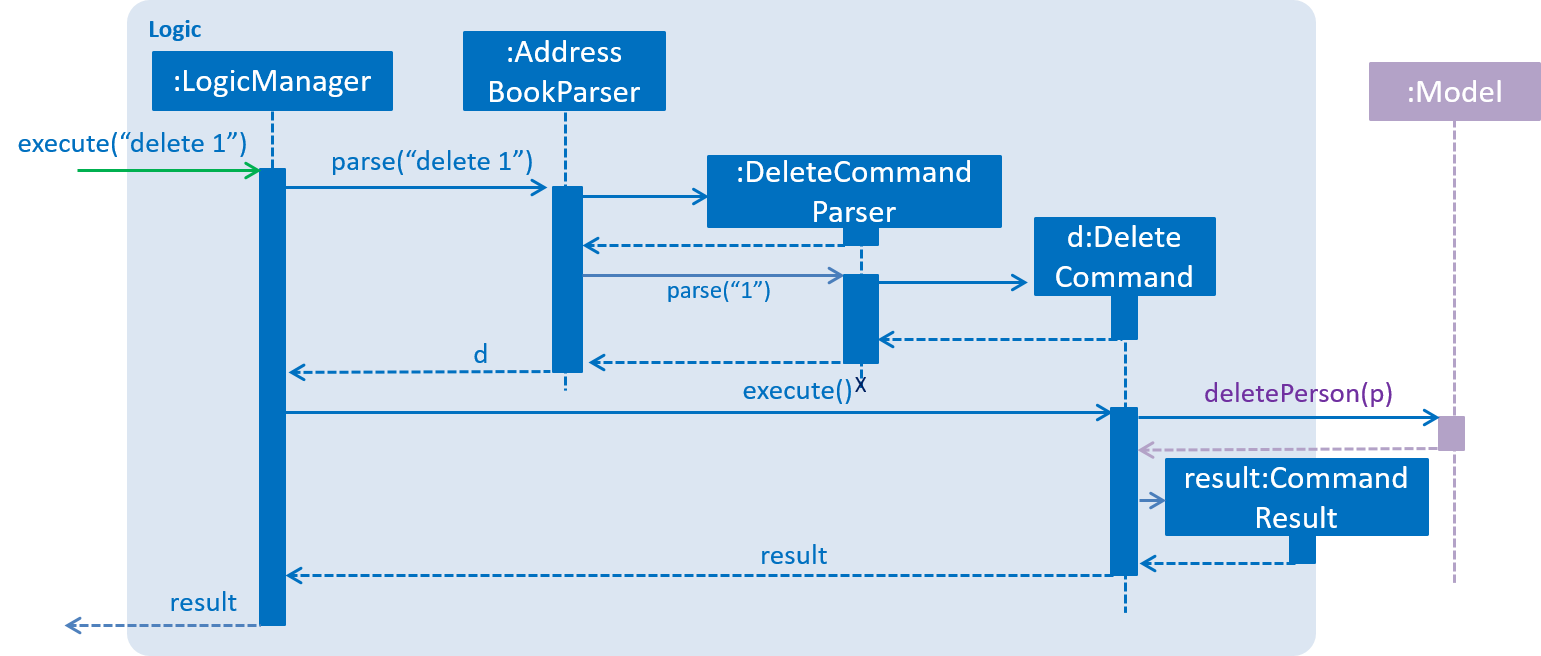
The diagrams *(Figure 2.2.2.1 to Figure 2.2.2.3)* given below show the structure of whole logic component, and structure of commands in details.

*Figure 2.2.2.1: Structure of the Logic Component*

*Figure 2.2.2.2: Structure of Commands in the Logic Component. This diagram shows finer details concerning XYZCommandand Command in Figure 2.3.1*

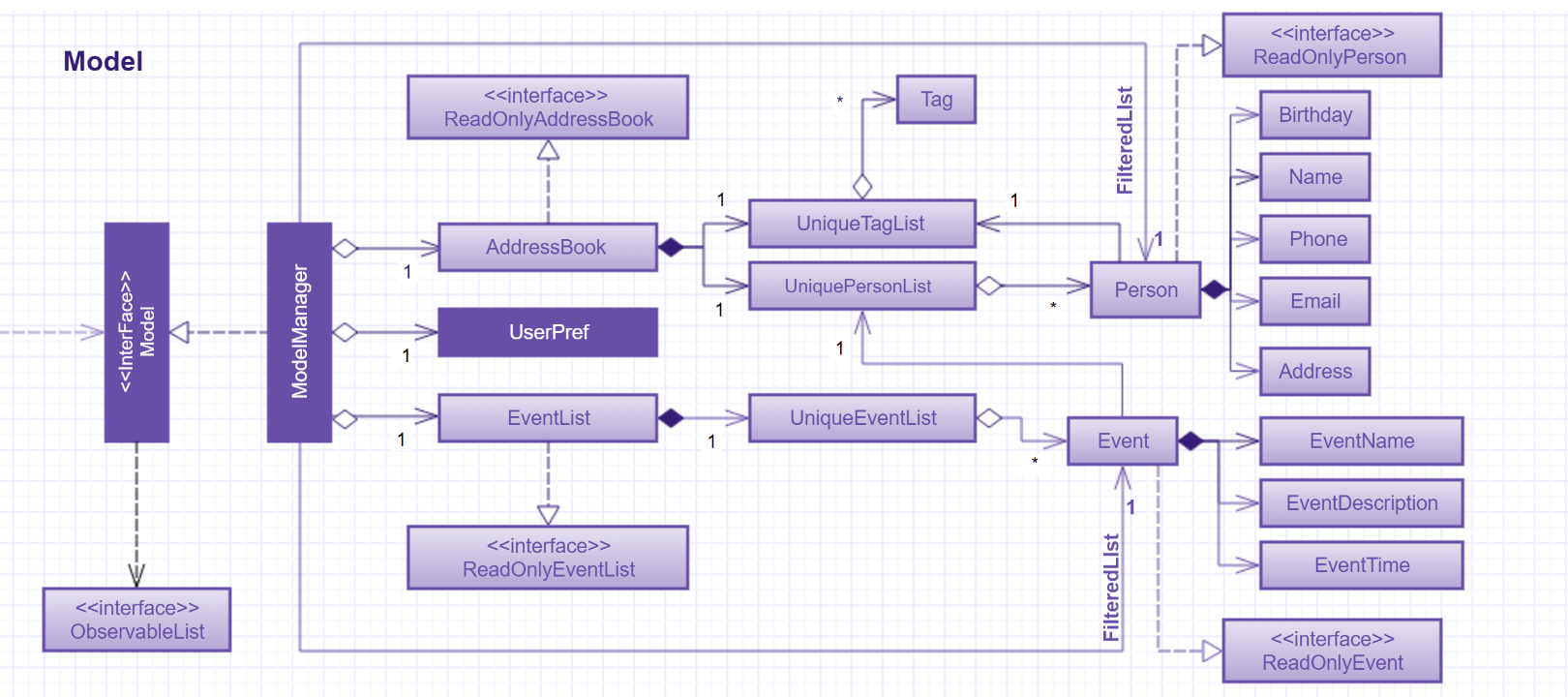
**API** : [Logic.java](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/logic/Logic.java)

The logic will perform the following steps:

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.
5. Given below is the ***Sequence Diagram*** *(Figure 2.2.2.3)* for interactions within the Logic component for the execute("delete 1") API call.

*Figure 2.2.2.3: Interactions Inside the Logic Component for the delete 1 Command*

### [2.2.3. Model component](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#model-component)

The following diagram *(Figure 2.2.3)* shows the class structure of the Model component.

*Figure 2.2.3: Structure of the Model Component*

**API** : [Model.java](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/model/Model.java)

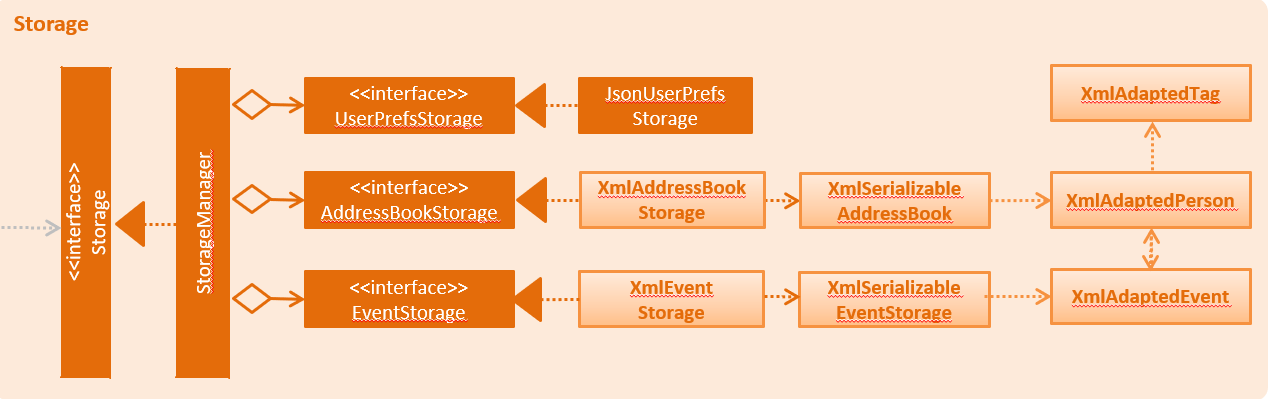
Generally, the Model is managed by a Model manager, which

* stores a UserPref object that represents the user’s preferences.
* maintains an AddressBook and a EventList.
* stores 2 unmodifiable list: ObservableList<ReadOnlyPerson> and ObservableList<ReadOnlyEvent>. They are bounded to UI so that the UI can automatically update when the data in the list change.
* does not depend on any of the other three components.

In detail, the AddressBook and the EventList are respectively responsible for person and event information.

* The AddressBook
  + stores people’s information as a person list with no duplicate persons. The information includes one’s personal information and contact details.
  + keeps track of all the tags that had been added to some people in the person list
  + for each person in the list, the person holds a modifiable tag list that contains all the tag this person has.
* The EventList
  + stores event’s information as an event list.
  + for each event in the list, the event maintains a modifiable list to keep track of who the participants of the events are.

### [2.2.4. Storage component](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#storage-component)



*Figure 2.2.4: Structure of the Storage Component*

**API** : [Storage.java](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/storage/Storage.java)

The diagram *(Figure 2.2.4)* above shows the structure of the Storage component. The StorageManager handles the saving and loading of data for both AddressBookStorage and EventStorage. XmlSerializableAddressBook and XmlSerializableEventStorage handle the conversion from Java to Xml format using XmlAdaptedPerson, XmlAdaptedEventand XmlAdaptedTag.

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.
* can save event storage data in xml format and read it back.

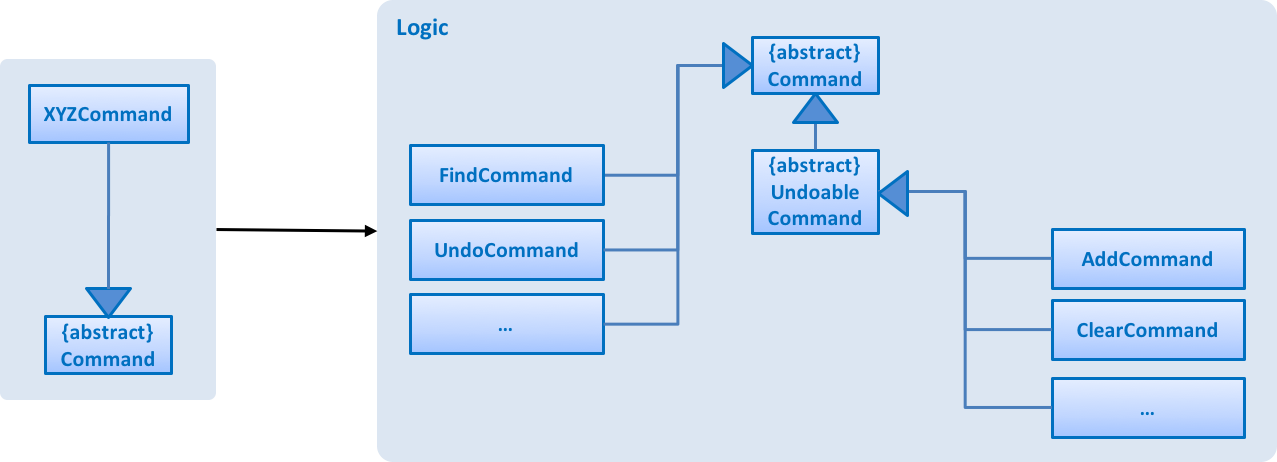
# [3. Implementation](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#implementation)

This section describes some noteworthy details on how certain features are implemented. For the features described in this section, their design considerations are included where applicable.

