Lau Kar Rui - Project Portfolio

Project: iungo

iungo is a desktop address book application used for learning Software Engineering principles. The user interacts with it using a CLI, and it has a GUI created with JavaFX. It is written in Java, and has about 6 kLoC.

Code contributed: [Functional code] [Test code] {give links to collated code files}

Enhancement Added: Setting of homepage

External behavior

Start of Extract [from: User Guide] written primarily by original author, with relevant homepage information by me

Editing a person : `edit`, `e`

Edits an existing person in the address book.
Format: `edit INDEX [n/NAME] [p/PHONE] [e/EMAIL] [a/ADDRESS] [h/HOMEPAGE] [t/TAG]…`

- Edits the person at the specified `INDEX`. The index refers to the index number shown in the last person listing. The index **must be a positive integer** 1, 2, 3, …
- At least one of the optional fields must be provided.
- Existing values will be updated to the input values.
- When editing tags, the existing tags of the person will be removed i.e adding of tags is not cumulative.
- You can remove all the person’s tags by typing `t/` without specifying any tags after it.
- You can reset the homepage to the default homepage by typing `h/` without specifying any URL after it.

NOTE If the name is edited and the contact did not previously have a custom homepage set, the default homepage will switch to a Google search of the new name. If there was a custom homepage set, the homepage will not change

Examples:

- `edit 1 p/91234567 e/johndoe@example.com`
  Edits the phone number and email address of the 1st person to be `91234567` and `johndoe@example.com` respectively.
- `e 2 n/Betsy Crower t/`
  Edits the name of the 2nd person to be `Betsy Crower` and clears all existing tags.

Adding a person: `add`, `a`

Adds a person to the address book
Format: `add n/NAME p/PHONE_NUMBER [e/EMAIL] [a/ADDRESS] [h/HOMEPAGE] [t/TAG]…`

TIP A person can have any number of tags (including 0)

TIP The `EMAIL`, `ADDRESS`, `HOMEPAGE`, and `TAG` parameters are OPTIONAL

NOTE A person will have a default homepage of a Google search of his/her name, if `/h` was not included in the add command

Examples:

- `add n/John Doe p/98765432 e/johnd@example.com a/John street, block 123, #01-01 h/http://www.johndoe.com`
- `add n/Betsy Crowe t/friend a/Newgate Prison p/1234567 t/criminal`
- `a n/Jane Doe p/87654321 e/janede@example.com`

End of Extract

Justification

Previous behaviour of both add and edit did not allow the user to set their homepage to something more useful.

This implementation now enhances the usability of the application by allowing the user to set relevant homepages for each individual contact, such as their Facebook or LinkedIn profiles.

Implementation

Start of Extract [from: Developer Guide]

Setting of homepage for a `Person`

The mechanism to set a homepage for a specified `Person` relies on `AddCommand` and `EditCommand`.
It supports both the setting and resetting of a homepage. Resetting a homepage returns the homepage to the default homepage of a Google search of the Person’s full name.

NOTE Care is also given to make sure the homepage is changed when the name is edited if the current homepage is the default homepage (i.e. not manually set).

NOTE If the homepage has been set before, it will not change until it is reset by `h/` or a new homepage is manually set.

`AddCommand` and `EditCommand` both checks for the `h/` parameter that indicates whether the current homepage is to be modified.
If `h/` is parsed to be empty (`""` by `AddCommandParser` or `EditCommandParser`, a `Person` constructor is used to create the person with the default homepage.
If `h/` is a non-empty valid URL (determined by `Homepage.isValidHomepage`, a different `Person` constructor is invoked to create a person with the set homepage.
`AddCommandParser` code snippet to determine if user’s `AddCommand` contains `h/` parameter:

```
if (arePrefixesPresent(argMultimap, PREFIX_HOMEPAGE)) {
    Homepage homepage =
ParserUtil.parseHomepage(argMultimap.getValue(PREFIX_HOMEPAGE)).get();
    person = new Person(name, phone, email, address, tagList, homepage);
} else {
    person = new Person(name, phone, email, address, tagList);
}
```

