Yang Zi Yun - Project Portfolio

PROJECT: Duke Email Manager

Product Overview

My team and I developed an Email Manager as a solution for the overwhelming emails received daily in our mailbox.

We were tasked with enhancing a basic Command Line Interface (CLI) task manager for our Software Engineering project. We chose to morph it into an email manager while enhancing the task manager. We also provide ways to link the management between tasks and emails.

The figure below shows interface of our app:

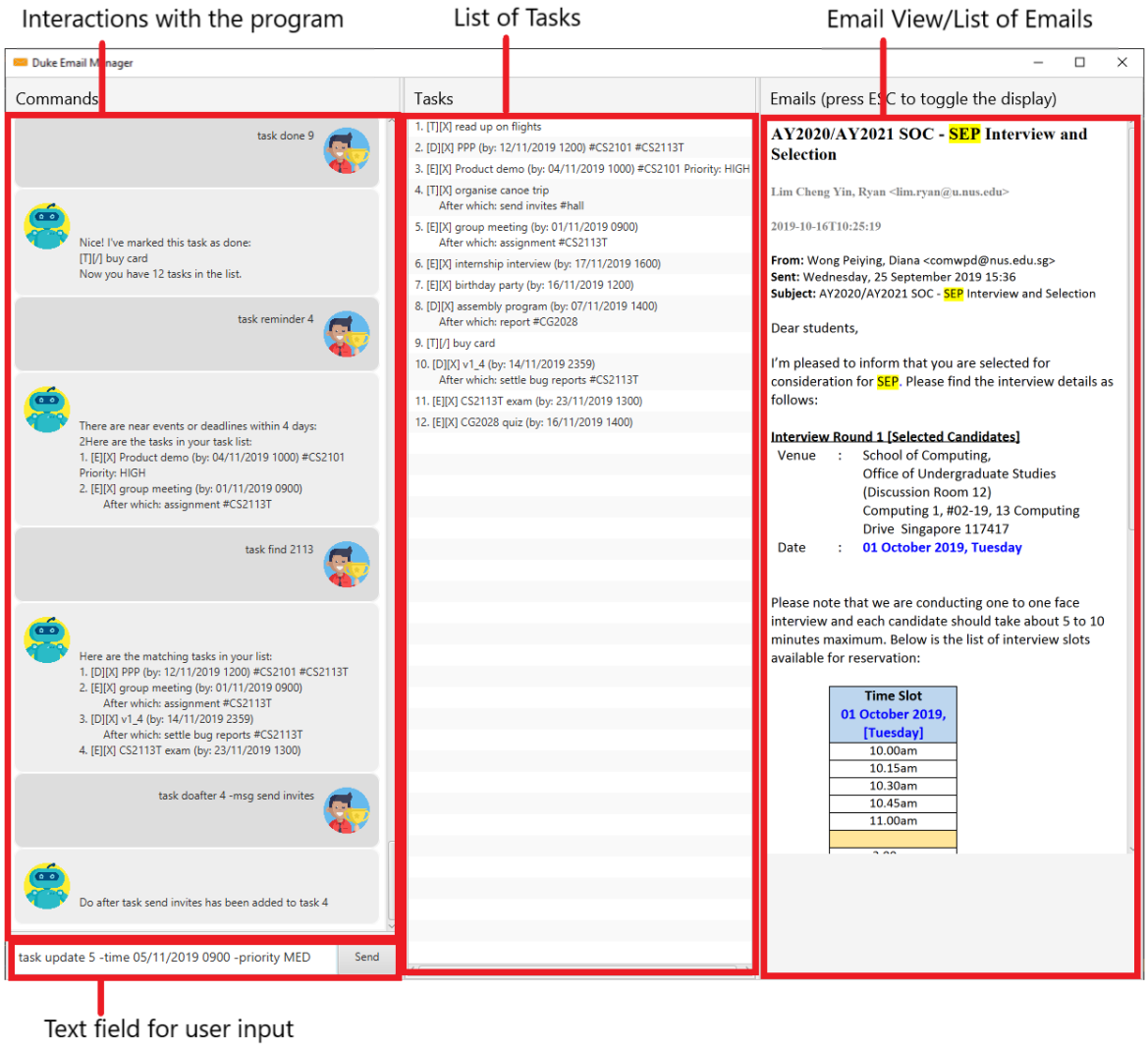

Figure 1. GUI interface of Email Manager

Duke Email Manager is an email and task manager desktop app, specifically designed for NUS School of Computing Students to manage their emails and busy schedules. As a text-based application, it is optimized for those who prefer typing and working with CLI. Email Manager also

has a developed Graphical User Interface (GUI) that allows users to view email and task details in an appealing, well-organized format.

