Teo Meng Shin, Ryan - Project Portfolio

Project: AB&B

AB&B is a java-based address book application. The application allows the user to easily manage his/her contacts' information. Although there is a GUI, main interactions from the user are done via CLI. While there are features which benefit the generic user, the team has included features tailored specifically for administrative personnel.

The objective of this portfolio is to document notable contributions I have made to the application. A record of my contributions towards other similar projects can be found towards the end of the document.

Code contributed: [Functional code (https://github.com/CS2103AUG2017-T10-B3/main/blob/master/collated/main/AceCentury.md)] [Test code (https://github.com/CS2103AUG2017-T10-B3/main/blob/master/collated/test/AceCentury.md)]

Enhancement Added: Deletion by Name

External behavior

Start of Extract [from: User Guide]

Deleting a person by name : `deletebyname`

Deletes the specified person from the address book. Case insensitive.

Format: `deletebyname NAME`

> - Deletes the person with the specified `NAME`.
> - The name refers to the exact name of the person in the address book.
> - The name is case insensitive.

Examples:

- `deletebyname John Doe`
  Deletes John Doe in the address book, if the person exists.

If no person with a matching name can be found, it will update the displayed person list to provide suggestions on possible persons to delete.

If there is more than 1 person with the exact same name, it will update the displayed person list to show all the persons with the same name. You will then be prompted to use the `delete` command.

End of Extract

Justification

To allow users to easily delete a contact with their name instead of having to scroll through the displayed person list to obtain the corresponding index. It also provides some time saving benefits by integrating a `find` function with the deletion query in the event that the user cannot recall the exact name of the contact to delete.

Implementation

Start of Extract [from: Developer Guide]

DeleteByName Command

The `DeleteByNameCommand` extends the `UndoableCommand` class. It enables the deletion of a person from AB&B when given an input `Name` parsed by `DeleteByNameCommandParser`.

The class diagram of the command is shown below:

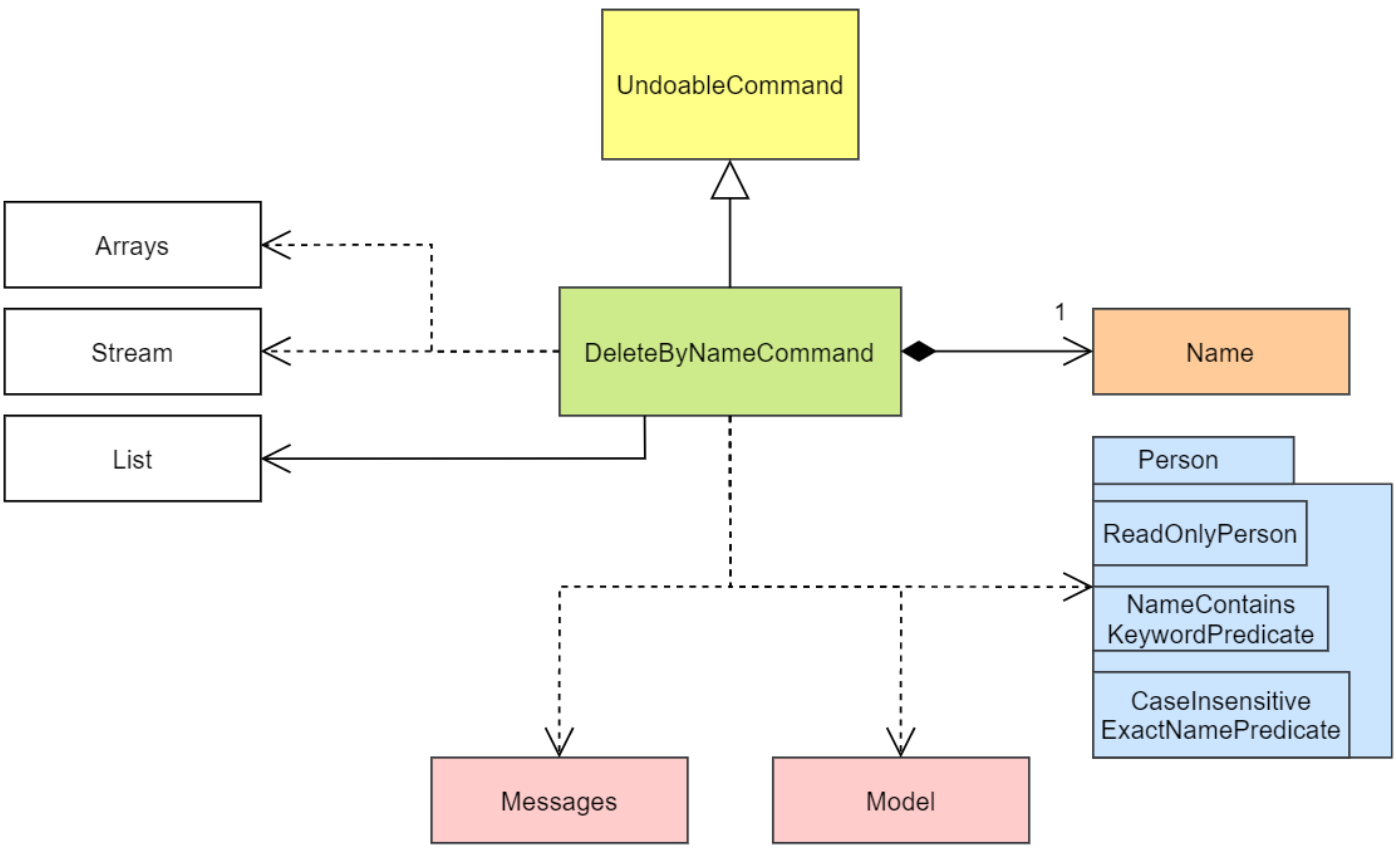

Figure 3.1A - Class Diagram of DeleteByNameCommand

In *Figure 3.1A* the `DeleteByNameCommand` class is highly dependent on the `Person` package as well as the `Model` of AB&B. This dependency allows it to carry out its delete operation. The `Model` of AB&B will be directly updated within the command.

The self calls of `DeleteByNameCommand` in the `executeUndoableCommand()` method are illustrated in the code fragment below:

```
@Override
public CommandResult executeUndoableCommand() throws CommandException {
    this.personList = model.getAddressBook().getPersonList();
    ReadOnlyPerson personToDelete = obtainPersonToDelete();

    if (personToDelete == null) { // No matching name found
        provideSuggestions();
    }
    //...deletion logic...
}
```

The sequence diagram of the Main Success Scenario of the command is shown below:

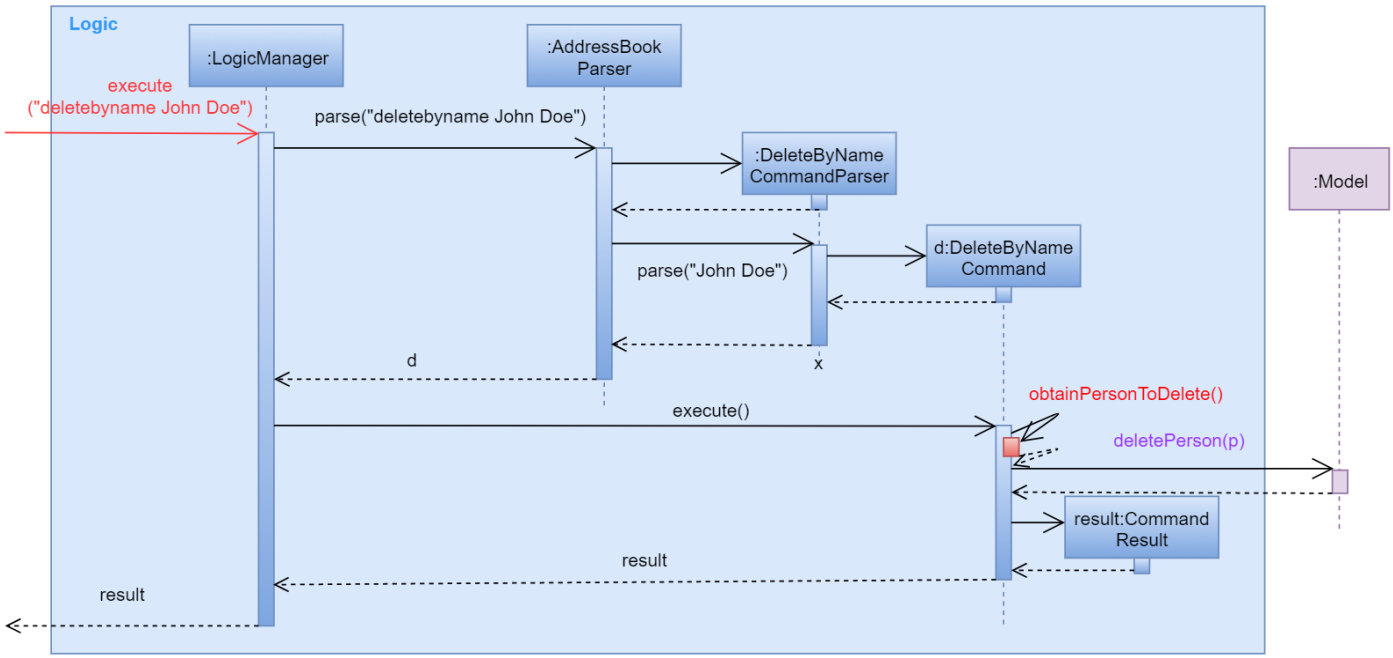

Figure 3.1B - Sequence Diagram of DeleteByNameCommand

The sequence of operations carried out in *Figure 3.1B* are detailed below:

1. The `execute("deletebyname John Doe")` command is called on the `LogicManager`.
2. `LogicManager` calls the `parse` method on `AddressBookParser`.

3. `AddressBookParser` parses the command word, `deletebyname` and calls `parse` on `DeleteByNameCommandParser` to parse the remaining argument, `"John Doe"`.
4. `DeleteByNameCommandParser` creates a new `DeleteByNameCommand`, `d`, and returns it all the way back to `LogicManager`.
5. `LogicManager` calls the `execute()` method on `d`, a `DeleteByNameCommand`.
6. `DeleteByNameCommand` calls itself to obtain `p`, the person to delete.
7. `DeleteByNameCommand` deletes the person, `p`, from the `Model`, and generates a new `CommandResult`, `result`.
8. The `result` is returned to the `LogicManager` which returns it back to the `UI`.

Advantage(s) versus `DeleteCommand`:

- Allows users to carry out delete operations regardless of the last shown list.

Disadvantage(s) versus `DeleteCommand`:

- Requires the exact name of the person to be deleted in order to perform a successful deletion.
- Cannot delete a person if there is another person with the exact name present in the AB&B.

In order to mitigate the disadvantages when compared to `DeleteCommand`, `DeleteByNameCommand` also suggests possible persons with similar names for deletion. The `Model` is also updated to display the list of suggested persons, similar to `FindCommand`.

`DeleteByNameCommand` also updates the `Model` to list all persons with matching names if there is more than one person with the exact same name as the person to be deleted. This is an enhancement over the traditional `FindCommand` as it will not list any other persons whose names match part of the query. It will then prompt users to utilise the `DeleteCommand`.

Design Considerations

Aspect: Implementation of `DeleteByNameCommand`

Current choice: Filter the list of persons present in AB&B and creating a helper `Predicate`, `CaseInsensitiveExactNamePredicate`.

Pros:

- Filtering from entire list of persons present in the AB&B facilitates a complete search.
- Creating a helper class `CaseInsensitiveExactNamePredicate` allows for better exception handling of `DeleteByNameCommand` to show a list of persons with matching names. It also improves abstraction, allowing it to be maintained and updated easily.

Cons:

- Filtering from entire list of persons present in AB&B can be time consuming.
- Creating an additional class `CaseInsensitiveExactNamePredicate` present within the `Person` package which is currently unused by any other function.

Aspect: Implementation of `DeleteByNameCommandParser`

Alternative 1 (current choice): Create a separate command word, `deletebyname`.

Pros:

- Not overloading the `delete` command word, providing clear distinctions for the user.

Cons:

- Creating an additional and lengthier command word for the user to enter.
- Creating an additional class within the `Parser` package.

Alternative 2: Overload `DeleteCommandParser`

Pros:

- Achieving different results with the same command word.
- Removing the need for extra classes within the `Parser` package.