`EditCommand` will create an `EditPersonDescriptor` with the arguments entered, and pass the resulting `EditPersonDescriptor` into a method to create the updated `Person`.
`EditCommand.createEditedPerson` code snippet to check if homepage has been manually set before:

```
private static Person createEditedPerson(ReadOnlyPerson personToEdit,
EditPersonDescriptor editPersonDescriptor) {
    Homepage originalHomepage = personToEdit.getHomepage();

    Homepage updatedHomepage =
editPersonDescriptor.getHomepage().orElse(personToEdit.getHomepage());

    // ... other logic...

    if (updatedHomepage.value.equals(RESET_HOMEPAGE)) {
        return new Person(updatedName, updatedPhone, updatedEmail, updatedAddress,
updatedTags);
    }
    if (personToEdit.isHomepageManuallySet() ||
!(originalHomepage.toString().equals(updatedHomepage.toString()))) {
        return new Person(updatedName, updatedPhone, updatedEmail, updatedAddress,
updatedTags, updatedHomepage);
    } else {
        return new Person(updatedName, updatedPhone, updatedEmail, updatedAddress,
updatedTags);
    }
}
```

Design Considerations

Aspect: Implementation of homepage changing when name of contact is changed
Alternative 1 (Current choice): Change homepage to a Google search of the name name when name is changed if homepage has not been manually set prior

Pros: Consistent behaviour - if name changes but the default homepage was still referring to the old name, user will be confused.
Cons: New developers will have to take note of the extra `Homepage` check when enhancing or refactoring `AddCommand` or `EditCommand`
Alternative 2: `Homepage` will not be changed after creation.
Pros: Less complexity in the code, easier for new developers to handle.
Cons: Results in a less user friendly application.

End of Extract

Enhancement Added: Sort

External behavior

Start of Extract [from: User Guide]

Sorting the contact list : `sort`

{since v1.2}
Sorts the contact list in either [a]scending or [d]escending order and shows the list.
Format: `sort [a / d]`

- The parameters are OPTIONAL. `sort` on its own will default to a sort in ascending order.

End of Extract

Justification

Previously, the `list` view of the application would show contacts in the order they were added. For an address book with many contacts added, sorting would allow the user (who, for some reason, does not want to use the `find` command) to be able to view the contact list in alphabetical ascending/ descending order.

Implementation

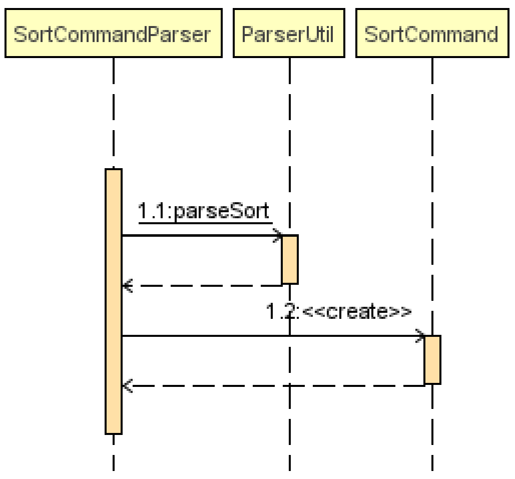
Start of Extract [from: Developer Guide]

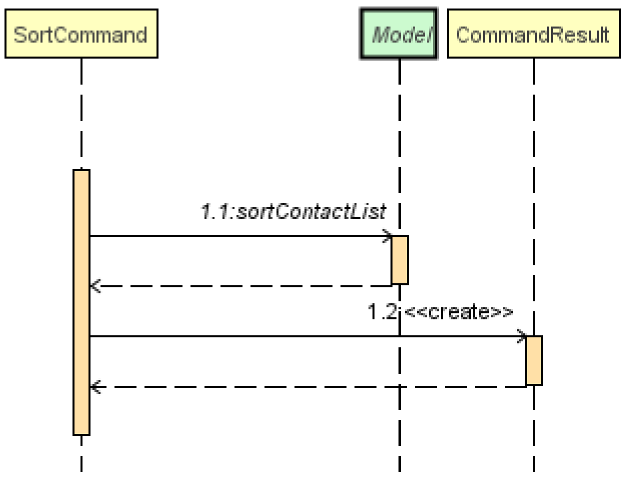
Sort mechanism

The sort mechanism is facilitated by the `SortCommand`, which is an `UndoableCommand`.