## [3.1. Undo/Redo mechanism](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#undo-redo-mechanism)

### [3.1.1. General implementation](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#general-implementation)

The undo/redo mechanism is facilitated by an UndoRedoStack, which resides inside LogicManager. It supports undoing and redoing of commands that modify the state of the 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 *(Figure 3.1.1.1)* shows the inheritance diagram for commands:

*Figure 3.1.1.1: Structure of commands*

As you can see from the diagram *(Figure 3.1.1.1)*, UndoableCommand adds an extra layer between the abstract Commandclass 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](https://www.tutorialspoint.com/design_pattern/template_pattern.htm).

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();

}

**protected** **abstract** **void** undo();

}

**public** **class** **DeleteCommand** **extends** UndoableCommand {

@Override

**public** CommandResult executeUndoableCommand() {

*// ... delete logic ...*

}

@Override

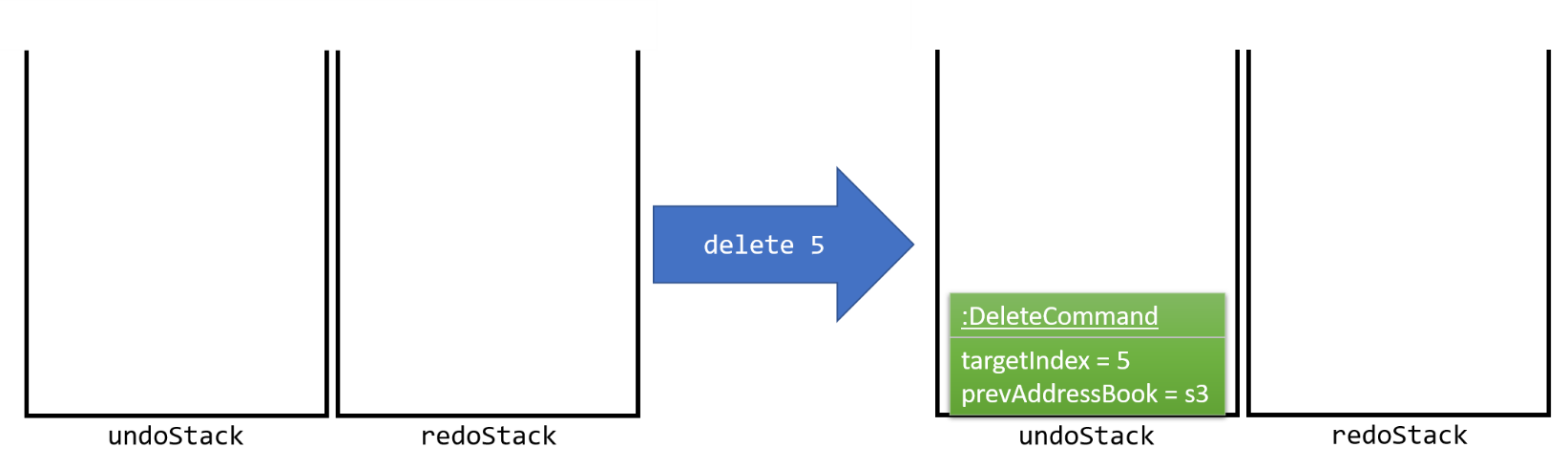
**protected** **void** undo() {

*// ... undo delete logic ...*

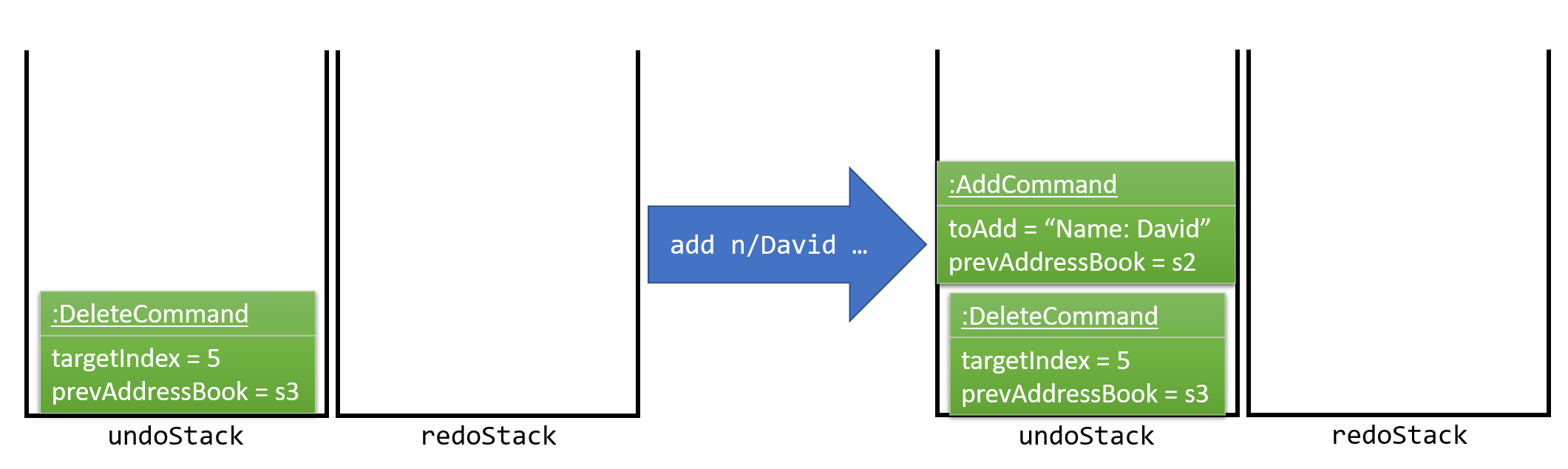
}

}

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). This is shown in the image *(Figure 3.1.1.2)*below.

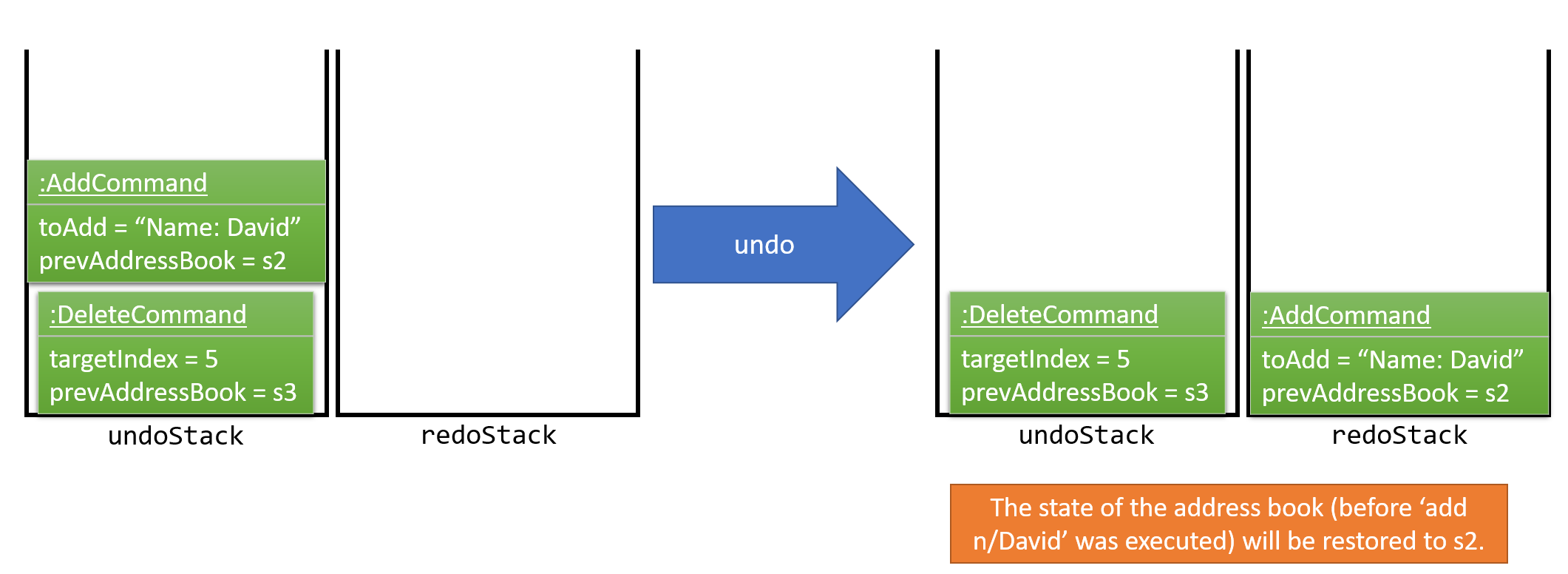
*Figure 3.1.1.2: undo example part 1*

As the user continues to use the program, more commands are added into the undoStack. For example, the user may execute add n/David …​ to add a new person. This is show in the image *(Figure 3.1.1.3)* below.

*Figure 3.1.1.3: undo example part 2*

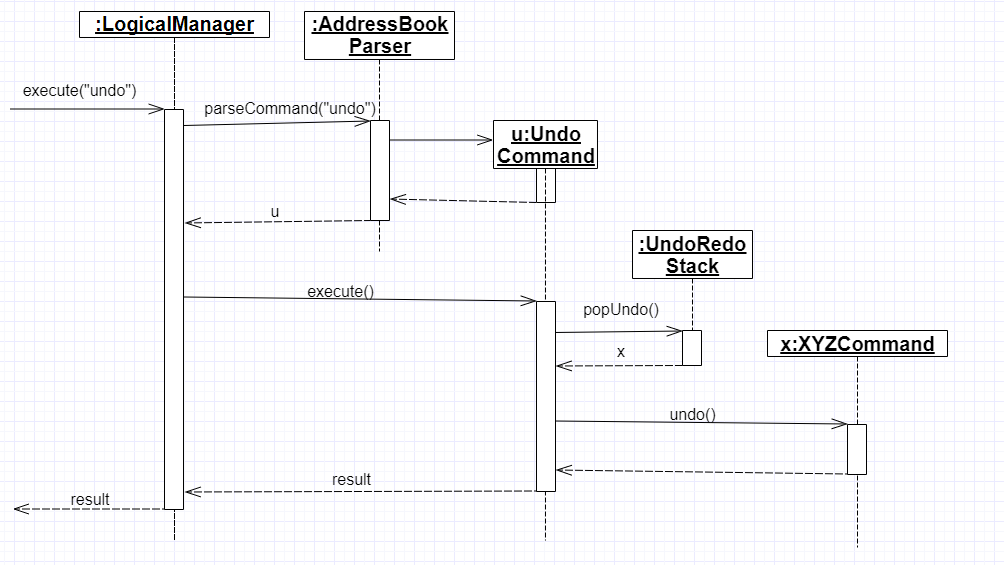
|  |  |
| --- | --- |
| 信息 | If a command fails its execution, it will not be pushed to the UndoRedoStack at all. |

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. This is shown in the image *(Figure 3.1.1.4)* below.

