Benjamin Ang - Project Portfolio

Project: AddressBook - Level 4

AddressBook - Level 4 is a desktop address book application used for teaching 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: [https://github.com/CS2103AUG2017-F10-B3/main/blob/master/collated/main/NUSe0032202.md] [https://github.com/CS2103AUG2017-F10-B3/main/blob/master/collated/test/NUSe0032202.md]

Enhancement Added: Sort Name

External behavior

Start of Extract [from: User Guide]

Sorting the addressbook: `sort`

Sorts the addressbook by address or by name.User can specify if they want to save the sorted state.
Format: `sort OPTION SAVE_OPTION` , for quick input use the letter `s` instead of `sort` .

- OPTION keyword allows the user to select whether to sort the addressbook by name or by address.
- SAVE_OPTION keyword allows the user to save the sorted state of the addressbook and it is an optional input.
- The sort command is tied to the undo/redo mechanism to facilitate convenience in the event the user enters the wrong sort option.
- To change or edit the lookup table data, locate the .csv file in `\main\src\main\resources\tables` folder and do the necessary changes on the file.
- All changes to the .csv file have to conform to the present format which is each value starts on a new line.
- **DO NOT RENAME THE FILE OR CHANGE THE FILE LOCATION BEFORE BUILDING THE JAR FILE.**

Examples:

- `s n`
  Sorts the addressbook by name alphabetically.
- `sort a`
  Sorts the addressbook by address, in particular by the street name.
- `s n se`
  Sorts the addressbook by name and saves the sorted state of the addressbook.
- `s n` followed by `undo`
  Sorts the addressbook by name alphabetically.The next input undoes the sort reverting the addressbook to the previous state.

End of Extract

Justification

To have a better overview of contacts in the addressbook.Sorting by name will sort the addressbook by name in alphabetical order. Sorting by address will sort the addressbook by address in particular by their street names in alphabetical order. As sorting by address is base on a look up table of street names in Singapore, contacts whose addresses that cannot be found will be placed at the end of the addressbook.

Implementation

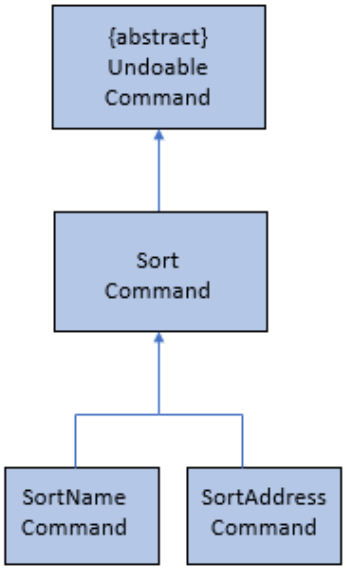
Start of Extract [from: Developer Guide]

Sort by name mechanism

The sorting by name mechanism is facilitated by implementing a comparator followed by overriding the compare method in the java class.The compare method is overridden in the Java class in this way:

```
public class SortName implements Comparator<ReadOnlyPerson> {
    @Override
    public int compare(ReadOnlyPerson a, ReadOnlyPerson b) {
      return a.getName().fullName.compareToIgnoreCase(b.getName().fullName);
    }
}
```

Suppose the user has just launched the application and executes a new `SortCommand sort n` to sort the addressbook by name. A new `SortCommand` parser object would be first instantiated which parsers the user's arguments and returns a new `SortCommand` object. Base on the inheritance diagram below a new SortName object is instantiated from the SortCommand class.



From within the `SortName` object the `sortAddressBook()` method than gets called which resides in the `ModelManager` class. The method is implemented this way:

```
public synchronized void sortAddressBook(int option, int saveOption) {
        addressBook.sort(option);
        updateFilteredPersonList(sortListPredicate);
        if (saveOption == SAVE) {
            indicateAddressBookChanged();
        }
    }
```

The parameter `option` being the user's choice to sort by name or address and `saveOption` being the user's choice for saving the addressbook state.

updateFilteredPersonList(sortListPredicate) is to ensure only the previously shown persons are shown after the command is executed.