`SortCommandParser` checks for the OPTIONAL `a` or `d` parameters, and calls the relevant `SortCommand` to sort the `internalList` backing the address book.

The `internalList` is sorted using the full name of the contact using Java’s `sort`.
The following sequence diagrams shows how the sort mechanism works:





End of Extract

Enhancement Added: Set avatar

External behavior

Start of Extract [from: User Guide]

Setting an avatar for a contact : `setavatar`, `sa`

{since v1.3}
Sets an avatar for a contact referenced by the index number used in the last person listing.
Format: `setavatar INDEX sa/AVATAR_URL`
Alias: `sa INDEX sa/AVATAR_URL`

NOTE This requires an active Internet connection to work, as the application requires an URL to retrieve the image.

- Sets the avatar for the contact at the specified `INDEX`.
- The index refers to the index number shown in the most recent listing.
- The index **must be a positive integer** `1, 2, 3, …`
- If `AVATAR_URL` is empty; i.e `""`, the avatar will be removed and the default avatar will be shown
- The image size **must not be bigger than 50KB**.

Examples:

- `setavatar 1 sa/https://i.imgur.com/xPHOeWL.png`
  Sets the avatar of the 1st person listed to be the image as referenced by https://i.imgur.com/xPHOeWL.png.

End of Extract

Justification

Setting an avatar is a natural extension of an address book application

Implementation

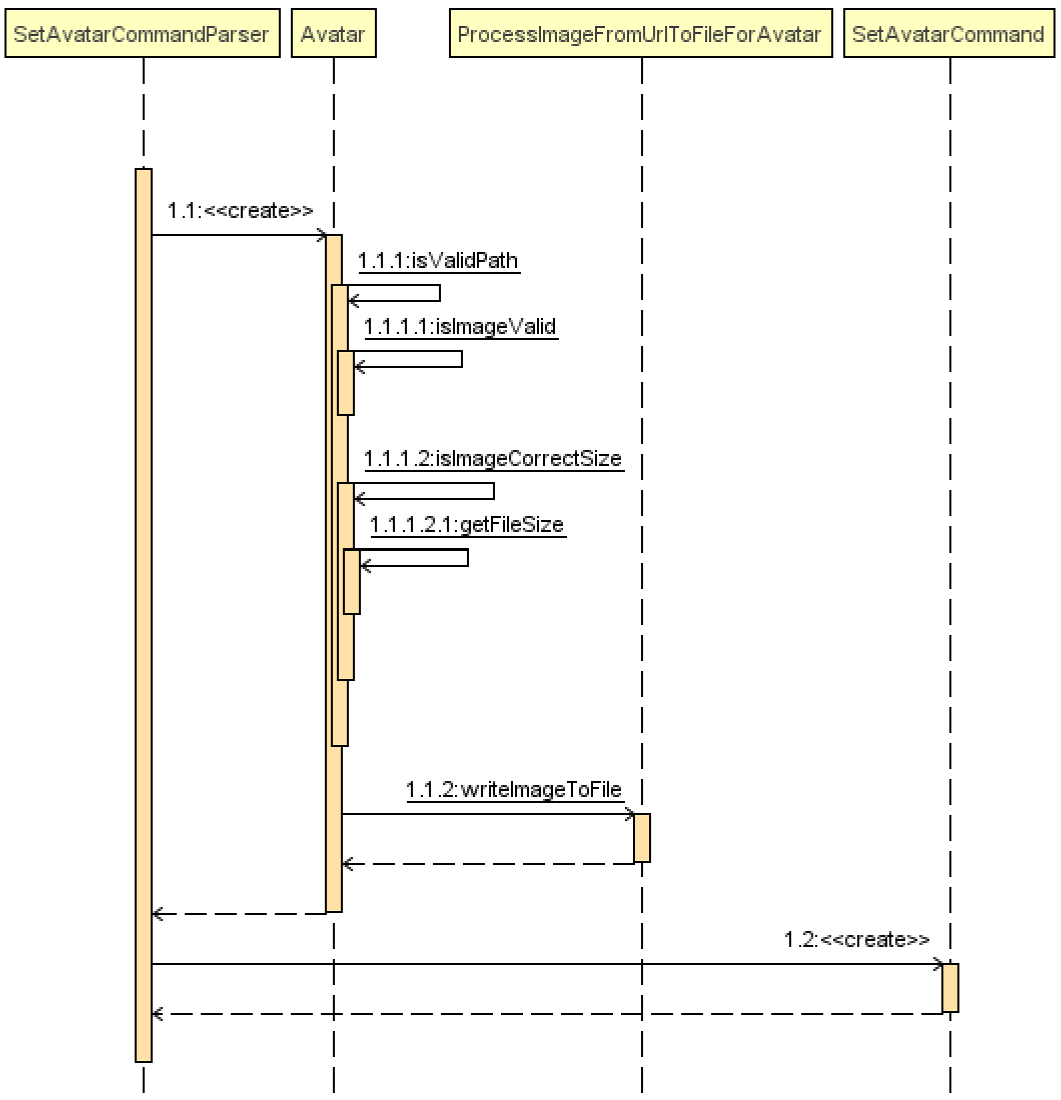
Start of Extract [from: Developer Guide]