Cons:

- Parsing logic for `DeleteCommandParser` becomes more complicated.

End of Extract

Enhancement Added: Export to CSV

External behavior

Start of Extract [from: User Guide]

Exporting the data : `export`

Saves the current data in the address book into a CSV (Comma-Separated Values) file. The file will be saved in the same location where you stored the application. The file will be named `AddressBookData.csv`.

Format: `export`

End of Extract

Justification

To allow users to easily make use of the data stored in AB&B with other applications. An example would be to upload the CSV file into an email client as a quick and simple way to mass update email addresses.

Implementation

Start of Extract [from: Developer Guide]

Export

The `ExportCommand` extends the `Command` class. It allows users to export the current AB&B data into a CSV file.

The class diagram of the command is shown below:

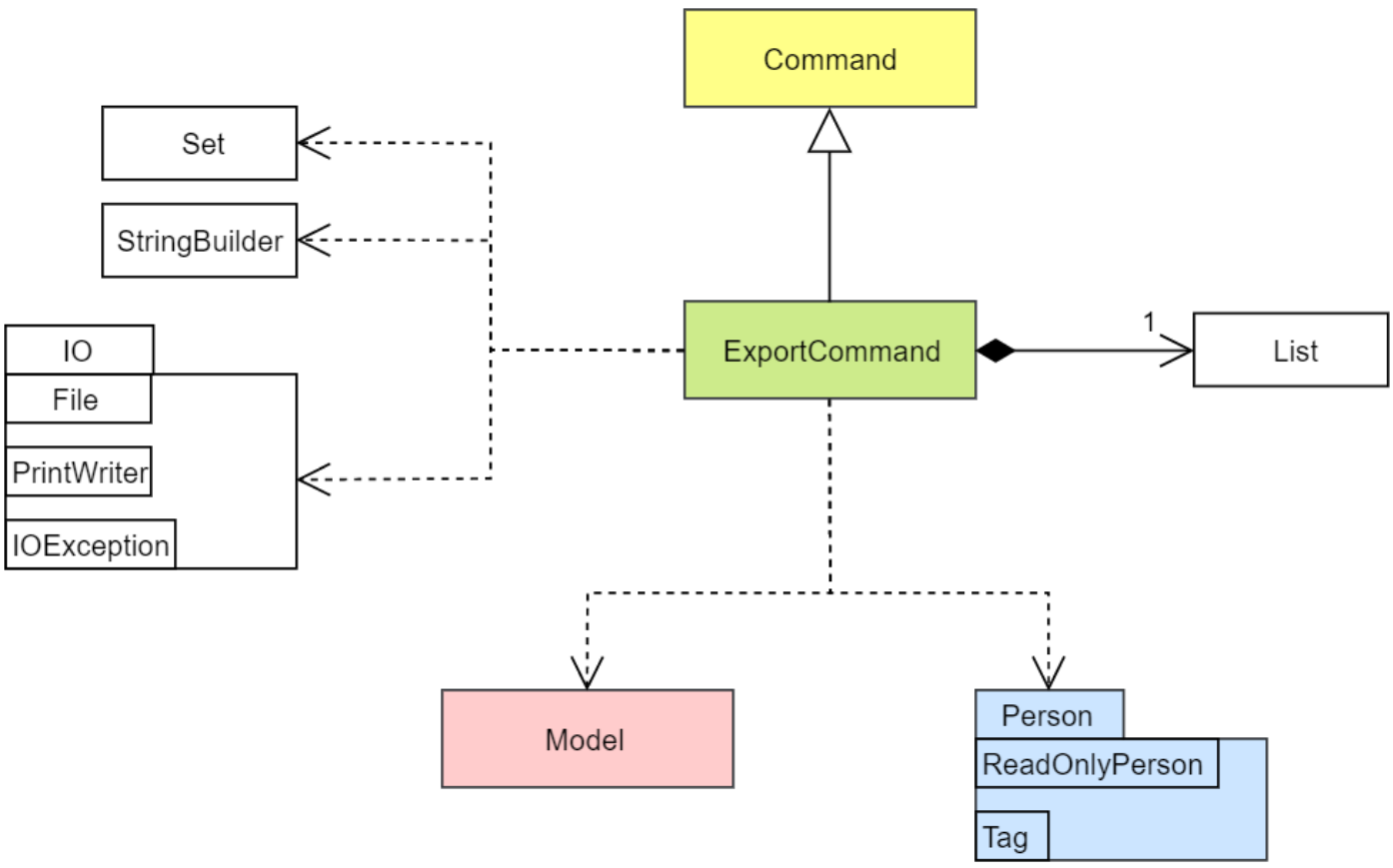

Figure 3.6A - Class Diagram of ExportCommand

From *Figure 3.6A*, the ExportCommand depends on `java.io` to carry out the file IO operations to create the CSV file. It also depends on the `Model` class and `Person` package in order to extract the required information to export.

The self calls of `ExportCommand` in the `execute()` method are illustrated in the code fragment below:

```
@Override
    public CommandResult execute() throws CommandException {
        this.currentData = model.getAddressBook().getPersonList();

        if (currentData.isEmpty()) {
            throw new CommandException(MESSAGE_EMPTY_ADDRESS_BOOK);
        }

        if (fileExists()) {
            deleteFile();
        }

        createFile();
        writeData();
        return new CommandResult(MESSAGE_EXPORT_SUCCESS);
    }
```

The sequence diagram of the command is shown below:

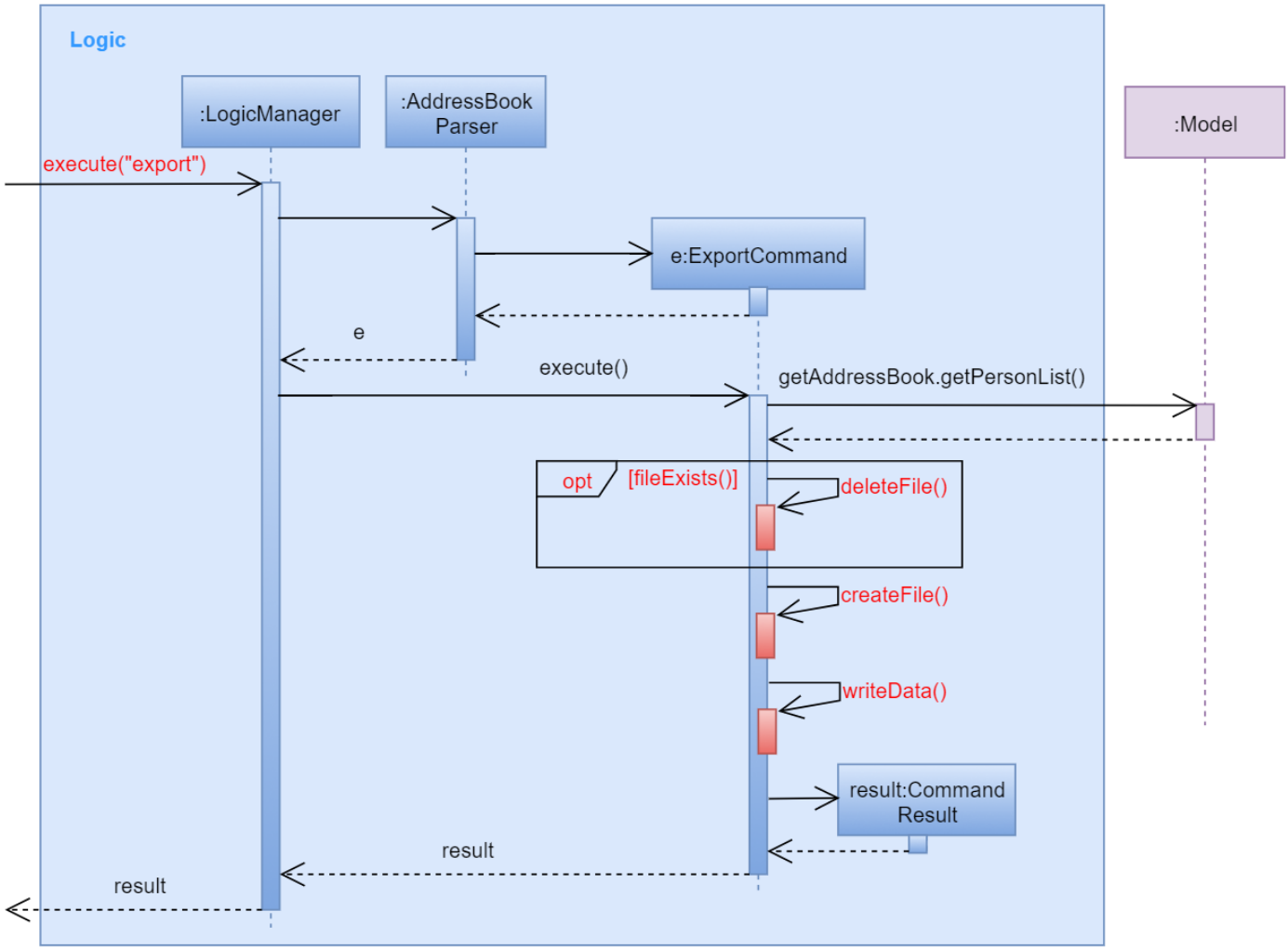

Figure 3.6B - Sequence Diagram of ExportCommand

The sequence of operations carried out in *Figure 3.6B* are detailed below:

1. The `execute("export")` command is called on the `LogicManager`.
2. `LogicManager` calls the `parse` method on `AddressBookParser`.
3. The `parse` method creates a new `ExportCommand` which returns an `ExportCommand`, `e`, all the way back to `LogicManager`.
4. `LogicManager` calls the `execute()` method on `e`, an `ExportCommand`.
5. `ExportCommand` obtains the `PersonList` from the `Model`.
6. `ExportCommand` checks if an exported file exists. If the file exists, it will delete the file.
7. `ExportCommand` generates the exported file through a series of self calls and generates a new `CommandResult`, `result`.
8. The `result` is returned to the `LogicManager` which returns it back to the `UI`.

Design Considerations

Aspect: Construction of Strings

Alternative 1 (current choice): Use `StringBuilder`

Pros:

- Can easily construct the `String` due to the mutability of `StringBuilder`.

Cons:

- Increasing dependency on external classes, specifically the `StringBuilder` class.

Alternative 2: Use `String` only.

Pros:

- Reducing dependency on additional classes.

Cons:

- Increasing number of re-assignments of resultant `String` is required as it is not mutable.

Aspect: Generating Person Data

Alternative 1 (current choice): "Hard-code" the data to obtain from a `ReadOnlyPerson`.

Pros:

- Accelerates code execution.

Cons:

- Increasing coupling with `ReadOnlyPerson`.