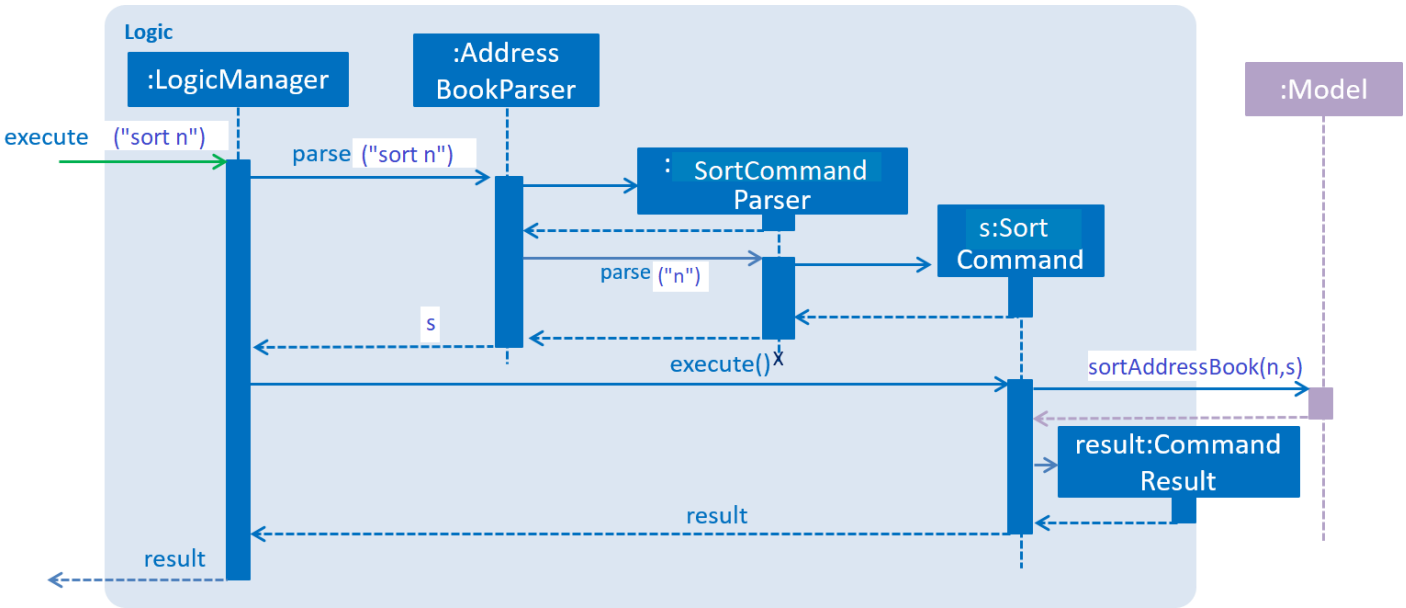
Due to the many layers in place, the `sort` method has to be called which resides in the `AddressBook` class which simply just operates as a link to the `UniquePersonList` class where the actual sorting logic takes place. The method `sort` within the `UniquePersonList` class is implemented this way:

```
public void sort(int option) throws AddressBookIsEmpty, LoadLookUpTableException {
        try {
            if (!internalList.isEmpty()) {
                if (option == OPTION_NAME) {
                    Collections.sort(internalList, new SortName());
                } else {
                    Collections.sort(internalList, new SortAddress());
                }
            } else {
                throw new AddressBookIsEmpty();
            }
        } catch (IOException e) {
            throw new LoadLookUpTableException();
        }
    }
```

AddressBookIsEmpty exception gets thrown if the user has cleared the addressbook and is attempting a sort command.

LoadLookUpTable exception can be ignored for now as it would be explained in the next segment.

As shown, sorting is done with using the Java API method `Collections.sort()` . The below diagram summarizes the entire command execution from the start to finish.



Sort by address mechanism

The sorting by address mechanism is backed by an Arraylist lookup table and is facilitated by implementing a comparator followed by overriding the compare method in the java class.The compare method is overridden in the Java class in this way:

```
public class SortAddress implements Comparator<ReadOnlyPerson> {
   @Override
   public int compare(ReadOnlyPerson a, ReadOnlyPerson b) {
           String compareFirst = "";
           String compareSecond = "";
           Boolean firstAddressFound = false;
           Boolean secondAddressFound = false;

           for (String compare : table) {
               if (a.getAddress().value.toLowerCase().contains(compare.toLowerCase())) {
                   compareFirst = compare;
                   firstAddressFound = true;
               }
           }

           for (String compare : table) {
               if (b.getAddress().value.toLowerCase().contains(compare.toLowerCase())) {
                   compareSecond = compare;
                   secondAddressFound = true;
               }
           }

           if (firstAddressFound == false) {
               return 1;
           }

           if (secondAddressFound == false) {
               return -1;
           }

           return compareFirst.compareToIgnoreCase(compareSecond);
       }
}
```

The two enhanced `for` loop is for comparison against the lookup table and should a match be made, the street name is noted down for comparison later. The two boolean values are to note whether a match can be found in the lookup table. In the event no match can be found, the person would get "pushed" to the bottom of the shown list. Hence a value of 1 or -1 is returned for either if their boolean variable equates to `false`.