*Figure 3.1.1.4: undo example part 3*

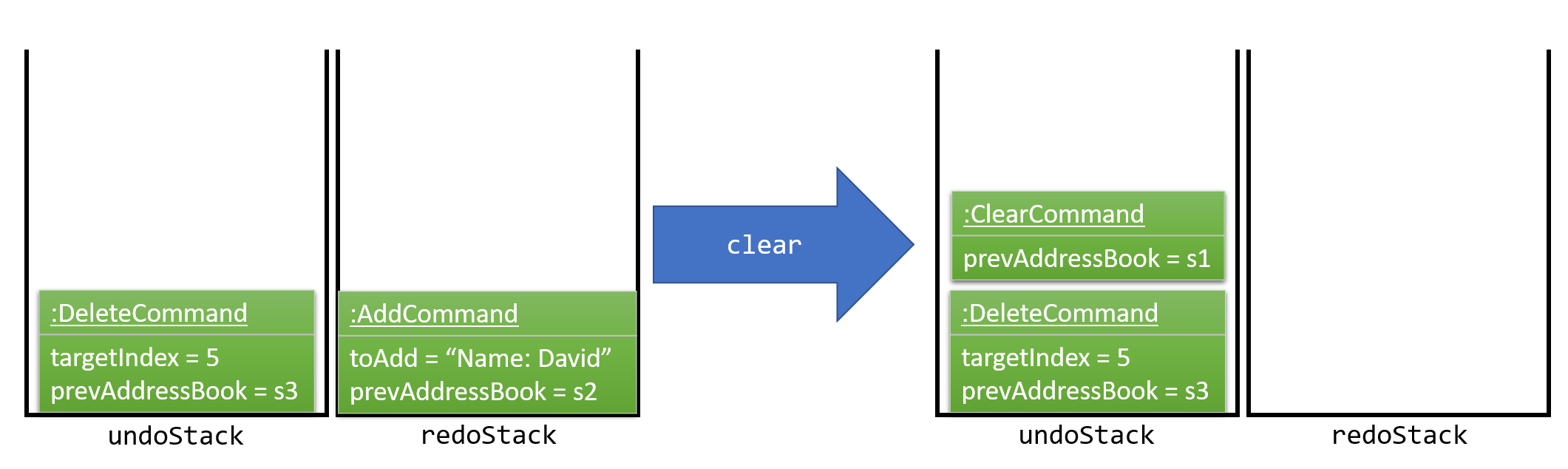
|  |  |
| --- | --- |
| 信息 | If the undoStack is empty, then there are no other commands left to be undone, and an Exception will be thrown when popping the undoStack. |

The following ***sequence diagram*** *(Figure 3.1.1.5)* shows how the undo operation works:

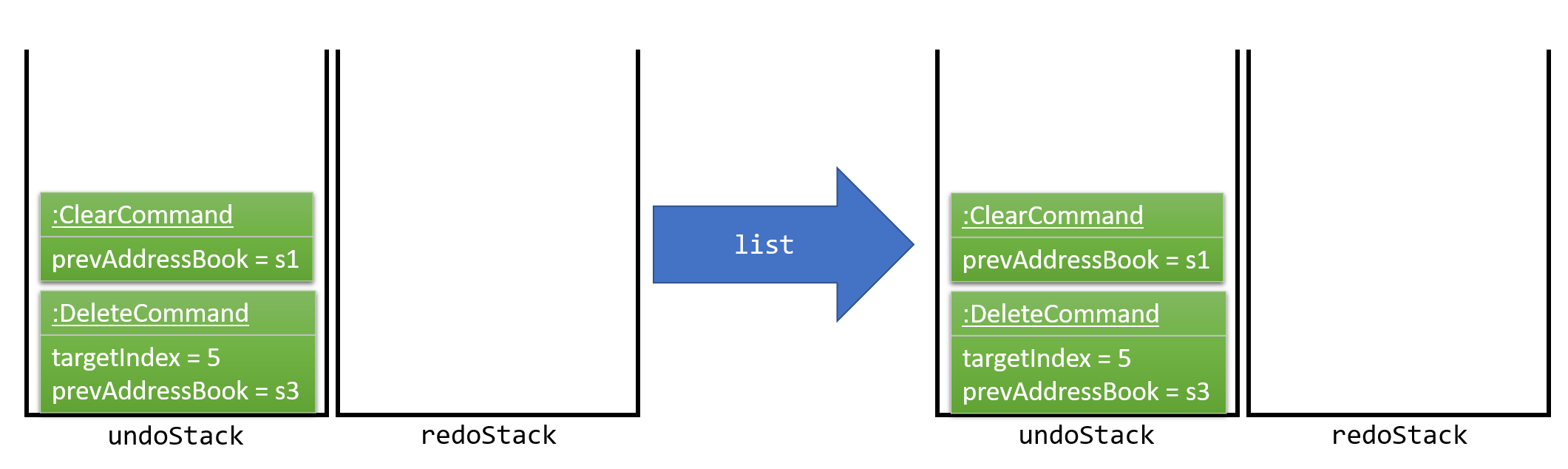
*Figure 3.1.1.5: undo sequential diagram*

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).

|  |  |
| --- | --- |
| 信息 | 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. |

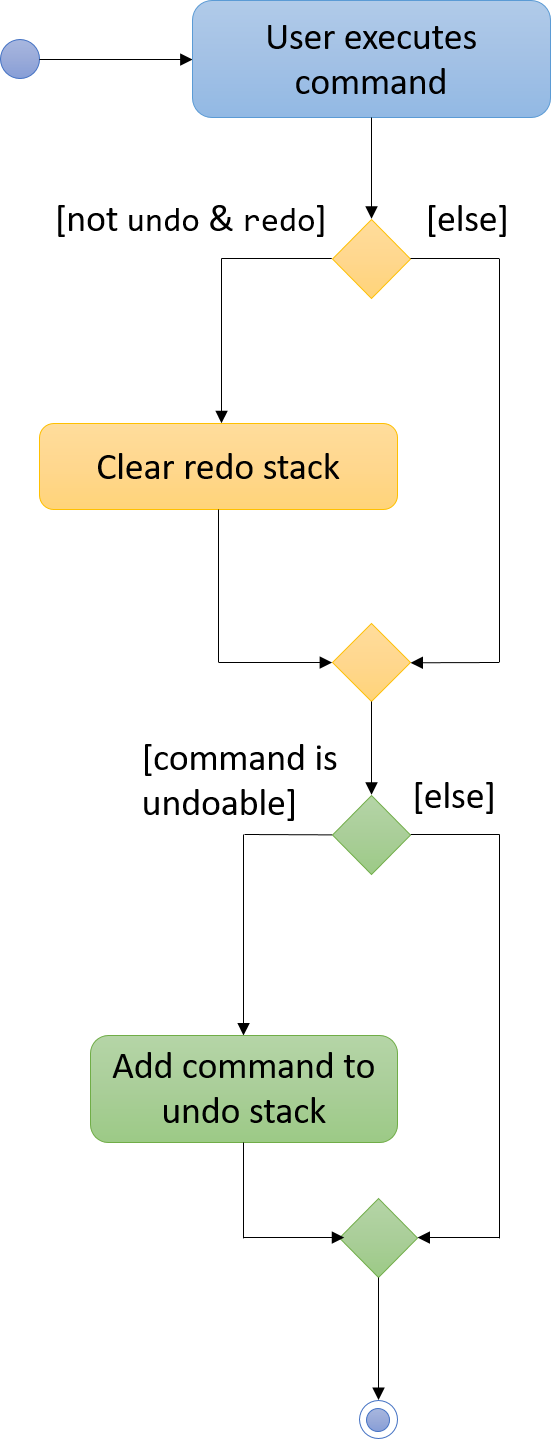
The user now decides to execute a new command, clear. As before, clear will be pushed into the undoStack. This time the redoStack is no longer empty. It will be purged as it no longer make sense to redo the add n/David command (this is the behavior that most modern desktop applications follow). This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

*Figure 3.1.1.6: redo example part 1*

Commands that are not undoable are not added into the undoStack. For example, list, which inherits from Commandrather than UndoableCommand, will not be added after execution. This is shown in the image *(Figure 3.1.1.7)* below.

*Figure 3.1.1.7: redo example part 2*

The following ***activity diagram*** *(Figure 3.1.1.8)* summarize what happens inside the UndoRedoStack when a user executes a new command.



*Figure 3.1.1.8: undo/redo activity*

### **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:** It might be hard for new developers to understand the template pattern.  
**Alternative 2:** Just override execute()  
**Pros:** This implementation does not involve the template pattern, and is 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):** Individual command knows how to undo/redo by itself  
**Pros:** The app 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 is correct.  
**Alternative 2:** Save the entire address book.  
**Pros:** This is easy to implement.  
**Cons:** This could have performance issues in terms of memory usage.

**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:** Users 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:** This could be more intuitive for the users.  
**Cons:** Users 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.

**Aspect:** Data structure to support the undo/redo commands  
**Alternative 1 (current choice):** Use a separate stack for undo and redo  
**Pros:** New incoming developers of our project, such as new Computer Science undergraduates are easy to understand.  
**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:** We should deal with commands that have already been undone: We must remember to skip these commands. This violates Single Responsibility Principle and Separation of Concerns, as HistoryManager now needs to do two different things.

**The following sections detail the specific implementation of undo and redo for each type of**UndoableCommand**:**

### [3.1.2. Undo an adding command](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#undo-an-adding-command)

* We keep the person added by that command.
* To undo it, we delete it from the address book.
* To redo it, we re-add it to the address book.

|  |  |
| --- | --- |
| 信息 | The person is associated with a list of tags, and we need to eliminate them when we undo. |

*The process is as follows:*

1. When we add a person to address book:
   1. Extract all tags which are associated with this person but not in the tag list before executing add command.
   2. Keep the tags extracted in step 1 in another list named newTags.
   3. Continue executing add command.
2. When we undo this add command:
   1. Delete the person from address book.
   2. Remove all tags which are in newTags from the tag list in the address book.
3. When we redo this command:
   1. Add this person into address book (Since newTags already exists).

### **Design Considerations**

**Aspect:** How to deal with tags when we undo add command  
**Alternative 1 (current choice):** Remove all tags which only exists in this person  
**Pros:** This allows the add command to be undone completely.  
**Cons:** Need to ensure each tag that is supposed to be removed does not exist in any other person.  
**Alternative 2:** Do not remove any tag  
**Pros:** Easy to implement.  
**Cons:** Tag list is not reverted to the state before add command executes.

### [3.1.3. Undo a delete command](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#undo-an-delete-command)

* We keep the person to be deleted, and his position in the address book.
* To undo it, we add this person to the original position in the address book.
* To redo it, we delete this person from the address book.

*The process is as follows:*

1. When we delete a person from the address book:
   1. Get the target person by index.
   2. Store both target person and the index.
   3. Continue executing delete command.
2. When we undo this delete command:
   1. Add target person to the position index.
3. When we redo this command:
   1. Delete the target person.

### **Design Considerations**

**Aspect:** Where to add the target person when we undo the delete person  
**Alternative 1 (current choice):** Add it to the original position index  
**Pros:** This allows the delete command to be undone completely, and the sequence of persons in address book will not change because of the undo.  
**Cons:** Need to spend more time adding a person back to its original position in the middle of the list, because it requires all persons behind index to switch to right.  
**Alternative 2:** Just add it at the back of address book  
**Pros:** Easy to implement, and more efficient.  
**Cons:** The sequence of persons in address book will change.

### [3.1.4. Undo an edit command](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#undo-an-edit-command)

* We keep the personToEdit and the editedPerson.
* To undo it, we swap the role of personToEdit and editedPerson, and edit it.
* To redo it, we swap back the role and re-edit it.

|  |  |
| --- | --- |
| 信息 | Similar as add command, if the editedPerson causes some change in tagList, we need to remove such tags accordingly. |

*The process is as follows:*

1. When we edit a person to address book:
   1. Extract all tags which are associated with editedPerson but not in the tag list before executing edit command.
   2. Keep the tags extracted in step 1 in another list named newTags.
   3. Continue executing add command.
2. When we undo this edit command:
   1. Swap the role of personToEdit and editedPerson.
   2. Execute edit operation.
   3. Remove all tags which are in newTags from the tag list in the address book.
3. When we redo this command:
   1. Swap back the role of personToEdit and editedPerson
   2. Execute edit operation (Since newTags already exists).

### [3.1.5. Undo a clear command](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#undo-an-clear-command)

* We keep the whole copy of address book for undo clear command.
* To undo it, we reset the data of the address book.
* To redo it, we clear the address book again.

*The process is as follows:*

1. When we clear the address book:
   1. Keep a copy of the address book.
   2. Assign an empty address book to clear the address book.
2. When we undo this clear command:
   1. Reset the data as the copy of address book.
3. When we redo this command:
   1. Assign an empty address book to the model.