- Reducing code adaptability should new fields be added to `ReadOnlyPerson`.

Alternative 2: Generate person data according to the fields present in `ReadOnlyPerson`.

Pros:

- Increasing code adaptability should new fields be added to `ReadOnlyPerson`.

Cons:

- Decelerates code execution.
- Increasing difficulty and time required to properly implement.

End of Extract

Enhancement Proposed: Ability to suggest frequently accessed contacts

Users may need to frequently look up a particular contact for numerous reasons. By having such a feature, it would save them time and effort if AB&B were able to suggest or prioritise frequently accessed contacts in the displayed person list.

Other contributions

- Made the `command word` case-insensitive. (Pull request #42 (https://github.com/CS2103AUG2017-T10-B3/main/pull/42))
- Offered `DeleteByNameCommand` for re-use (Re-Use Offer (https://github.com/nus-cs2103-AY1718S1/forum/issues/121))

Contributions to other teams' projects

- `Export` function re-used by `Tourist Book`. (Implementation (https://github.com/CS2103AUG2017-F09-B2/main/pull/78)) (Re-Use Offer (https://github.com/nus-cs2103-AY1718S1/forum/issues/158))
- Bug reports for `ContactPro` (Issues #62 (https://github.com/CS2103AUG2017-T12-B1/main/issues/62), #63 (https://github.com/CS2103AUG2017-T12-B1/main/issues/63), #71 (https://github.com/CS2103AUG2017-T12-B1/main/issues/71))
- Bug report for `AcquaiNote` (Issue #80 (https://github.com/CS2103AUG2017-T10-B1/main/issues/80))

Last updated 2017-11-12 23:31:04 +08:00