The present data for the lookup table only contains the street names of Singapore.

To load the date into the ArrayList a utility class `AddressData` is used to parse inputs from the .csv file which contains all the lookup table data its implementation is as follows:

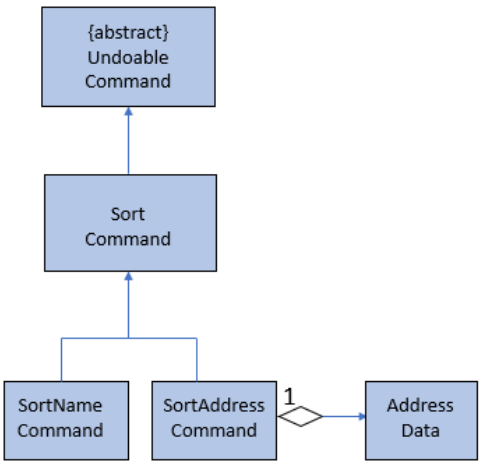
```
public class AddressData {

    private static BufferedReader br;
    private static String line;
    private static ArrayList<String> table = new ArrayList<>();

    public static void initTable() throws IOException {
        InputStream in = MainApp.class.getResourceAsStream("/tables/data.csv");
        br = new BufferedReader(new InputStreamReader(in));
        while ((line = br.readLine()) != null) {
            table.add(line.trim());
        }
    }

    public static ArrayList<String> getTable() {
        return table;
    }
}
```

LoadLookUpTable exception get thrown in the event an error occurs while reading in data from the .csv file.

When a new SortAddress object is instantiated both the methods `initTable()` and `getTable()` gets called attached below shows them getting called under the default constructor `SortAddress()`:

```
public class SortAddress implements Comparator<ReadOnlyPerson> {
    public SortAddress() throws IOException {
        AddressData.initTable();
        table = AddressData.getTable();
    }
}
```

Association diagram:



The command flow that happens after the user executes a new `SortCommand sort a` is the same as described in the SortName mechanism the only thing that varies are the arguments. The `sortAddressBook()` method gets called within the `SortAddress` class which leads to the `sort` method in the `Addressbook` class being called followed by the the `sort` method being called last in the `UniqueuePersonList` class where the actual sorting takes place.

End of Extract

Enhancement Added: Copy

External behavior

Start of Extract [from: User Guide]

Copying person's information over to system clipboard : `copy`

Copies the selected person's information over to the system clipboard. Format: `copy INDEX [CHOICE]` , for quick input use the letter `c` instead of `copy` .

> - Both fields after commnad word input `copy` or `c` must be provided.
> - To access the copied data, simply use `ctrl v` to paste the copied data.

Examples:

- `c 1 n`
  Copies the topmost person's name.
- `c 2 p`
  Copies the second topmost person's phone number.
- `copy 2 e`
  Copies the second topmost person's email.
- `copy 2 a`
  Copies the second topmost person's address.

End of Extract

Justification

To allow the user more convenience if he wants to copy over the data field of a particular person.

Implementation

Start of Extract [from: Developer Guide]