## [3.2. Find mechanism](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#find-mechanism)

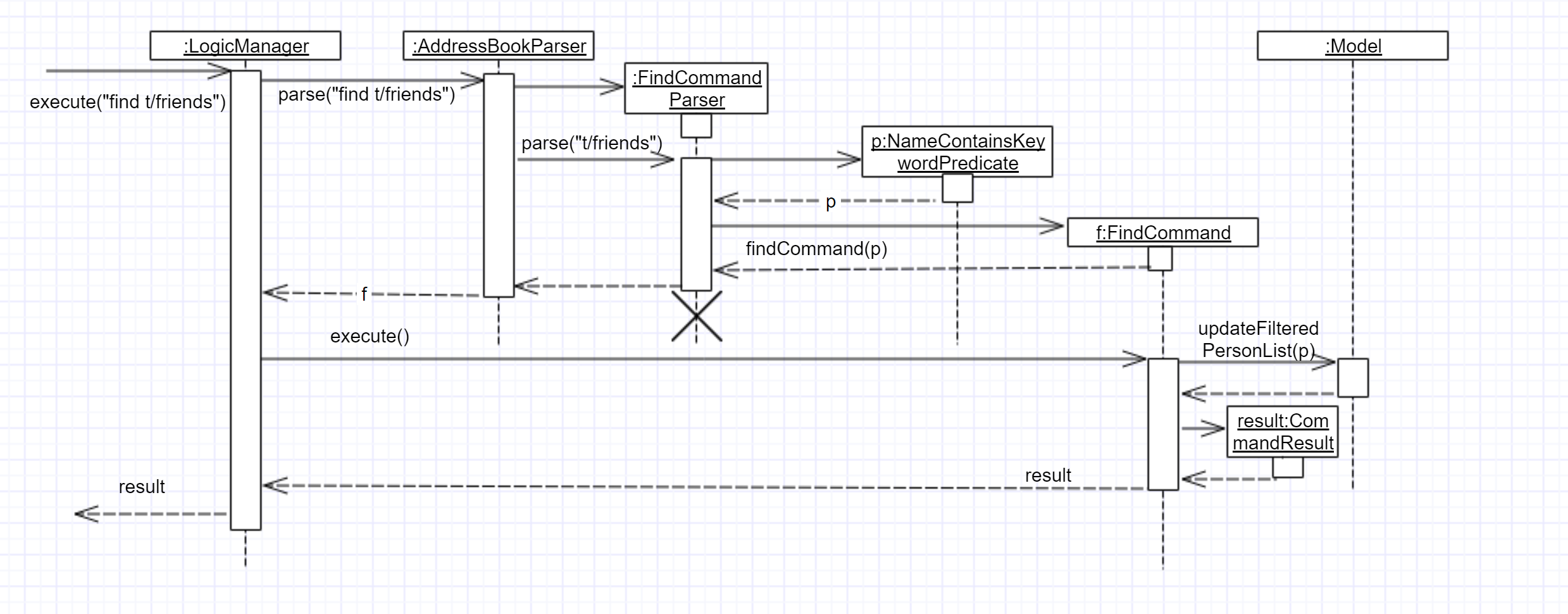
The Find mechanism is facilitated by NameContainsKeyWordPredicate class, which resides in model.person package. This command supports users of finding a list of persons by tags and names. Any person that has either of the entered keyword will be filtered into the list. Such command will inherit from Command.

The find command accept two types of keyword:

* tag: it is identified by a prefix t/.  
  e.g. find t/friends means to find any person that have a tag called friends
* name: anything that does not begin with tag is identified as name keyword.  
  e.g. find friends means to find any person whose name contains the keyword friends

|  |  |
| --- | --- |
| 信息 | if a tag name is not preceded with a prefix t/, then the tag name will be identified as a person name. As a result, the people with their name containing the tag name will be displayed. |

Below is a ***sequence diagram*** *(Figure 3.2)* for executing a find command: find t/friends. It will find persons that have the tag friends.



*Figure 3.2: sequence diagram for find command*

The command will be sent to LogicManager where it will call AddressBookParser to parse the command. Subsequently, FindCommandParser will parse argument t/friends and create a new findCommand with predicate t/friends. The findCommand is then executed in LogicManager, during which the findCommand will update the filtered person list with predicate tag called friends. This update will cause the person list displayed in GUI to be updated, so that the persons with tag friends will be displayed.

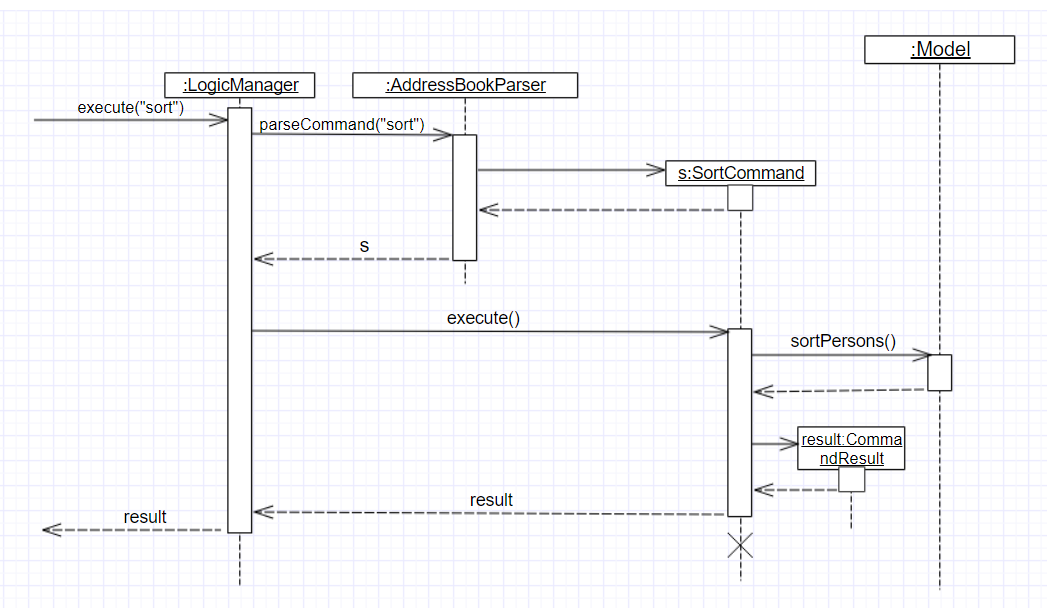
### **Design considerations**

**Aspect:** Implementation of find command  
**Alternative 1 (current choice):** Select people that have any of the keywords  
**Pros:** For every person, we only need to ensure that at least one keyword exists, so we don’t need to go through every information of a person.   
**Cons:** It is difficult to know by which keyword a person is selected.  
**Alternative 2:** Select people that has any of the keywords, and highlight selected keywords in each person’s person card  
**Pros:** Shows clearly what keywords each selected person contains.  
**Cons:** We need to go through everything of a person, in order to highlight EVERY keyword the person has.

**Aspect:** Type of keywords that can be found  
**Alternative 1 (current choice):** Only person name and tag can be used in the find command  
**Pros:** We only need to check a person’s name and tag list to find any matches.  
**Cons:** User will not be able to find a person by other information such as phone or email.  
**Alternative 2:** Include every information of a person (i.e. phone, email, address)  
**Pros:** A person can be found in many ways.  
**Cons:** Implementation is complicated.

## [3.3. Sort mechanism](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#sort-mechanism)

The sort command is facilitated by the LogicManager class. It supports sorting contact list and makes it easier for users to find contacts they want according to alphabetical order.

The following ***sequence diagram*** *(Figure 3.3)* shows how the sort operation works:

*Figure 3.3: sequence diagram for sort command*

The user enters a sort command which is received by LogicManager. LogicManager calls AddressBookParser to parse the command. Then AddressBookParser creates the sort command object and returns it. LogicManager receives the command object and executes the command. The sort command object calls sortPersons() in the Model. Then Model returns the sorted list. The sort command object receives the sorted list and creates CommandResult object and returns it.LogicManager receives the result and displays it through UI.

### **Design considerations**

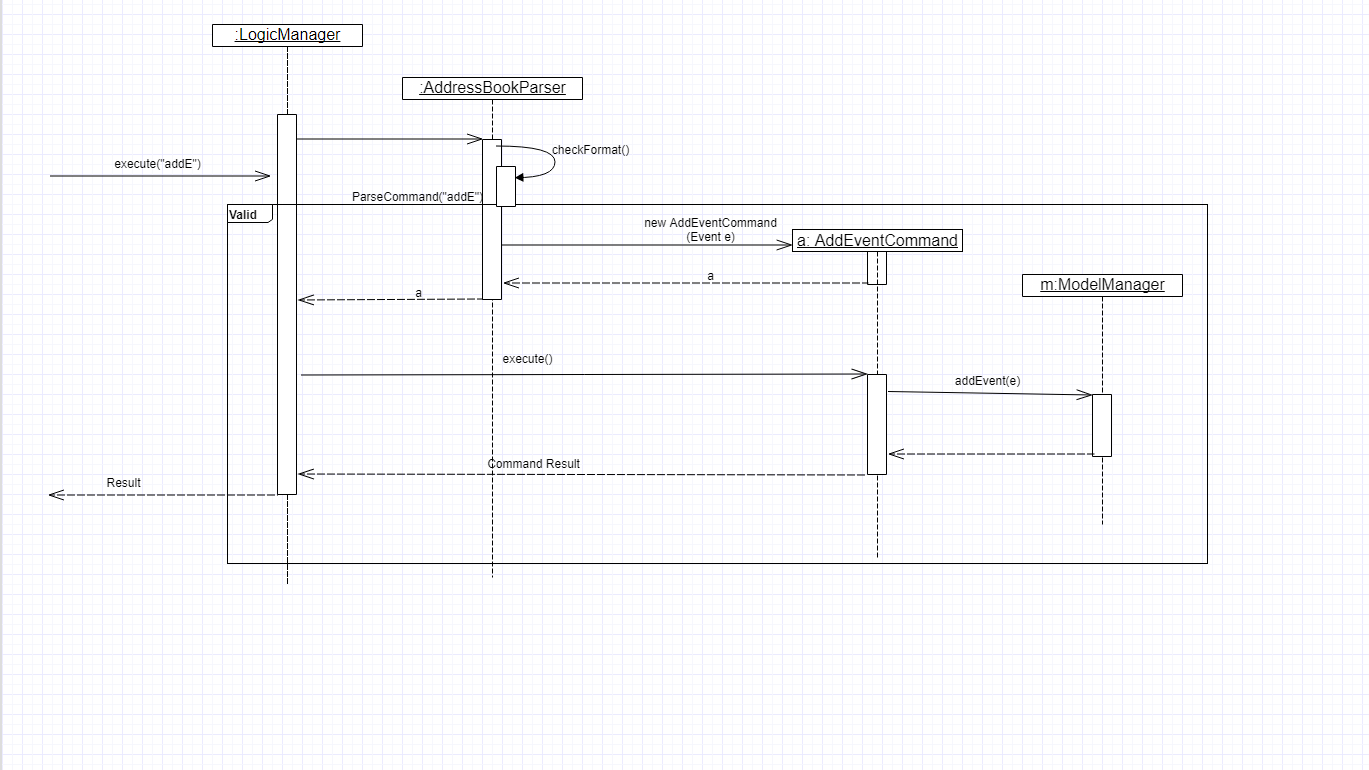
**Aspect:** How sort command affects the data  
**Alternative 1 (current choice):** Sort the copy of contact list and return it.  
**Pros:** It is more defensive and keeps the data unchanged.  
**Cons:** Users have to sort the list every time when they open the application.  
**Alternative 2:** Sort the actual contact list and return it.  
**Pros:** Users do not need to type the command everytime.  
**Cons:** Sort command is not undoable which means that the order of list cannot change anymore after you call it. It is also less defensive because you can frequently change the original data.

**Aspect:** How sort command executes  
**Alternative 1 (current choice):** Sorts the contact list according to alphabetical order (ascending order).  
**Pros:** Easy to implement.  
**Cons:** Less function for sort command.  
**Alternative 2:** Sorts the contact list according to different order (ascending order or descending order).  
**Pros:** Users will have more choices.  
**Cons:** It will be more complex for developers to maintain and test.

**Aspect:** What can be sorted  
**Alternative 1 (current choice):** According to names  
**Pros:** It is useful and necessary for general users.  
**Cons:** It is too narrow and common.  
**Alternative 2:** According to address/email/phone number  
**Pros:** It provides more choices for users.  
**Cons:** Because address/email/phone number is unique, you can usually find them without sorting the list.

## [3.4. AddEvent mechanism](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#addevent-mechanism)

The add event command is facilitated by LogicManager class. It allows user to add a new event to the event list.

