

Jonas Ong - Project Portfolio

PROJECT: Tagline

Purpose of Personal Project Portfolio

This portfolio is to document my contributions made towards the project done in CS2103T. It highlights several of my code contributions as well as documentation additions to show the main feature I have been working on as part of the project.

About the project: Tagline

Tasked with enhancing the capabilities of a basic command line interface desktop addressbook application, my team of five chose to extend the functionality to include note-taking capabilities. Combining the existing contact infrastructure with tags and note support, we aim to make a more people-centric note taking platform we call Tagline. This application aims to help students more easily organize and find their notes based on the people the notes are connected to.

This is what our project looks like

My role for this project is to implement the **Group** feature to allow for organizing **Contact`s together into `Groups** which can be used to manage notes as a whole. The following sections together with documentation added to the user and developer guides depict this enhancement in greater detail.

Summary of contributions

In this project, I my main feature was to develop a way to organize contact together into Groups. This allows users to organize contacts in a more natural way based on social circles of relationships

rather than as an alphabetically ordered list. It would also lay the groundwork for other features such as group tagging of notes.

¥ Major enhancement: I added a range of features providing the ability to group contacts together for better organization

! What it does: with `group create` a group of contacts can be created, contacts can be added or removed using `group add` and `group remove` commands. Groups can be searched for and displayed using `group list` and `group find`.

! Justification: social circles exist in more than just a one to one relationship and can belong in a group with a shared feature be it an orientation group of friends who all know each other or a project group with common purpose. Having the capability to quickly reference other contacts in the same group would make the user experience much better compared to scrolling through an alphabetically ordered list in an addressbook.

! Highlights: this enhancements supplements the existing structure of the addressbook.

¥ Code contributed: The following links highlight samples of my code [[Functional code](#)] [[Test code](#)] _

¥ Other contributions:

! Documentation:

" Updated the developer guide to better reflect `Model` and `Storage` components after they have been changed to accommodate other features: [NoteModel](#), [StorageComponent](#)

! Community:

" Contributed to forum discussions to clarify potential tripups (examples: [131](#)

" Reviewed and offered suggestions for other teams in class (examples: [SecureIT#22](#)

Contributions to the User Guide

The following sample highlights some of my additions to Tagline User Guide. This is primarily for `group` features I had developed.

The following excerpt for `group add` is chosen to represent other group command which are written in a similar style but omitted due repetition. The sample aims to demonstrate my approach to writing documentation to help end users better understand the features of the product.

Sample: `group add` command

Add member to a group: `add`

Adds members to a group.

Format:

```
group add GROUP_NAME [--i CONTACT_ID]+
```

Example:

¥ Over time our 'ao3' group has grown with new members joining. We need to update our group to record the recent addition. To add a member to our 'ao3' group we can use the command `group create ao3 --i 90040`

Figure 1. Entering the command

¥ This adds a contact with ID '90040' to the group with name 'ao3' The display will show the group and the contact that has been added.

Figure 2. Command executed Group created

Contributions to the Developer Guide

The following sample highlights some of my additions to Tagline Developer Guide. This is primarily for **group** features I had developed.

The following excerpt demonstrate my approach to writing documentation to showcase how I write technical documentation to explain how my developed feature functions to aid future developers better understand how the feature functions and the design choices considered and made.

Group Contacts feature

Description

Groups allows users to better organize contacts into relevant social circles (represented as **Group**) to better express relationships much like how they exists as in real life. This feature would provide the foundation for further more advanced features such as tagging of notes with group tags.

The user can work with groups by using the commands as detailed in the **group** section.

Commands currently available:

¥ **group create** - creates a new group

¥ **group remove** - removes a contact from a group

- ¥ `group add` - adds an existing contact to the group
- ¥ `group list` - list all available groups
- ¥ `group find` - searches for group by exact name and displays contacts in the group
- ¥ `group delete` - disbands a group (contacts in group are not deleted)

Implementation

The grouping feature is facilitated by `GroupBook`, an additional Model component in addition to the current `AddressBook`. It extends the functionality of `AddressBook` by providing a way to group contacts together into unique `Group` classes identified by their `GroupName`. This allows users to form more natural associations of contacts such as "BTS-members". Identifying which contacts are group members of a `Group` is done by storing a record of their `ContactId` in the `Group`. Additionally, `GroupManager` extends Tagline with the following operations to support commands dealing with groups:

- ¥ `GroupManager#getGroupBook()` - Retrieves a view only version of the groups for storing data after app quits.
- ¥ `GroupManager#deleteGroup()` - Deletes a group from the list of groups currently available.
- ¥ `GroupManager#addGroup()` - Adds a group to the list of groups currently available.
- ¥ `GroupManager#setGroup()` - Replaces a group in the list of groups with another group.
- ¥ `GroupManager#getFilteredGroupList()` - Returns a view only list of groups containing a subset of available Groups.
- ¥ `GroupManager#updateFilteredGroupList()` - Specifies which groups will be retrieved by `GroupManager#getFilteredGroupList()`.