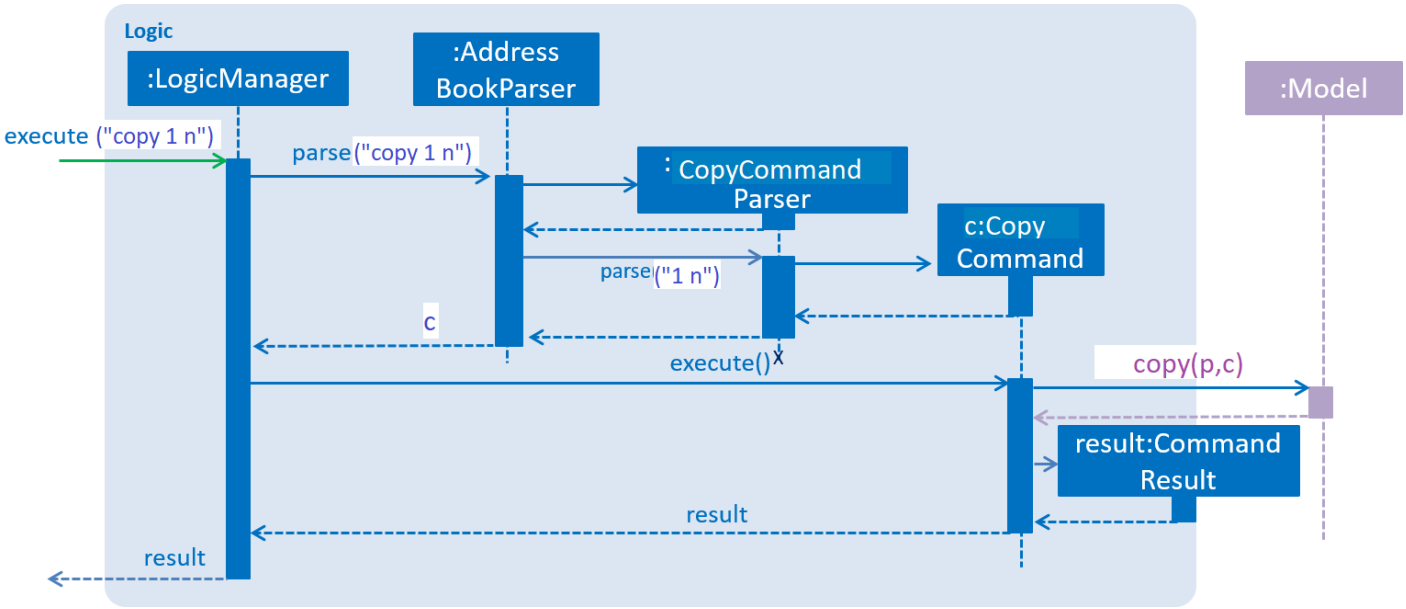
Copy Command mechanism

The copy command mechanism is backed by both the Java `Toolkit` class as well as the system clipboard. The copy method which resides in the `ModelManager` class and is implemented this way:

```
public void copy(ReadOnlyPerson person, String choice) {
    Person target = new Person(person);
    StringSelection sel;

    switch (choice) {
    case CHOICE_NAME:
        sel = new StringSelection(target.getName().fullName);
        clipboard.setContents(sel, null);
        break;

    case CHOICE_PHONE:
        sel = new StringSelection(target.getPhone().value);
        clipboard.setContents(sel, null);
        break;

    case CHOICE_EMAIL:
        sel = new StringSelection(target.getEmail().value);
        clipboard.setContents(sel, null);
        break;

    case CHOICE_ADDRESS:
        sel = new StringSelection(target.getAddress().value);
        clipboard.setContents(sel, null);
        break;

    default:
        break;
    }
}
```

Suppose the user has just launched the application and executes a new `CopyCommand copy 1 n`. A new `CopyCommand` parser object would be first instantiated which parsers the user's arguments and returns a new `CopyCommand` object. The method `copy` gets called and according to the user's choice copies over to the system clipboard the appropriate data via getter methods. `StringSelection` is used as it implements the `Transferable` interface which is appropriate for this situation. To access the copied data just use `ctrl v`.

Attached below is a diagram which describes the entire command execution from the start to finish.