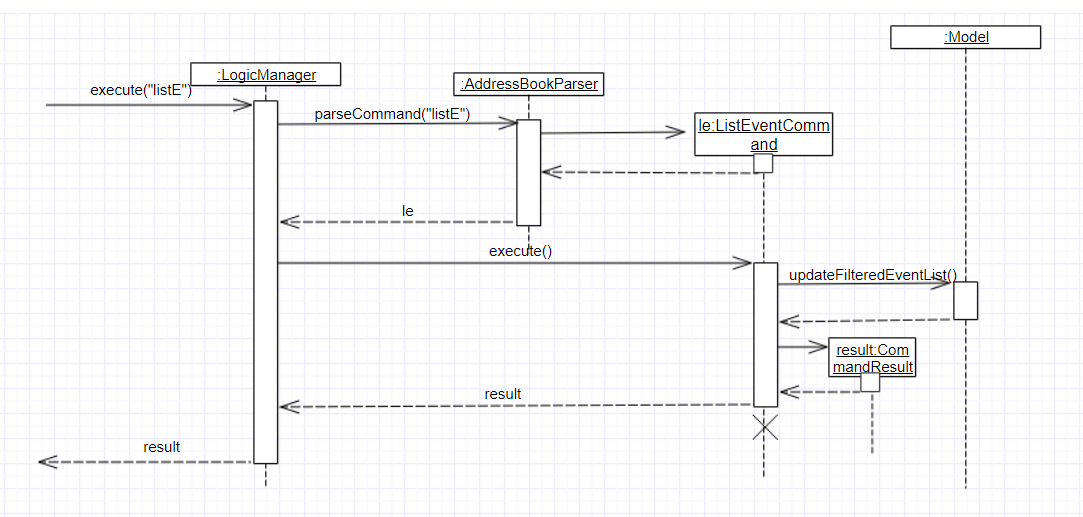
The following ***sequence diagram*** *(Figure 3.4)* shows how add event command works:

*Figure 3.4: AddEvent command sequence diagram*

The user enters an addEvent command, which is received by LogicManager. LogicManager calls AddressBookParser to parse the command.  
AddressBookParser will first check the format of the parameters. If the format is valid, it calls AddEventCommandconstructor, which returns an AddEventCommand object.  
Logic Manager receives the AddEventCommand object and calls execute(). Then, AddEventCommand uses Model to execute adding operation, and returns CommandResult.  
Logic Manager receives the CommandResult, and returns to UI.

## [3.5. ListEvent mechanism](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#listevent-mechanism)

The list event command is facilitated by the LogicManager class. It supports listing all the events for user to process.

The following ***sequence diagram*** *(Figure 3.5)* shows how the list event operation works:

*Figure 3.5: sequence diagram for list command*

The user enters a listE command which is received by LogicManager. LogicManager calls AddressBookParser to parse the command. Then AddressBookParser creates the list event command object and returns it. LogicManager receives the command object and executes the command. The list event command object calls updateFilteredEventList() in the Model. Then Model processes the event list. The list event command object creates CommandResult object and returns it.LogicManager receives the result and shows it through UI.

## [3.6. DeleteEvent mechanism](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#deleteevent-mechanism)

The delete event command is facilitated by the LogicManager. It supports undo and redo as it inherits from UndoableCommand. Other delete commands are implemented similarly.

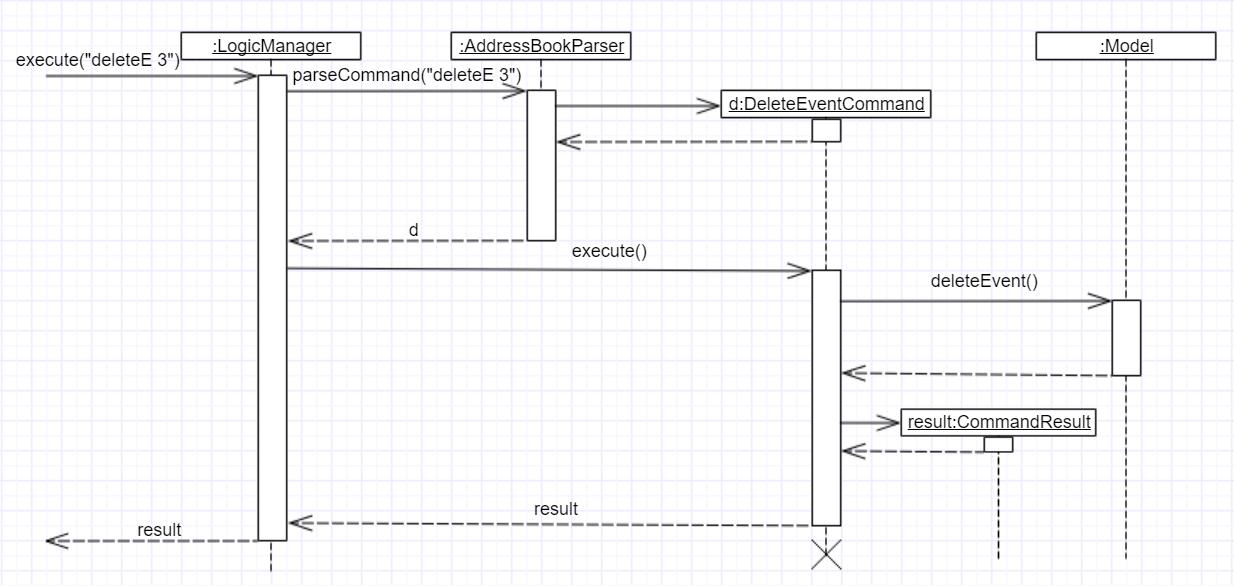
The following ***sequence diagram*** *(Figure 3.6)* below shows how the delete event operation works:

Figure 3.6: sequence diagram for deleteE command

The user enters a deleteE command which is received by LogicManger. LogicManager calls AddressBookParser to parse the user command. AddressBookParser creates an DeleteEventCommand object and returns it. LogicManager receives the Command object and executes it. The DeleteEventCommand calls deleteEvent() in the Model. Model will proceed to delete the event. The DeleteEventCommand then creates a CommandResult object and returns it to LogicManager.LogicManager receives the result and displays it through the UI.

### **Design Considerations**

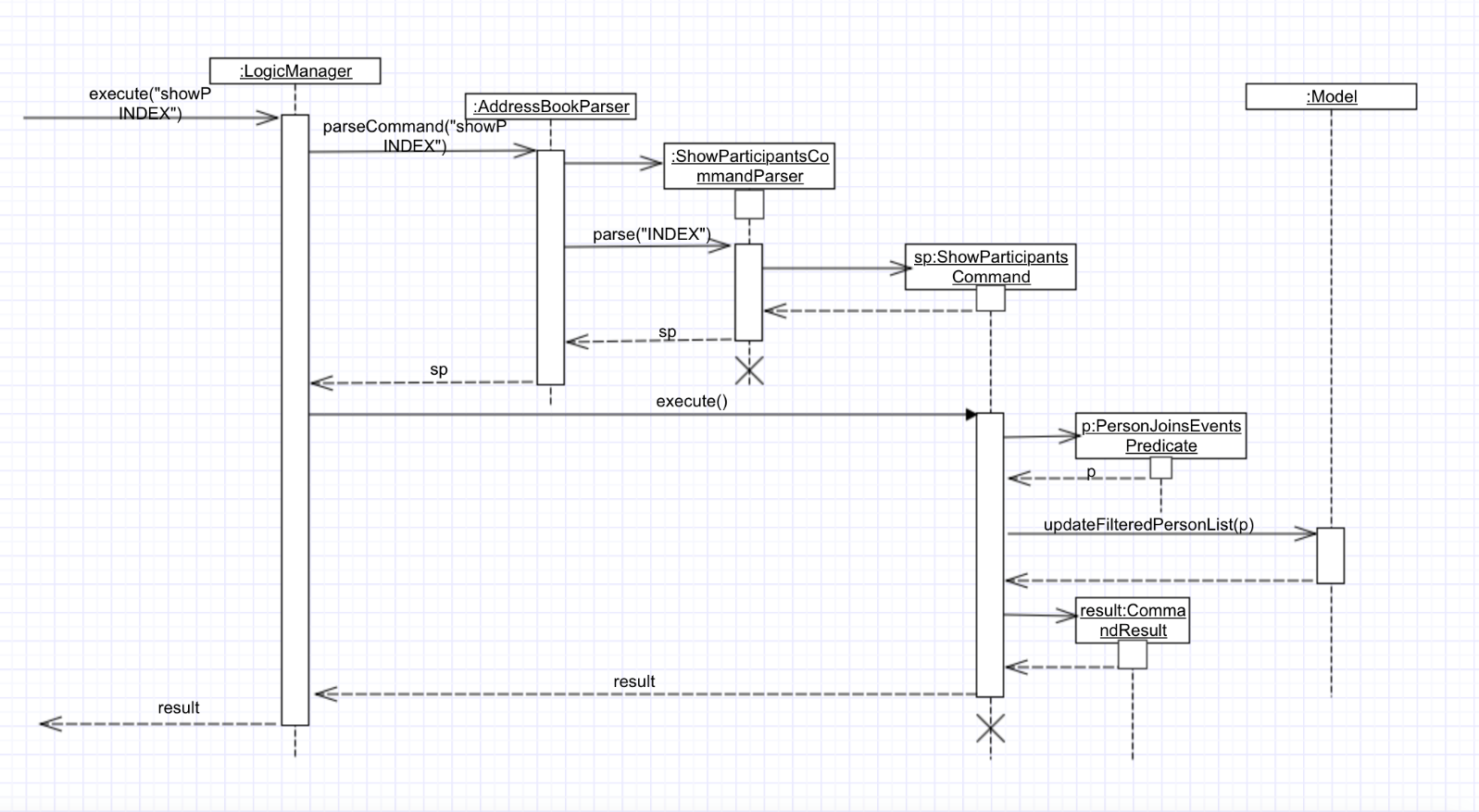
**Aspect:** Implementation of DeleteEventCommand.  
**Alternative 1 (current choice):** Create a new command to delete events.  
**Pros:** It is responsible only for deleting events, and not people. Easier to implement.  
**Cons:** Users may accidentally type in "delete" for deleting a person instead of "deleteE" for deleting an event.  
**Alternative 2:** Modify DeleteCommand to handle deletion for both persons and events.  
**Pros:** Users would be less likely to type the wrong command.  
**Cons:** Any changes to the deletion process of either a person or event may affect the other. Harder to implement.

**Aspect:** How deleteE command executes.  
**Alternative 1 (current choice):** Deletes the event at the specified index.  
**Pros:** Easy to imlement.  
**Cons:** User has to list/sort event list first.  
**Alternative 2:** Delete the event with the specified name.  
**Pros:** Users can delete directly without having to get the index.  
**Cons:** Users would have to type more if the event name is long. It will be more complex for developers to maintain and test.

## [3.7. ShowParticipants mechanism](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#showparticipants-mechanism)

The show participants mechanism is facilitated by PersonJoinsEventsPredicate and LogicManager. This command will help users to find participants of an event. For achieving this function, PersonJoinsEventsPredicate will filter the person list. This command inherits from Command.