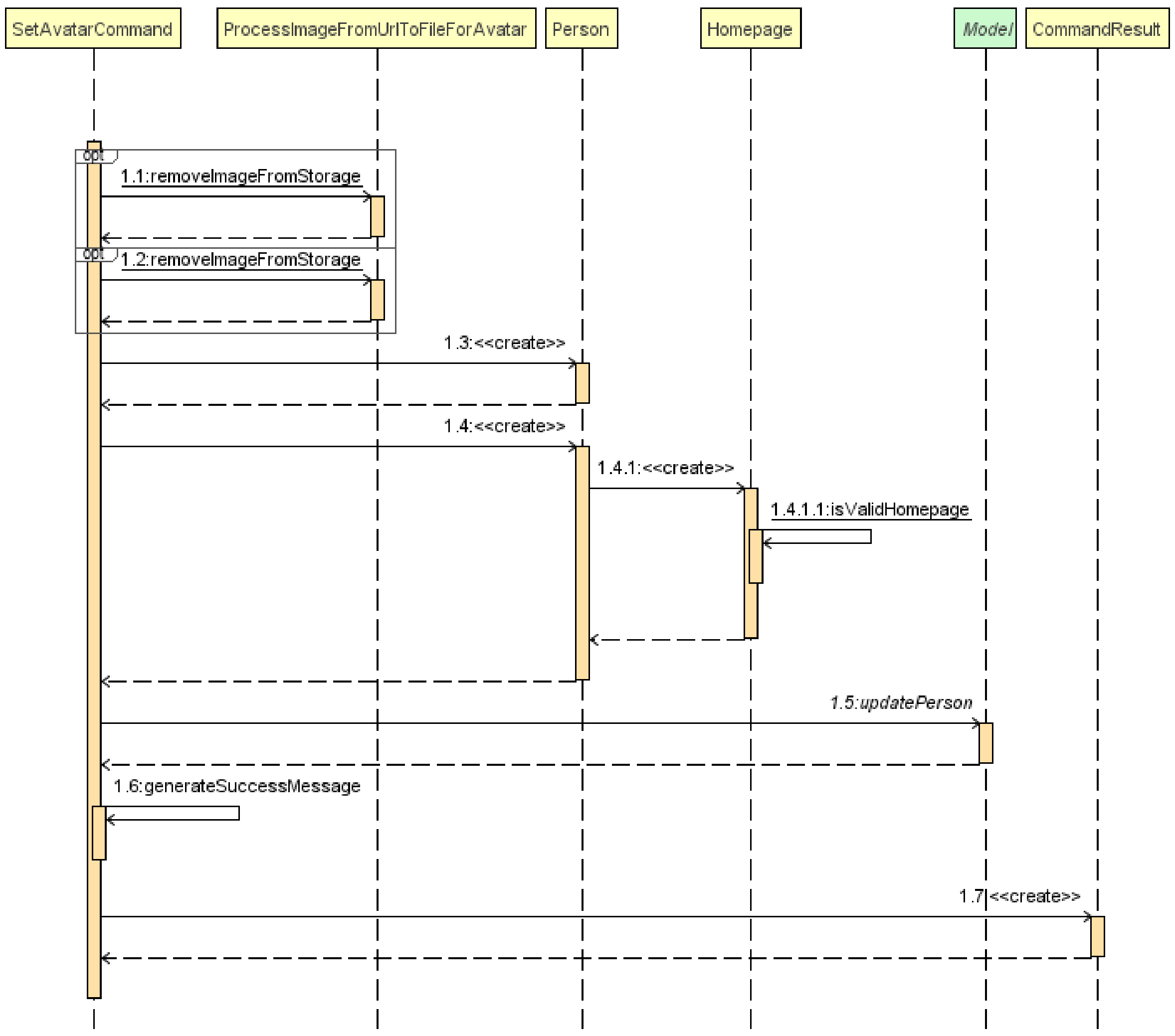
Avatar mechanism

Logic and Model implementation

The avatar mechanism is facilitated by the using `ImageView` in `JavaFX` to display the avatar in the UI. The main driver to create an `Avatar` is handled by `SetAvatarCommandParser`, which is invoked after `AddressBookParser` parses the arguments provided by the user.

The following sequence diagrams shows how the setting of avatar is achieved:


SetAvatarCommandParser
Avatar
ProcessImageFromUrlToFileForAvatar
SetAvatarCommand
1.1:<<create>>
1.1.1:isValidPath
1.1.1.1:isImageValid
1.1.1.2:isImageCorrectSize
1.1.1.2.1:getFileSize
1.1.2:writeImageToFile
1.2:<<create>>



The arguments needed are `INDEX` and `AVATAR_URL`.
An `Avatar` object is created using `AVATAR_URL` before being passed as a parameter into the creation of the `SetAvatarCommand` object.

The `Avatar` class performs a series of validity checks to ensure the URL provided is valid. Validity is defined by having:

1. A valid URL OR is empty
2. The image must not be larger than 20KB (due to application slowdowns if the file is too big)

`SetAvatarCommand` also performs a series of checks in order to decide the steps to take before editing the `Person` referenced.
Below is the code snippet of `SetAvatarCommand` for the various checks:

```
 public CommandResult executeUndoableCommand() throws CommandException {
     // ... irrelevent logic ...

     // avatar refers to the avatar object created during the construction of
SetAvatarCommand
     if ("".equals(avatar.path) && !"".equals(personToSetAvatarPath)) { // delete