End of Extract

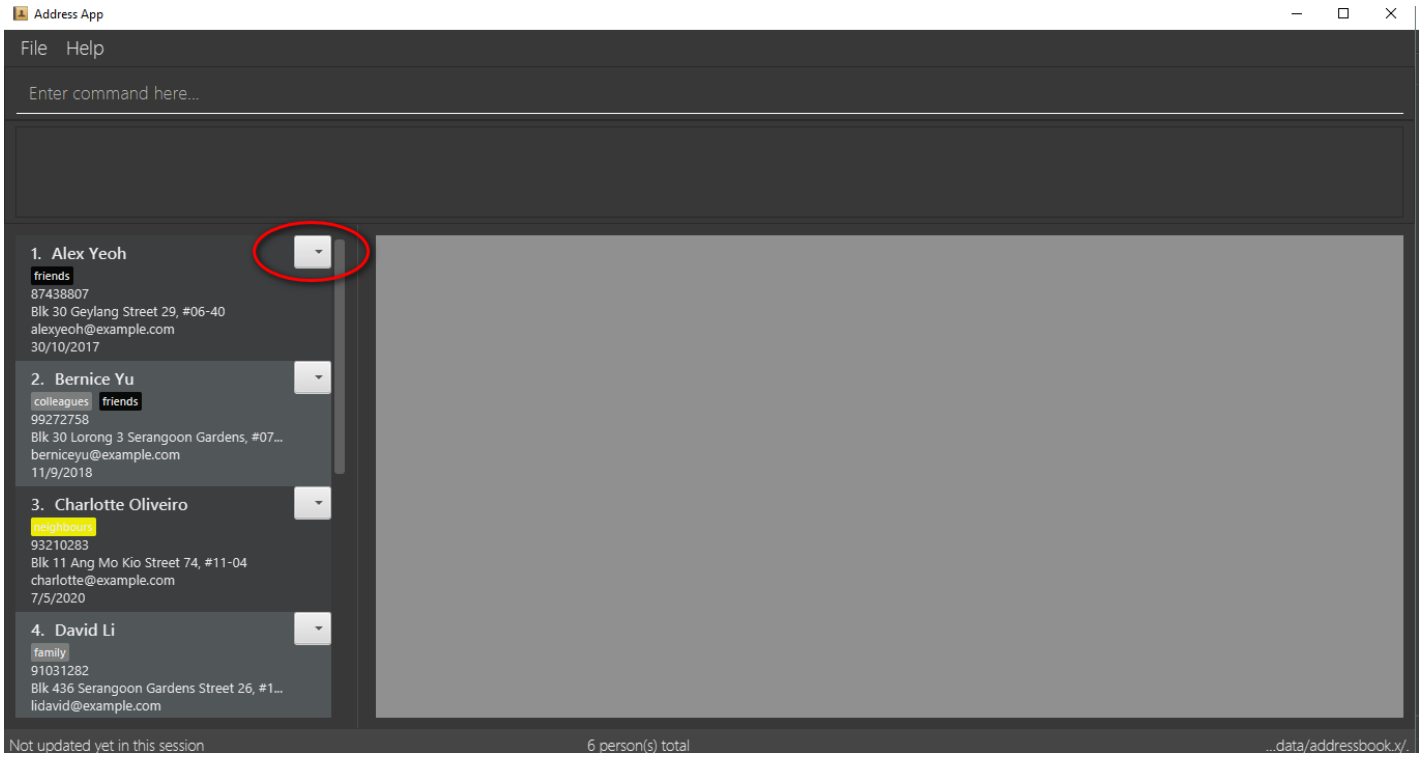
Enhancement Added: Adding birthday via GUI or CLI