The following ***sequence diagram*** *(Figure 3.7)* below show how the show participants operation works:



*Figure 3.7: sequence diagram for showP INDEX command*

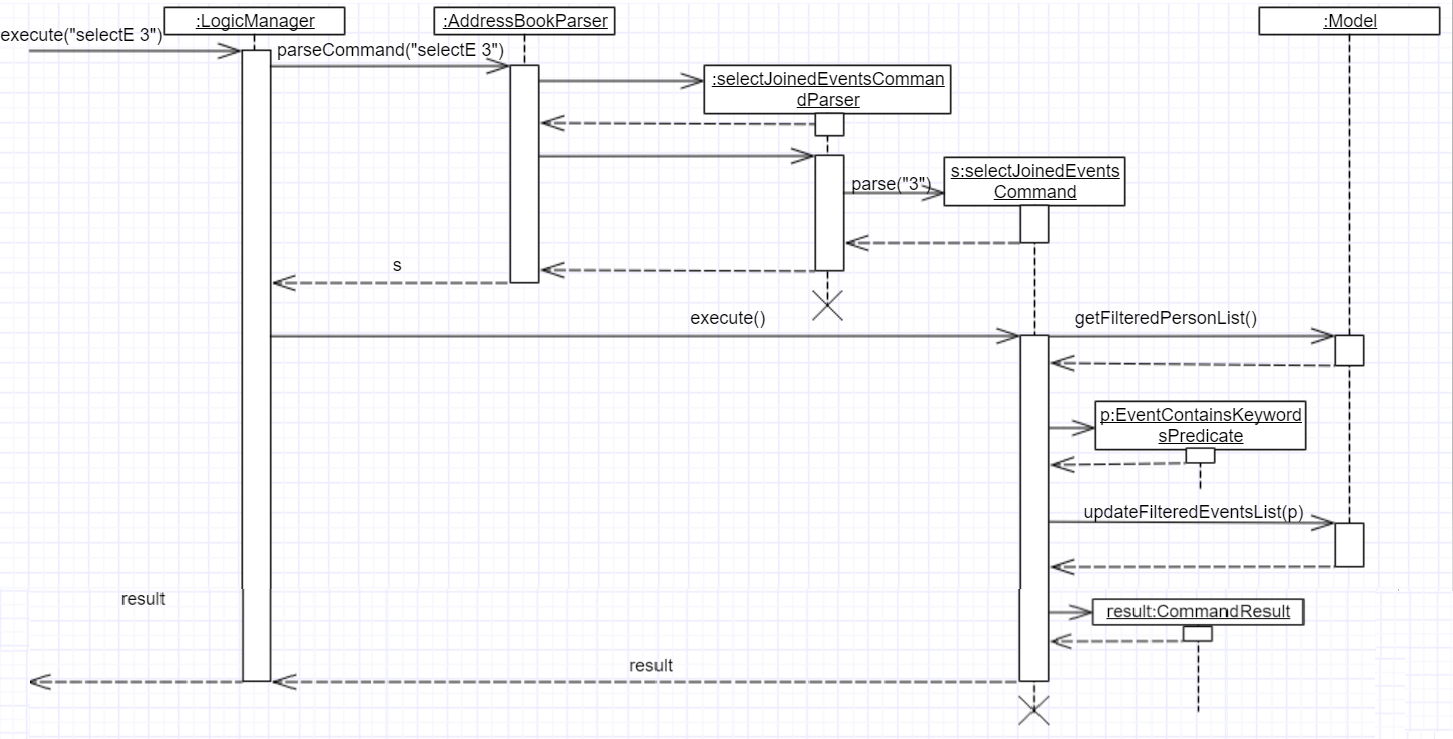
The user enters a showP INDEX command which is received by LogicManager. LogicManager calls AddressBookParser to parse the command. AddressBookParser will call ShowParticipantsCommandParser to parse the INDEX.ShowParticipantsCommandParser will create ShowParticipantsCommand object using the INDEX. Then AddressBookParserreceives the command object and returns it to LogicManager. LogicManager receives the command object and executes the command. ShowParticipantsCommand will create the PersonJoinsEventsPredicate according to the INDEX. Then, ShowParticipantsCommand calls updateFilteredPersonList(p) in the Model, which uses the predicate created before. Then Model processes the person list. The show participants command object creates CommandResult object and returns it. LogicManager receives the result and displays it through UI.

### **Design Considerations**

**Aspect:** How showP command executes  
**Alternative 1 (current choice):** Show participants of the event at the specified index.  
**Pros:** Easy to implement.  
**Cons:** Sometimes users have to list all the events so that they can get all the indexes.  
**Alternative 2:** Show participants of the event with the specified name.  
**Pros:** Users can use the command without listing events.  
**Cons:** Users need more time to type if the event name is too long.

## [3.8. ShowAllJoinedEvent mechanism](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#showalljoinedevent-mechanism)

The selectE command is facilitated by the LogicManager. This command inherits from Command.

The following ***sequence diagram*** *(Figure 3.8)* shows how the show all joined events operation works:

*Figure 3.8: sequence diagram for selectE command*

As seen in the ***sequence diagram*** *(Figure 3.8)* above, when the user enters a selectE command, the input is parsed by the SelectJoinedEventsCommandParser. A SelectJoinedEventsCommand is then created from the parsed user input. When the LogicManger executes the SelectJoinedEventsCommand , the latter calls getFilteredPersonList() to get the current person list. The SelectJoinedEventsCommand will then check the entered index values and get the names of the events the person/s are participants of. This is shown in the code snippet below:

**public** **class** SelectJoinedEventsCommand extends Command {

@Override

Public CommandResult execute() throws CommandException {

for (Index targetIndex: indexList) {

throw new CommandException(Messages.MESSAGE\_INVALID\_PERSON\_DISPLAYED\_INDEX);

}

personNames.append(lastShownList.get(targetIndex.getZeroBased()).getName().append(“, ”));

for (ReadOnlyEvent event:

lastShownList.get(targetIndex.getZeroBased()).getParticipation()) {

eventNames.append(event.getEventName()).append(“[-]”);

}

}

}

As seen in the code snippet above, the SelectJoinedEventsCommand will throw an exception if the entered index values are invalid and will combine the participated event names into a String named “eventNames”. The SelectJoinedEventsCommand will then separate the event names using the combination of symbols, “[-]”, so they can be used by a predicate to find the participated events. This is shown in the code snippet below:

**public** **class** SelectJoinedEventsCommand extends Command {

@Override

Public CommandResult execute() throws CommandException {

String[] eventNameKeywords = (eventNames.toString().trim()).split(“\\[-]+”);

EventContainsKeywordPredicate predicate = new

EventContainsKeywordPredicate(Arrays.asList(eventNameKeywords));

// Update the UI and return result.

}

}

As shown in the code snippet above, the event names will be separated and stored in a String[] named “eventNameKeywords”. A EventContainsKeywordPredicate will be created and will use the “eventNameKeywords” to find the events that are participated by the entered person/s. The UI will then be updated to display the events found.

### **Design Considerations**

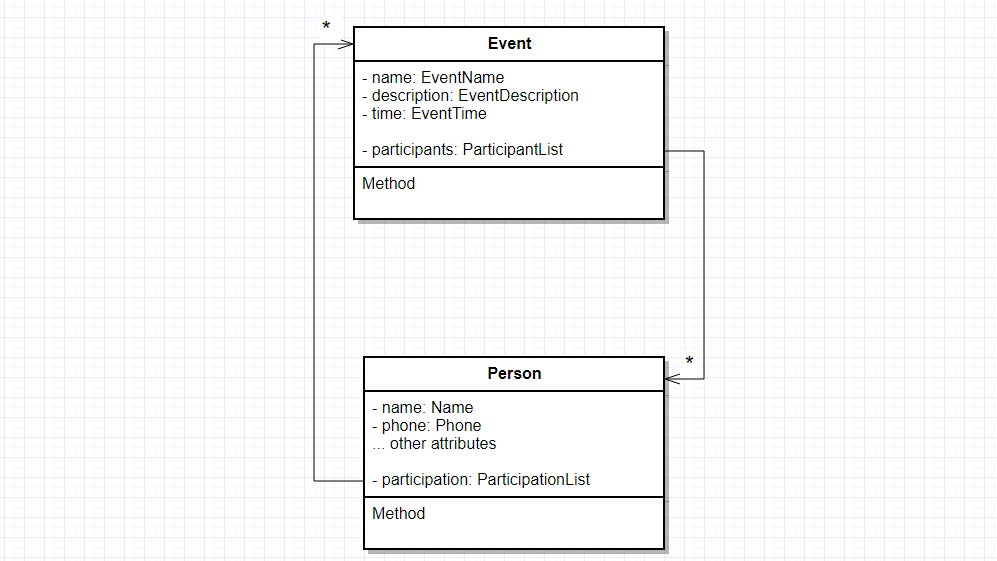
**Aspect:** How selectE command executes.  
**Alternative 1 (current choice):** Shows events joined by person/s at the selected indexes.  
**Pros:** The User does not have to type out names.  
**Cons:** The User has to list/sort person list first.  
**Alternative 2:** Show events joined by person/s using user entered name.  
**Pros:** User can directly enter names to search for events where a person with the entered name has joined.  
**Cons:** Users would have to type more if the name is long. User may mistype the name.

## [3.9. Person-Event interaction](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#person-event-interaction)

In this application, we allow users to connect events and persons to show that some persons participate some events.  
We will explain it in terms of Model and Storage.

### **Model level**

For model, we simply use the "referencing model", which is shown in the diagram *(Figure 3.9.1)* below:



*Figure 3.9.1: interaction model*

Referring to the diagram *(Figure 3.9.1)* above, there is a participant list in the Event model, which stores all the persons who are involved in this event. Similarly, the participation list in Person model stores all the event in which this person joins.

Hence, we need to maintain both lists when operating join and disjoin commands.  
The following code shows how disjoin operates:

**public** **void** disjoin(Person personToRemove, Event eventToRemove) {

eventList.remove(personToRemove); *//Maintain participantList in Event*

personList.remove(eventToRemove); *//Maintain participationList in Person*

*// Save changes to the storage.*

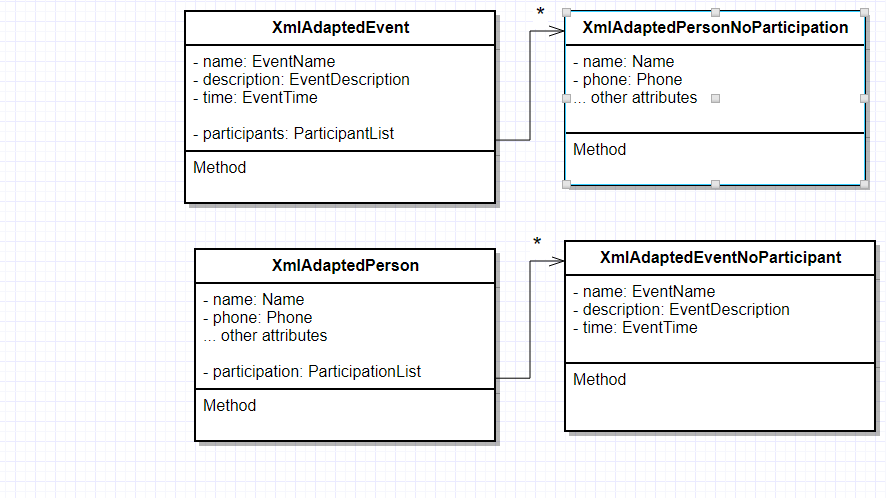
}

### **Storage level**

For storage, since persons and events reference each other, it is easy to have an **Infinity loop of reference**, as shown in the diagram *(Figure 3.9.2)* below:

image::DG\_infinity loop.png[width = "800"] *Figure 3.9.2: infinity loop*

To solve it, we create two storage entities: XmlAdaptedEventNoParticipant and XmlAdaptedPersonNoParticipation. Both of them do not have participant or participation information. Hence, we can avoid the infinity referencing problem.  
The updated storage is shown in the diagram *(Figure 3.9.3)* below:



*Figure 3.9.3: interaction storage*

### **Design Considerations**

**Aspect:** How to implement person-event interaction  
**Alternative 1 (current choice):** Set participationList in Person, and participantList in Event. They reference each other  
**Pros:** This is easy for us to implement.  
**Cons:** We need to maintain both lists when operating join and disjoin.  
**Alternative 2:** Use association class  
**Pros:** This allows us to save every participation entry.  
**Cons:** We need to implement extra storage for association class, and it is costly in terms of time to operate show person and select event.

## [3.10. Logging](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#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](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#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 are not usually noteworthy but may be useful in debugging e.g. print the actual list instead of just its size

## [3.11. Configuration](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#configuration)

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

# [4. Documentation](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#documentation)

We use asciidoc for writing documentation.

|  |  |
| --- | --- |
| NOTE | We choose asciidoc over Markdown, because asciidoc is more flexible in formatting. |

## [4.1. Editing documentation](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#editing-documentation)

See [UsingGradle.adoc](file:///C:\\Users\\Celia\\Documents\\planno\\build\\docs\\html5\\UsingGradle.html" \l "rendering-asciidoc-files) 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](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#publishing-documentation)

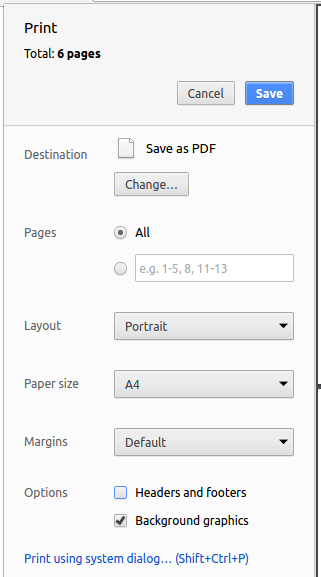
See [UsingTravis.adoc](file:///C:\\Users\\Celia\\Documents\\planno\\build\\docs\\html5\\UsingTravis.html" \l "deploying-github-pages) to learn how to deploy GitHub Pages using Travis.