image from storage

ProcessImageFromUrlToFileForAvatar.removeImageFromStorage(personToSetAvatarPath);
         } else {
             if (!"".equals(personToSetAvatarPath)) {   // has a previously set
avatar, remove first before processing

ProcessImageFromUrlToFileForAvatar.removeImageFromStorage(personToSetAvatarPath);
             }
         }
     }
 }
```

The utility class `ProcessImageFromUrlToFileForAvatar` is used to process images retrieved from the Internet. It contains two methods — `writeImageToFile(…)` and `removeImageFromStorage(…)` which stores the image into the `DEFAULT_AVATAR_FILE_LOCATION` and removes the image respectively.
Below is the code snippet of the write method:

```
void writeImageToFile(String path) {
    // ... irrelevant ...

    // Using hashCode() + checking if file exists assures uniqueness of name of
created file
    File file = new File(DEFAULT_AVATAR_FILE_LOCATION + path.hashCode() + ".jpg");
    while (file.exists()) {
        file = new File(DEFAULT_AVATAR_FILE_LOCATION + (path.hashCode() + ++i) +
".jpg");
    }
    ImageIO.write(image, "jpg", file);
    return file.getPath().replace('\\', '/');
}
```

As `SetAvatarCommand` is an `UndoableCommand`, `removeImageFromStorage(…)` only deletes the image when the application exits, in order to allow the user to undo the command.
Below is the code snipper of the remove method:

```
void removeImageFromStorage(String path) {
    File file = new File(path);
    file.deleteOnExit();    // so as to allow undoable command
}
```