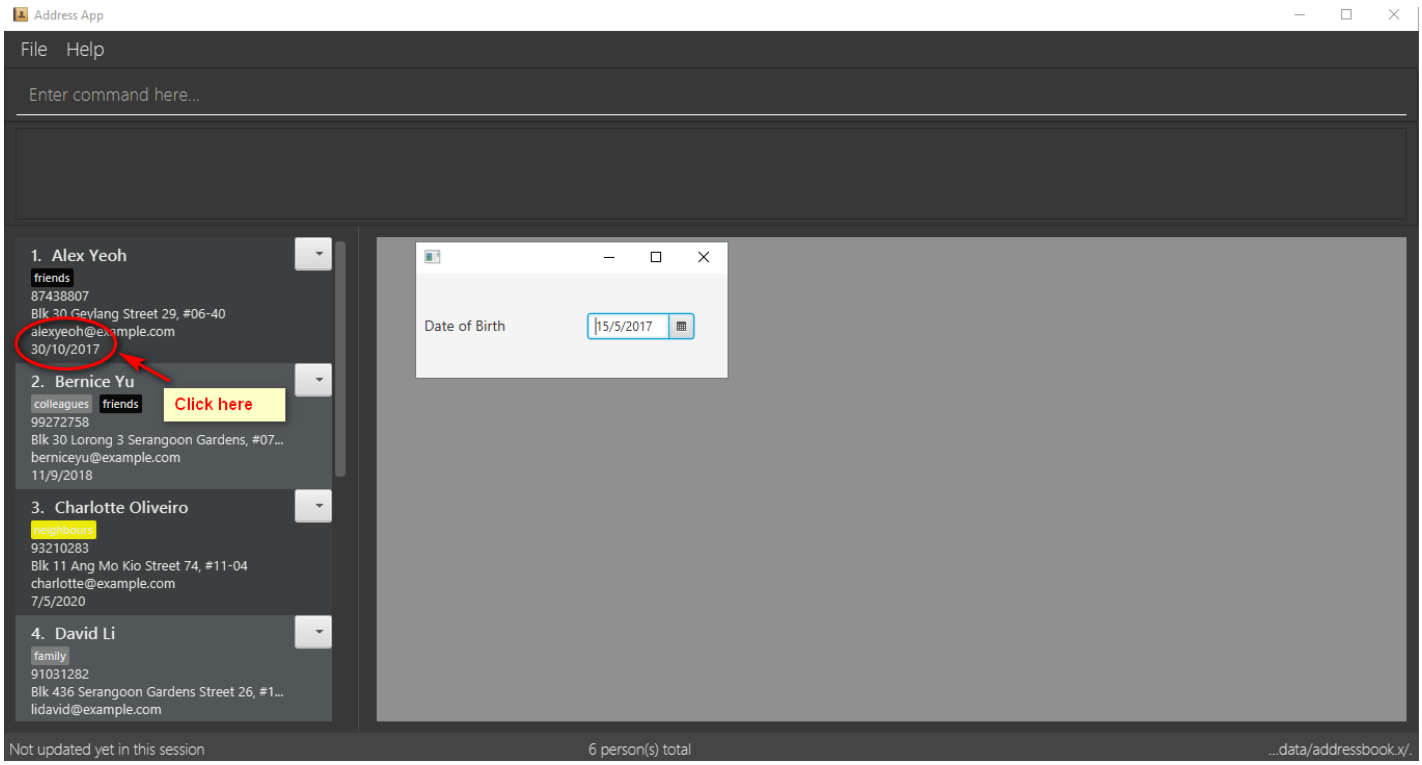
External behavior

Start of Extract [from: User Guide]

Adding a birthday for a person

Clicking on the menu next to each person's names followed by clicking "Set Birthday" allows a birthdate to be added for a person. After selecting the birthdate in the date picker, the birthday label of the person has to be clicked on for the changes to be shown.





End of Extract

Justification

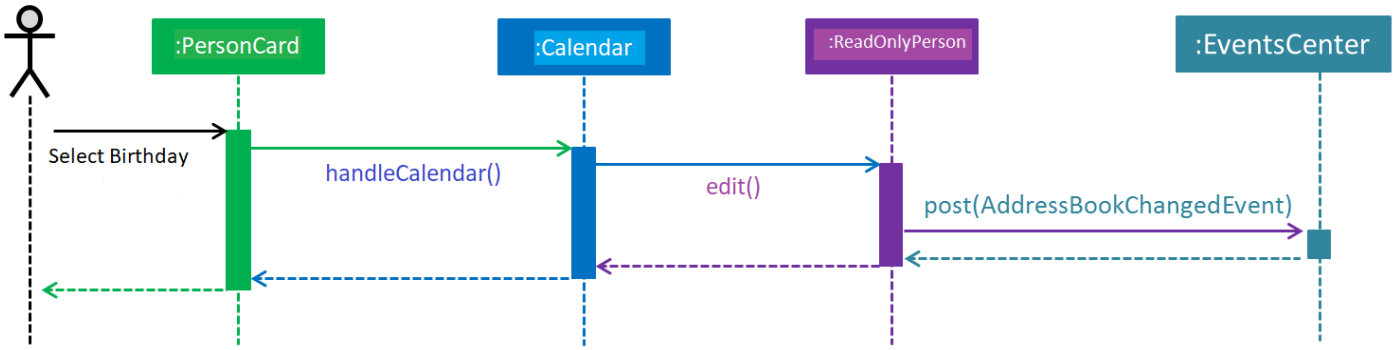
For easier convenience to shorten the add command input, the user can add a birthdate for a person after he has added him to the addressbook.

Implementation

Start of Extract [from: Developer Guide]

The add birthday mechanism

A person's birthday can be added/changed via the GUI. This ensures that the input for a birthdate always conform to a standard format. In this situation, since the Label has to change whenever a new date is picked from the date picker a bidirectional binding is used instead of the normal binding.



End of Extract

Other contributions

Last updated 2017-11-14 02:36:43 +08:00