## [4.3. Converting documentation to PDF format](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#converting-documentation-to-pdf-format)

We use [Google Chrome](https://www.google.com/chrome/browser/desktop/) 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](file:///C:\\Users\\Celia\\Documents\\planno\\build\\docs\\html5\\UsingGradle.html" \l "rendering-asciidoc-files) 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 *(Figure 4.3)* below.



*Figure 4.3: Saving documentation as PDF files in Chrome*

# [5. Testing](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#testing)

## [5.1. Running tests](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#running-tests)

There are three ways to run tests.

|  |  |
| --- | --- |
| NOTE | The most reliable method to run tests is Method 3.  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, 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](file:///C:\Users\Celia\Documents\planno\build\docs\html5\UsingGradle.html) for more information on how to run tests using Gradle. |

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

Thanks to the [TestFX](https://github.com/TestFX/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](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#types-of-tests)

We have two types of tests:

1. **GUI Tests** - These are tests involving the GUI. They include:
   1. *System Tests* that test the entire App by simulating user actions on the GUI. They are in systemtests package.
   2. *Unit tests* that test the individual components. They are in seedu.address.ui package.
2. **Non-GUI Tests** - These are tests not involving the GUI. They include:
   1. *Unit tests* targeting the lowest level methods/classes.  
      e.g. seedu.address.commons.StringUtilTest
   2. *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
   3. Hybrids of unit and integration tests. These test are checking multiple code units as well as how they are connected together.  
      e.g. seedu.address.logic.LogicManagerTest

## [5.3. Troubleshooting testing](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#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](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#dev-ops)

## [6.1. Build automation](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#build-automation)

See [UsingGradle.adoc](file:///C:\\Users\\Celia\\Documents\\planno\\build\\docs\\html5\\UsingGradle.html) to learn how to use Gradle for build automation.

## [6.2. Continuous integration](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#continuous-integration)

We use [Travis CI](https://travis-ci.org/) and [AppVeyor](https://www.appveyor.com/) to perform *Continuous Integration* on our projects. See [UsingTravis.adoc](file:///C:\\Users\\Celia\\Documents\\planno\\build\\docs\\html5\\UsingTravis.html) and [UsingAppVeyor.adoc](file:///C:\\Users\\Celia\\Documents\\planno\\build\\docs\\html5\\UsingAppVeyor.html) for more details.

## [6.3. Making a release](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#making-a-release)

Here are the steps to create a new release:

1. Update the version number in [MainApp.java](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/MainApp.java).
2. Generate a JAR file [using Gradle](file:///C:\Users\Celia\Documents\planno\build\docs\html5\UsingGradle.html#creating-the-jar-file).
3. Tag the repo with the version number. e.g. v0.1
4. [Create a new release using GitHub](https://help.github.com/articles/creating-releases/) and upload the JAR file you created.

## [6.4. Managing dependencies](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#managing-dependencies)

A project often depends on third-party libraries. For example, Address Book depends on the [Jackson library](http://wiki.fasterxml.com/JacksonHome) 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. Including those libraries in the repo (this bloats the repo size)  
b. Requiring developers to download those libraries manually (this creates extra work for developers)

# [Appendix A: Suggested programming tasks to get started](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#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](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#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. The section [Creating a new command: remark](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#creating-a-new-command-code-remark-code) explains how to go about adding such a feature.

## [A.1. Improving each component](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#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

|  |  |
| --- | --- |
| **Note** | Do take a look at the [Design: Logic Component](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#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 how we store each individual command word constant COMMAND\_WORD inside \*Command.java (e.g.[FindCommand#COMMAND\_WORD](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/logic/commands/FindCommand.java), [DeleteCommand#COMMAND\_WORD](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/logic/commands/DeleteCommand.java)), you need a new constant for aliases as well (e.g. FindCommand#COMMAND\_ALIAS).
     + [AddressBookParser](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/logic/parser/AddressBookParser.java) is responsible for analyzing command words.
   * Solution
     + Modify the switch statement in [AddressBookParser#parseCommand(String)](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/logic/parser/AddressBookParser.java) such that both the proper command word and alias can be used to execute the same intended command.
     + See this [PR](https://github.com/se-edu/addressbook-level4/pull/590/files) for the full solution.

Model component

|  |  |
| --- | --- |
| **Note** | Do take a look at the [Design: Model Component](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#model-component) section before attempting to modify the Model component. |

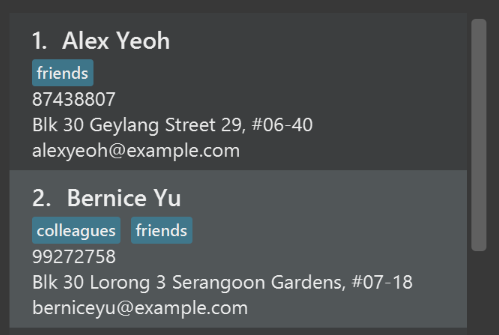
1. Add a removeTag(Tag) method. The specified tag will be removed from everyone in the address book.
   * Hints
     + The [Model](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/model/Model.java) API needs to be updated.
     + Find out which of the existing API methods in [AddressBook](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/model/AddressBook.java) and [Person](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/model/person/Person.java) classes can be used to implement the tag removal logic. [AddressBook](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/model/AddressBook.java) allows you to update a person, and [Person](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/model/person/Person.java) allows you to update the tags.
   * Solution
     + Add the implementation of deleteTag(Tag) method in [ModelManager](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/model/ModelManager.java). Loop through each person, and remove the tag from each person.
     + See this [PR](https://github.com/se-edu/addressbook-level4/pull/591/files) for the full solution.

UI component

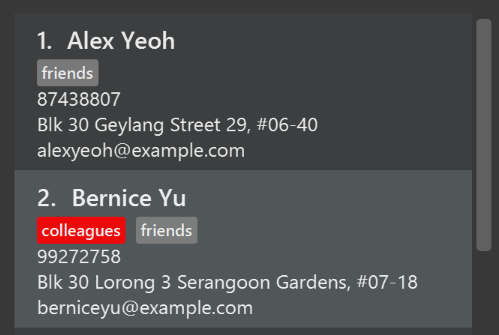
|  |  |
| --- | --- |
| **Note** | Do take a look at the [Design: UI Component](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#ui-component) section before attempting to modify the UI component. |

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

**Before**



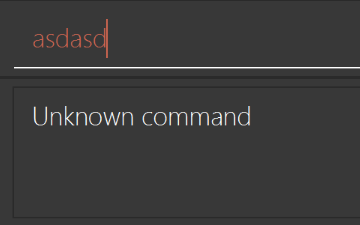
**After**



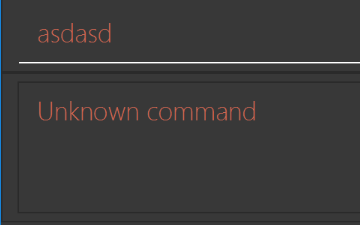
* + Hints
    - The tag labels are created inside [PersonCard#initTags(ReadOnlyPerson)](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/ui/PersonCard.java) (new Label(tag.tagName)). [JavaFX’s Label class](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/Label.html) 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](https://github.com/se-edu/addressbook-level4/pull/592/files) for the full solution.

1. Modify [NewResultAvailableEvent](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/commons/events/ui/NewResultAvailableEvent.java) such that [ResultDisplay](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/ui/ResultDisplay.java) can show a different style on error (currently it shows the same regardless of errors).

**Before**



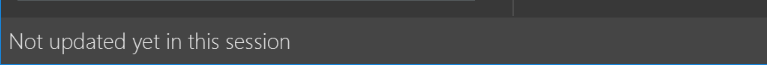
**After**



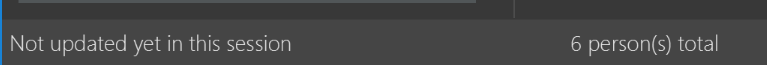
* + Hints
    - [NewResultAvailableEvent](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/commons/events/ui/NewResultAvailableEvent.java) is raised by [CommandBox](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/ui/CommandBox.java) which also knows whether the result is a success or failure, and is caught by [ResultDisplay](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/ui/ResultDisplay.java) which is where we want to change the style to.
    - Refer to [CommandBox](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/ui/CommandBox.java) for an example on how to display an error.
  + Solution
    - Modify [NewResultAvailableEvent](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/commons/events/ui/NewResultAvailableEvent.java) 's constructor so that users of the event can indicate whether an error has occurred.
    - Modify [ResultDisplay#handleNewResultAvailableEvent(event)](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/ui/ResultDisplay.java) to react to this event appropriately.
    - See this [PR](https://github.com/se-edu/addressbook-level4/pull/593/files) for the full solution.

1. Modify the [StatusBarFooter](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/ui/StatusBarFooter.java) to show the total number of people in the address book.

**Before**



**After**



* + Hints
    - [StatusBarFooter.fxml](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/resources/view/StatusBarFooter.fxml) will need a new StatusBar. Be sure to set the GridPane.columnIndex properly for each StatusBar to avoid misalignment!
    - [StatusBarFooter](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/ui/StatusBarFooter.java) 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](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/ui/StatusBarFooter.java) to take in the number of persons when the application just started.
    - Use [StatusBarFooter#handleAddressBookChangedEvent(AddressBookChangedEvent)](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/ui/StatusBarFooter.java) to update the number of persons whenever there are new changes to the addressbook.
    - See this [PR](https://github.com/se-edu/addressbook-level4/pull/596/files) for the full solution.

Storage component

|  |  |
| --- | --- |
| **Note** | Do take a look at the [Design: Storage Component](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#storage-component) section before attempting to modify the Storagecomponent. |

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](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/storage/AddressBookStorage.java) interface.
     + Implement the logic in [StorageManager](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/storage/StorageManager.java) class.
   * Solution
     + See this [PR](https://github.com/se-edu/addressbook-level4/pull/594/files) for the full solution.

## [A.2. Creating a new command: remark](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#creating-a-new-command-code-remark-code)

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**](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#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**](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#step-by-step-instructions)

[[Step 1] Logic: Teach the app to accept 'remark' which does nothing](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#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](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/logic/commands/UndoableCommand.java). Upon execution, it should just throw an Exception.
2. Modify [AddressBookParser](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/logic/parser/AddressBookParser.java) to accept a RemarkCommand.

**Tests:**

1. Add RemarkCommandTest that tests that executeUndoableCommand() throws an Exception.
2. Add new test method to [AddressBookParserTest](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/test/java/seedu/address/logic/parser/AddressBookParserTest.java), which tests that typing "remark" returns an instance of RemarkCommand.

[[Step 2] Logic: Teach the app to accept 'remark' arguments](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#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](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/logic/parser/AddressBookParser.java) 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](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/test/java/seedu/address/logic/parser/AddressBookParserTest.java) to test that the correct command is generated according to the user input.

[[Step 3] Ui: Add a placeholder for remark in PersonCard](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#step-3-ui-add-a-placeholder-for-remark-in-code-personcard-code)

Let’s add a placeholder on all our [PersonCard](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/ui/PersonCard.java) s to display a remark for each person later.

**Main:**

1. Add a Label with any random text inside [PersonListCard.fxml](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/resources/view/PersonListCard.fxml).
2. Add FXML annotation in [PersonCard](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/ui/PersonCard.java) to tie the variable to the actual label.

**Tests:**

1. Modify [PersonCardHandle](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/test/java/guitests/guihandles/PersonCardHandle.java) so that future tests can read the contents of the remark label.

[[Step 4] Model: Add Remark class](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#step-4-model-add-code-remark-code-class)

We have to properly encapsulate the remark in our [ReadOnlyPerson](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/model/person/ReadOnlyPerson.java) 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](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/model/person/Address.java), 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](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#step-5-model-modify-code-readonlyperson-code-to-support-a-code-remark-code-field)

Now we have the Remark class, we need to actually use it inside [ReadOnlyPerson](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/model/person/ReadOnlyPerson.java).