Storage implementation

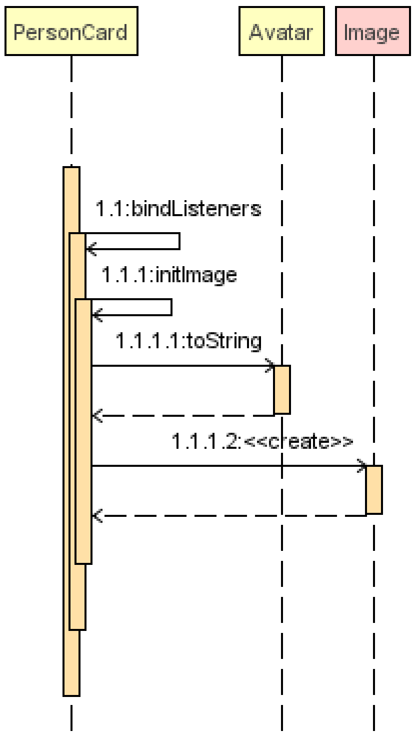
`AvatarStorage` is called during `Model` initialization to check for the existence of an `avatar` folder. If the folder does not exist, the folder will be created to store the avatar images.
Below is the code snippet for the relevant code flow:

```
private Model initModelManager(Storage storage, UserPrefs userPrefs, AvatarStorage
avatarStorage) {
    // ... other logic ...

    avatarDirectoryPath = userPrefs.getAvatarFileDirectoryPath();
    if (!new File(avatarDirectoryPath).exists()) {
        logger.info("Directory not found. Will be attempting to create new
directory");
        avatarStorage.createDirectory(avatarDirectoryPath);
    }
    // ... other logic ...
}
```

Ui implementation

The `PersonCard` is updated via the `bindListeners()`. The loose sequence diagram is displayed below:



Below is a more in-depth look at `initImage()`:

```
/**
 * Binds the correct image to the person.
 * If url is "", default display picture will be assigned, else image from URL will be
assigned
 */
private void initImage(ReadOnlyPerson person) {
    String path = person.getAvatar().toString();
    Image image;
    if (!"".equals(path)) {   // not default image
        File file = new File(path);
        image = new Image(file.toURI().toString());
        avatar.setImage(image);
        avatar.setFitHeight(90);
        avatar.setPreserveRatio(true);
        avatar.setCache(true);
    }
}
```

Design Considerations

Aspect: Saving of image from URL to local disk
Alternative 1 (Current choice): File is saved during construction of `Avatar` in `SetAvatarCommand`, before the Avatar is set to a `Person`
Pros: Filepath (not URL) of avatar class will be ascertained during binding to `editedPerson`
Cons: New developers might find it hard to follow the sequence
Alternative 2: Only save file after `Avatar` has been assigned to `Person`, assign URL as path when assigning before file is saved.
Pros: Easier for developers to follow the sequence flow
Cons: Possibility of program crashing (`NullPointerException`) if URL is added and Internet connection is disrupted before file creation is invoked.

Aspect: Image source
Alternative 1 (Current choice): Only accept image URL from the Internet
Pros: Easy for user to input source; Check guarantees validity of image from "HEAD" request
Cons: Requires internet connection.
Alternative 2: Accept images from user’s local disk in addition from the Internet
Pros: Natural extension of a function to set avatar.
Cons: Difficult for user to input source; difficulty in writing code to ascertain if file entered is entered.

End of Extract

Enhancement Added: Recent contacts

External behavior

Start of Extract [from: User Guide]

Display recently searched contacts : `recent`, `rc`

{since v1.4)

Shows a list of all contacts that was returned by `find` command since application was started.
Format: `recent`

End of Extract

Justification

Finding recently searched for contacts is a natural extension of an address book application, provides convenience to user.

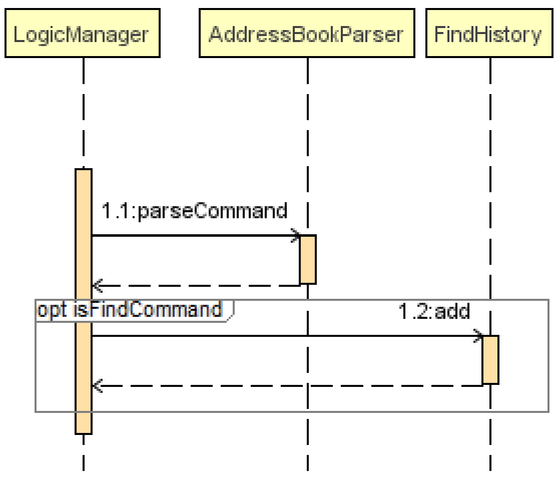
Implementation

Start of Extract [from: Developer Guide]