The above operations are exposed in the `Model` interface by their respective method names.

- ¥ `GroupCommand#findOneGroup()` - Retrieves one Group with name matching the exact provided String.
- ¥ `GroupCommand#verifyMemberIdWithModel()` - Compares members currently in a group with contacts in `AddressBook` and returns only those found in `AddressBook`.
- ¥ `GroupCommand#setDifference()` - Used to get contactids specified which do not exist in `AddressBook`.

These above are static utility functions which form the underlying structure of how a `GroupCommand` works.

Given below is an example usage scenario on how a typical lifecycle of a `Group` behaves at each step. With emphasis on showing the effects of `DeleteCommand` as an example of a command from `ContactCommand` would interact with `GroupCommand` and `GroupModel` state.

Step 1. The user initially has several contacts in `AddressBook`.

Figure 3. Simplified state of relevant Model components initially

The `AddressBook` model state contains all the `Contact` class that exists in the App. Since no `Group` has been created yet, `GroupBook` model state is currently empty. All of the contacts found in `AddressBook` are displayed on the `UI` by default.

Step 2. Wishing to better organize her contacts into groups, the user executes `group create BTS` calling `CreateGroupCommand`. to create a new `Group` instance with no members.

Figure 4. State after Group "BTS" is created

The `GroupBook` model state now contains a `Group` instance for "BTS" with no members recorded as `memberIds`. Any command regarding `Group` would prompt the `UI` to display the contacts in the group. A group with no members would cause the `UI` to be empty. As there are no contacts in the group. While a group with members in it would cause `UI` to display all the contacts belonging to that group.

Step 3. The user then executes `group add BTS --i 00001 --i 00002 --i 0013 --i 0004` calling the `addMembersToGroupCommand` to add several contacts to the group. Only the String representation of the

`ContactId` will be stored in the `Group`.

Figure 5. State after four contacts are added into Group "BTS"

`Group` "BTS" now has members in it and the `UI` would display all the contacts found in the group.

Step 4. The user realizes she has made a mistake adding a wrong contact and in a fit of rage chooses to delete the contact instead of merely removing the contact from the `Group`. Executing `contact delete 00013` which then deletes the `Contact` with `contactId` of 00013. However, this does not remove the `contactId` from the `memberId` attribute in the `Group` the contact was in. This step does not involve `GroupModel` in any way.

Figure 6. State after contact with `contactId = 00013` is deleted, `UI` for groups is not active at this point

Deleting a `Contact` would cause it to be removed from `AddressBook` model state and the `Contact` no longer exists. Due to the `contact` command, the active `UI` shifts to displaying a list of contacts (not illustrated here for simplicity) and the groups as shown in the image are actually not visible to the user. However behind the scenes, while the `UI` no longer has contact of 00013, it is still recorded as a member in `GroupBook` model state. The updating of `GroupBook` model state is deferred.

Step 5. The user then executes `group add BTS --i 00003` to add the correct contact as a member on the `Group` and view the `Contact` profiles. This calls `AddMemberToGroupCommand` which then updates the `Group` ensuring that all `memberIds` correspond to an existing `ContactId` found in `AddressBook`. The contacts of the group are also displayed to the user.

Figure 7. State after user views contacts of Group "BTS", UI displaying the group of contacts is now visible

Here, the `GroupBook` model state is updated and `memberId` of 00013 from the previous step is removed while `Contact` with `contactId` of 00003 is added into the `Group`. This change is also reflected in the `UI` which changes back to group display now that a group command is issued. Now all is as it should be in `Group` "BTS".

The following sequence diagram summarizes what happens when a user executes a `FindGroupCommand` which updates the `Group` similar to how `AddMemberToGroupCommand` does in the above example:

Figure 8. Sequence diagram of executing `FindGroupCommand` to view contacts in a `Group`

Design Considerations

Aspect: How groups stores contacts

¥ Alternative 1: Stores `ContactId` class in a `Collection` in `Group`

- ! Pros: Easy to get `ContactId` from `Group` to retrieve `Contact` classes from `Addressbook`.
- ! Cons: Increases coupling to implementation of `Contact`. Storage and retrieval after reloading the app would also cause new instances of `ContactId` to be created when loading `Group` or would require more complicated loading of `Group` from storage having to happen after `AddressBook` is loaded and having to reference `Contact` classes to ensure the same `ContactId` class is referenced by both `Contact` and `Group` it is in.

¥ Alternative 2 (current choice): Stores `Collection` of Strings which are able to uniquely identify `Contact`.

- ! Pros: Group classes are less coupled to implementation of `Contact`. Simpler to load `Group` classes from storage. due to not needing to check and obtain a reference to `ContactId`. User input is also parsed as Strings.
- ! Cons: Deciding when to check if members are still part of a `Group` since it need not be done at loading time. While it is more flexible, can be a potential source of confusion as it may be possible to forget to update the members in `Group`.