**Main:**

1. Add three methods setRemark(Remark), getRemark() and remarkProperty(). Be sure to implement these newly created methods in [Person](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/model/person/ReadOnlyPerson.java), which implements the [ReadOnlyPerson](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/model/person/ReadOnlyPerson.java) 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](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/model/util/SampleDataUtil.java/) 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](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#step-6-storage-add-code-remark-code-field-to-code-xmladaptedperson-code-class)

We now have Remark s for Person s, but they will be gone when we exit the application. Let’s modify [XmlAdaptedPerson](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/storage/XmlAdaptedPerson.java) 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](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/model/person/ReadOnlyPerson.java).

**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](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#step-7-ui-connect-code-remark-code-field-to-code-personcard-code)

Our remark label in [PersonCard](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/ui/PersonCard.java) is still a placeholder. Let’s bring it to life by binding it with the actual remarkfield.

**Main:**

1. Modify [PersonCard#bindListeners()](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/ui/PersonCard.java) to add the binding for remark.

**Tests:**

1. Modify [GuiTestAssert#assertCardDisplaysPerson(…​)](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/test/java/seedu/address/ui/testutil/GuiTestAssert.java) so that it will compare the remark label.
2. In [PersonCardTest](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/test/java/seedu/address/ui/PersonCardTest.java), call personWithTags.setRemark(ALICE.getRemark()) to test that changes in the [Person](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/model/person/ReadOnlyPerson.java) 's remark correctly updates the corresponding [PersonCard](https://github.com/CS2103AUG2017-W14-B4/main/tree/master/src/main/java/seedu/address/ui/PersonCard.java).

[[Step 8] Logic: Implement RemarkCommand#execute() logic](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#step-8-logic-implement-code-remarkcommand-execute-code-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**](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#full-solution)

See this [PR](https://github.com/se-edu/addressbook-level4/pull/599) for the step-by-step solution.

# [Appendix B: User stories](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#user-stories)

Priorities: High (must have) - \* \* \*, Medium (nice to have) - \* \*, Low (unlikely to have) - \*

| **Priority** | **As a …​** | **I want to …​** | **So that I can…​** |
| --- | --- | --- | --- |
| \* \* \* | user | add a person to an event | keep track of who is involved |
| \* \* \* | user | delete a person from an event | remove a person who is no longer participating |
| \* \* \* | user | list all events a person is involved in | easily check which events a person is a participant of |
| \* \* \* | user | list every person involved in an event | easily check who is participating |
| \* \* \* | new user | see usage instructions | refer to instructions when I forget how to use the App |
| \* \* \* | user | add a new person | add a person’s contact detail into the app |
| \* \* \* | user | add a new event | add a event’s date and information into the app |
| \* \* \* | user | delete a person or event | remove entries that I no longer need |
| \* \* \* | user | edit a person or event | change some information of the person or event |
| \* \* \* | user | clear my address book | refresh it quickly |
| \* \* \* | user with many friends | list all friends with a certain tag | group my friend by tag easily |
| \* \* \* | user | add a person event with [blank information](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#blank-information) | add him or the event even if I do not know some details (e.g his address) |
| \* \* \* | user | find a person by name and tag | locate details of persons without having to go through the entire list |
| \* \* \* | user | get help information | know where is wrong when I get errors |
| \* \* \* | user | update contact’s information | correct or add their information |
| \* \* \* | user | list contacts | view who are my contacts |
| \* \* \* | user | list events | view what are my events |
| \* \* \* | secretive user | hide private details | minimise the chance of someone else seeing them by accident |
| \* \* \* | careless user | undo decision | revert changes in case of mistake |
| \* \* | frequent user | change the font and background colour | use address book comfortably |
| \* \* | frequent user | use non-case sensitive command | type command easily |
| \* \* | user | sort contacts | view my list easily |
| \* \* | user | tag my contacts | remember who they are through tags |
| \* \* | secretive user | hide [private contact details](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#private-contact-detail)by default | minimize chance of someone else seeing them by accident |
| \* \* | forgetful user | stick some important people on the top | locate them quickly |
| \* \* | user | create events and assign contacts | keep track of who is involved in events |
| \* \* | user | create filter using multiple tags | make a specific search using tags |
| \* | user | add pictures to contacts | remember who they are through pictures |

{More to be added}

# [Appendix C: Use cases](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#use-cases)

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

## **Use case: Add a person**

**MSS**

1. User enters values for a new person.
2. Planno adds the new person into the database.

Use case ends.

**Extensions**

* 2a. There is already this person.

Use case ends.

* 2b. The list is empty.

Use case ends.

* 3a. The given index is invalid.
  + Planno shows an error message.

Use case resumes at step 2.

## **Use case: Delete person**

**MSS**

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

Use case ends.

**Extensions**

* 2a. The list is empty.

Use case ends.

* 3a. The given index is invalid.
  + Planno shows an error message.

Use case resumes at step 2.

## **Use case: Update person**

**MSS**

1. User find target person by his name.
2. Planno shows the result of find command.
3. User types new information for the target person.
4. Planno updates the information and displays successful message.

Use case ends.

**Extensions**

* 2a. The target person cannot be found.

Use case ends.

* 3a. User types invalid information.
  + Planno shows an error message.

Use case resumes at step 2.

* 3b. New information is the same as original.
  + Planno shows an unnecessary update message.

Use case ends.

## **Use case: Add a tag to a person**

**MSS**

1. User enters values for a new person.
2. Planno add the new person into the database.

Use case ends.

**Extensions**

* 2a. There is already this person.

Use case ends.

## **Use case: Undo a command**

**MSS**

1. User enters "history" command to view all history commands.
2. Planno displays all history commands.
3. User enters "undo" to request to undo the last [undoable command](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#undoable-command).
4. Planno undoes the last [undoable command](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#undoable-command) and displays successful message.

Use case ends.

**Extensions**

* 3a. No more [undoable command](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#undoable-command) commands in the history.
  + Planno displays error message.

Use case ends.

## **Use case: Redo a command**

**MSS**

1. User enters "history" command to view all history commands.
2. Planno displays all history commands.
3. User enters "redo" to request to undo the last [redoable command](file:///C:\\Users\\Celia\\Documents\\planno\\build\\docs\\html5\\DeveloperGuide.html" \l "redoable-command).
4. Planno undoes the last [redoable command](file:///C:\\Users\\Celia\\Documents\\planno\\build\\docs\\html5\\DeveloperGuide.html" \l "redoable-command) and displays successful message.

Use case ends.

**Extensions**

* 3a. No more [redoable command](file:///C:\\Users\\Celia\\Documents\\planno\\build\\docs\\html5\\DeveloperGuide.html" \l "redoable-command) commands in the history.
  + Planno displays error message.

Use case ends.

### **Use case: Clear Planno**

**MSS**

1. User enters "clear" command to delete all entries in address book.
2. Planno deletes all data.

Use case ends.

### **Use case: Show help window**

**MSS**

1. User enters value for help command.
2. Planno displays the help window.

Use case ends.

**Extensions**

* 1a. User presses F1 on the keyboard or clicks on help icon.

Use case resumes at step 2.

### **Use case: Exit the app**

**MSS**

1. User enters "exit" command for exiting the app.
2. Planno terminates its work and exit.

Use case ends.

**Extensions**

* 1a. User click File → Exit button.

Use case resumes at step 2.

### **Use case: Sort persons**

**MSS**

1. User enters "sort" command.
2. Planno shows a list of sorted persons.

Use case ends.

**Extensions**

* 2a. The list is empty.
  + Planno shows a successful message.

Use case ends.

### **Use case: List persons**

**MSS**

1. User enters "list" command.
2. Planno displays list of persons.

Use case ends.

**Extensions**

* 1a. There is no person in Planno.
  + Planno shows a no person message.

Use case ends.

### **Use case: List events**

**MSS**

1. User enters "listE" command.
2. Planno displays list of persons.

Use case ends.

**Extensions**

* 1a. There is no event in EventList.
  + Planno shows a no event message.

Use case ends.

### **Use case: Show participants of an event**

**MSS**

1. User requests to list events.
2. Planno shows a list of events.
3. User requests to show participants of a specific event in the list.
4. Planno shows participants of the event.

Use case ends.

**Extensions**

* 2a. The list is empty.

Use case ends.

* 3a. The given index is invalid.
  + Planno shows an error message.

Use case resumes at step 2.

### **Use case: Find person/s by name**

**MSS**

1. User enters values for find command.
2. Planno displays persons with name matching at least one keyword.

Use case ends.

**Extensions**

* 1a. There is no person with a name matching any keyword.
  + Planno shows a no person found message.

Use case ends.

### **Use case: Select person**

**MSS**

1. User enters value for list command.
2. Planno displays list of persons.
3. User enters index value for select command.
4. Planno displays Google search page of the person at the entered index value.

Use case ends.

**Extensions**

* 1a. There is no person in Planno.
  + Planno shows a no person message.

Use case ends.

* 1b. User enters value for find command.
  + Planno displays persons with name matching at least one keyword.

Use case resumes at step 3.

* 3a. User enters invalid index value.
  + Planno displays an invalid index message.

Use case ends.

### **Use case: List command history**

**MSS**

1. User enters value for history command.
2. Planno displays a list of commands the User has entered from most recent to earliest.

Use case ends.

**Extensions**

* 1a. User has not entered any previous commands.
  + Planno displays a no previous command entered message.

Use case ends.

### **Use case: Find a person by tag**

**MSS**

1. User enters tag names for a person list.
2. Planno displays the list of persons that contains either of the tag names.

Use case ends.

**Extensions**

* 1a. User does not enter tag names.
  + Planno shows an error message.

Use case ends.

* 1b. User does not use correct format.
  + Planno displays an empty list.

Use case ends.

### **Use case: Select all joined events**

**MSS**

1. User enters value for list command.
2. Planno displays list of persons.
3. User enters value for list event command.
4. Planno displays list of events.
5. User enters index value/s for selectJoinedEvents command.
6. Planno displays all events of the person/s at the entered index value/s.

Use case ends.

**Extensions**

* 1a. There is no person in Planno.
  + Planno shows a no person message.

Use case ends.

* 1b. User enters value for find command.
  + Planno displays persons with name matching at least one keyword.

Use case resumes at step 3.

* 3a. There is no event in Planno.
  + Planno displays a blank event list.

Use case resumes at step 5.

* 5a. User enters invalid index value.
  + Planno displays an invalid index message.

Use case ends.

* 5b. The selected person has not joined any events.
  + Planno displays a blank event list.

Use case ends.

# [Appendix D: Non-functional requirements](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#non-functional-requirements)

1. The application should work on any [mainstream OS](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#mainstream-os) as long as it has Java 1.8.0\_60 or higher installed.
2. The application 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.
4. The application should be able to give feedback message within 2 seconds for every command.
5. The user should be able to use the application without connecting to Internet.
6. The user should be able to use the application without any programming knowledge background.
7. The size of this application should not be larger than 10GB.
8. The application should work on both 32-bit and 64-bit environment.
9. The cost of this application should not be more than 10,000 dollars.
10. The application should protect users' privacy.
11. Font size should be suitable for all people aged 6 to 70 years old.
12. The application should not tolerate data loss.
13. The application should be usable to a novice who has never used such type of application before.
14. The application should startup within 7 seconds.
15. Background color should be appropriate to not discomfort users' eyes.

{More to be added}

# [Appendix E: Glossary](file:///C:\Users\Celia\Documents\planno\build\docs\html5\DeveloperGuide.html#glossary)

API

An Application Programming Interface (API) specifies the interface through which other programs can interact with a software component. It is a contract between the component and its clients.

Blank information

An information field with null value. It may happen when a user adds a person and does not know some information.

Build automation

Build automation is the process of automating the creation of a software build and the associated processes.

CI

Continuous integration(CI) is an extreme application of build automation in which integration, building, and testing happens automatically after each code change.

Gradle

Gradle is an open source build automation system that automates the creation of a software build.

Logging

Logging is the deliberate recording of certain information during a program execution for future reference. It can be useful for troubleshooting problems.

Mainstream OS

Windows, Linux, Unix, OS-X.

Private contact detail

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

Redoable command

A redoable command is a command which has been undone, and no other commands in between.

Travis

Travis CI is a hosted, distributed continuous integration service used to build and test projects hosted at GitHub.

Undoable command

An undoable command is a command which modifies data in Planno. For example, add, delete are undoable commands, while find and list are not.

Last updated 2017-10-27 18:02:15 +08:00