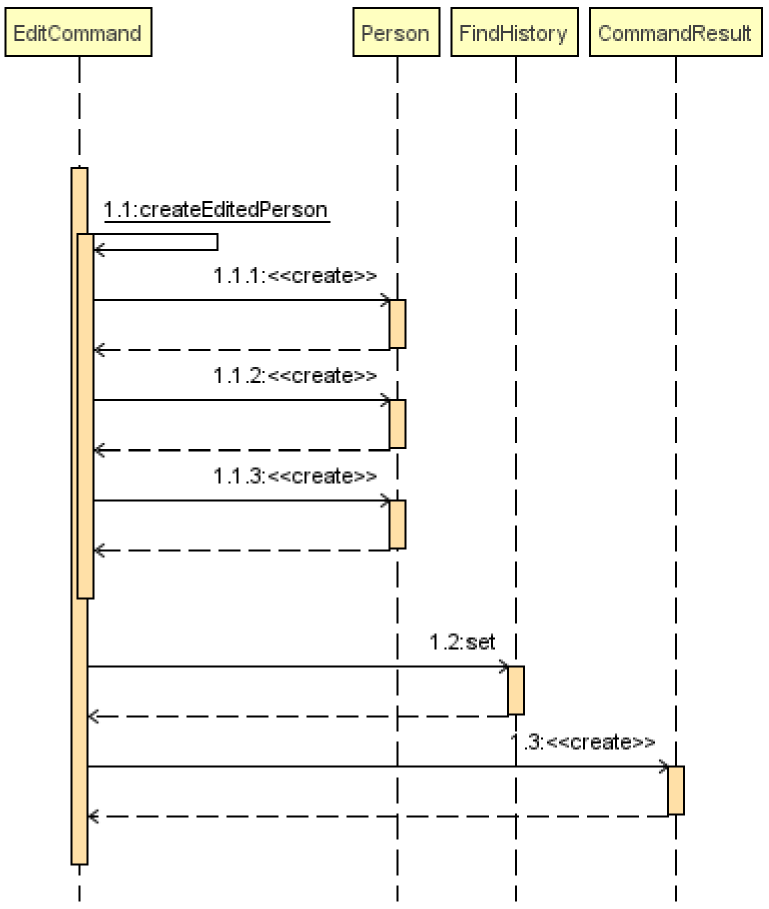
Recent command mechanism

`RecentCommand` uses a backing list from the `FindHistory` class to create predicates on which the `Model` will filter on and display to the user.
`FindHistory` is implemented with a LinkedList which is updated when relevant commands change persons returned by `FindCommand`.

When `FindCommand` is invoked, the `LogicManager` adds the entired `FilteredList` into the LinkedList in `FindHistory`.



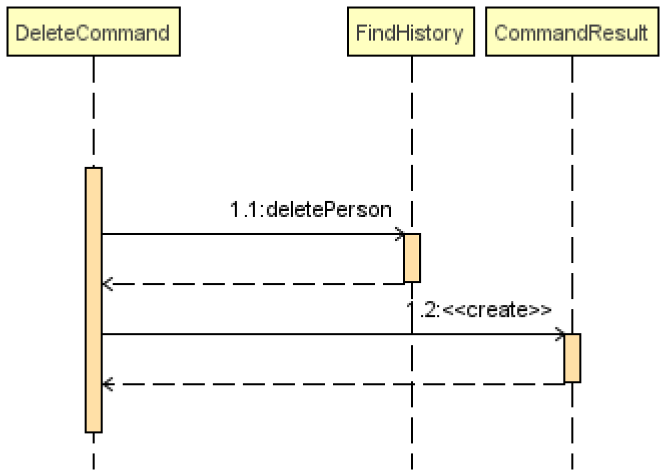
When `EditCommand` is invoked, the `set` method in `FindHistory` is invoked to replace the old `Person` with the edited `Person`.



The code snippet for the `set` method follows:

```
/**
* Changes {@code person} to {@code newPerson}
*/
public void set(ReadOnlyPerson person, ReadOnlyPerson newPerson) {
    if (userFindHistory.contains(person)) {
        userFindHistory.set(userFindHistory.indexOf(person), newPerson);
    }
}
```

When `DeleteCommand` is invoked, the `deletePerson` method in `FindHistory` is invoked to remove the person from the LinkedList.



A snapshot of the the LinkedList is also saved into `previousFindHistory` when an `UndoableCommand` is invoked, to preserve the changes in the backing list.

Design Considerations

Aspect: Order of list view to show to user
Alternative 1: Shows the list reverse order of persons returned by FindCommand (similar to HistoryCommand)
Pros: Intuitive for user, better UX
Cons: Requires an additional hook to `Model` to swap between two lists (addressbook and findhistory), additional complexity and chance of regressions
Alternative 2 (Current choice): Shows the list in current order of addressbook backing list.
Pros: Follows design logic of all commands that shows the list to user, allowing new developers to easily understand implementation.
Cons: Worse UX.

Aspect: Data structure to store `FindHistory` backing list
Alternative 1 (Current choice): LinkedList
Pros: Preserves order, fast add, able to use set to preserve ordering when mutating elements. Good for future implementations (if any) of returning the list in order user searched for contacts.
Cons: Slightly slower than competition when traversing and deleting.
Alternative 2: LinkedHashMap or LinkedHashSet
Pros: Keeps order, faster than LinkedList
Cons: Unable to use set, which is used when user edits a person.

End of Extract

Enhancement Added: Optional add command

External behavior

Start of Extract [from: User Guide] written by original author as my enhancement is just a modification of the method

Adding a person: `add`, `a`

Adds a person to the address book
Format: `add n/NAME p/PHONE_NUMBER [e/EMAIL] [a/ADDRESS] [h/HOMEPAGE] [t/TAG]…`

TIP A person can have any number of tags (including 0)

TIP The `EMAIL`, `ADDRESS`, `HOMEPAGE`, and `TAG` parameters are OPTIONAL

NOTE A person will have a default homepage of a Google search of his/her name, if `/h` was not included in the add command

Examples:

- `add n/John Doe p/98765432 e/johnd@example.com a/John street, block 123, #01-01 h/http://www.johndoe.com`
- `add n/Betsy Crowe t/friend a/Newgate Prison p/1234567 t/criminal`
- `a n/Jane Doe p/87654321 e/janede@example.com`

End of Extract

Justification

User might not always have all the details of the contact he/she wants to add. Only the name and phone number is compulsory.

Implementation

Start of Extract [from: Developer Guide]

Optional Add mechanism

The enhancement is based on the existing `AddCommandParser`, with a more relaxed restriction of only the `Name` and `Phone` fields being mandatory.

The relevant modifications are as follows:

```
public AddCommand parse(String args) throws ParseException {
    // ... other logic ...
    // only name and phone are compulsory
    if (!arePrefixesPresent(argMultimap, PREFIX_NAME, PREFIX_PHONE)) {
        throw new ParseException(String.format(MESSAGE_INVALID_COMMAND_FORMAT,
AddCommand.MESSAGE_USAGE));
    }
    try {
        Name name = ParserUtil.parseName(argMultimap.getValue(PREFIX_NAME)).get();
        Phone phone = ParserUtil.parsePhone(argMultimap.getValue(PREFIX_PHONE)).get();
        Email email =
ParserUtil.parseEmail(ParserUtil.parseValues(argMultimap.getValue(PREFIX_EMAIL))).get(
);
        Address address =
ParserUtil.parseAddress(ParserUtil.parseValues(argMultimap.getValue(PREFIX_ADDRESS))).
get();
        Set<Tag> tagList = ParserUtil.parseTags(argMultimap.getAllValues(PREFIX_TAG));
        // ... other logic ...
        return new AddCommand(person);
    } catch (IllegalValueException ive) {
        throw new ParseException(ive.getMessage(), ive);
    }
}
```

`parse()` utilizes `ParserUtil.parseValues(…)` to return an empty string if no such parameters were entered by the user, as seen below:

```
public static Optional<String> parseValues(Optional<String> value) {
    return Optional.of(value.orElse(STRING_IF_EMPTY));
}
```

The relevant parameters are then passed to create a `Person` object.

End of Extract

Other contributions

Wrote user stories for iungo

Volunteering own features for reuse

Recent command: [forum post]
Sort functionality: [forum post]
Avatar command: [forum post]

Helped in finding bugs

Bug discovered in AB4: [forum post]
Bugs discovered in Planno: [issues page]

Helping others with issues

Posts: [issues page]

{To be included in the future}