My role was mainly to design and write the codes for email managing including email showing, email tagging and email filtering. The following sections illustrate these enhancements in more detail, as well as the relevant sections I have added to the user and developer guides in relation to these enhancements.

Summary of contributions

This section shows a summary of my coding, documentation, and other helpful contributions to the team project.

Code contributed

Summary of code contribution

Enhancement added

- Email Tagging
  - What it does: Students can tag their emails with the command `email update <index> -tag <TagName1> -tag <TagName2>`
  - Justification: By tagging the emails, students can have a clearer overview of their list of emails. Hence, they can attend important emails in the fastest time. This also minimizes the chances of missing out crucial information being informed by email.
  - Highlights: Multiple tags can be added to an email. Duplicated tags can be detected and not added to the email again.
  - Code contributed: #82
- Email Filtering by Tag(s)
  - What it does: Students can filter the emails by the tags added with the command `email list -tag <TagName1> -tag <TagName2>`
  - Justification: This allows the students to view the list of emails being tagged with particular tag(s), helping them to filter out relevant emails to keep their email organised.
  - Highlights: Maximum number of input tags allowed is two, this will shows emails that are tagged with both tags.
  - Code contributed: #97 #105
- Email Showing
  - What it does: The command `email show <index>` will display the email in html format.
  - Justification: Student can view the content of email in the application without have to find the email in other places. This helps the student to access all the information in the email all in one place.
  - Highlights: Pressing `ESC` key can toggle the display between email list view and email content

view.

    - Code contributed: #71

- Switching mode between email and task
    - What it does: Students can easily switch the mode between `email` and `task` with the command `flip`. In the user input text field, a prefix either `email` or `task` will be displayed with respect to the current mode.
    - Justification: Students will need to frequently change their input when they try to handle their tasks and emails, this feature allows them to navigate between both modes easily.
    - Highlights: The prefix in the user input text field is design to reduce the amount of typing required from the students. The prefix is non-deletable.
    - Code contributed: #54

Other contributions

- Project management:
    - Managed releases versions `v1.2.1` on GitHub (#v1.2.1)
- Documentation:
    - Update User Guide and Developer Guide
- Community:
    - PRs reviewed (with non-trivial review comments)
    - Reported bugs and offered suggestions for other teams in the class ()
- Tools:
    - Integrated continuous integration (Travis) to the team repo (#36) (#37) (#38)
    - Integrated coverage report (Coveralls) to the team repo (#107 #102)
- GUI enhancement:
    - Implemented a different background colour for UserDialogBox and DukeDialogBox for clearer layout and view purpose (#114)
    - Resize window to fit screen (#71)
    - Display of email using WebView and toggling email list view and email content view by pressing `ESC` key (#71)
- Other functionalities or feature:
    - Task detection of anomalies (#32)
    - Basic email class implementation (#48 #49 #50)
    - Wrote additional tests for existing features to increase instruction coverage from 19% to 29%, and increase branch coverage from 15% to 20% (#114)
    - Implemented logger (#181)
    - Added key binding functionality to create keyboard shortcut (#78 #70)

Contributions to the User Guide

We had to update the User Guide with instructions for the enhancements that we had added. The following is an excerpt from our Email Manager User Guide, showing additions that I have made for the email managing features.

> *Given below are sections I contributed to the User Guide. They showcase my ability to write documentation targeting end-users.*

Change Mode: `flip`

Format: `flip`

Flips/toggles between email mode and task mode. The prefix of the command in the text box will also be changed.

> **NOTE** In task mode, the text box will display `task` as a prefix. In email mode, the text box will display `email` as a prefix.

Listing all emails: `list`

Format: `list`

Gives a complete list of emails.

Showing an email: `show`

Format: `show INDEX_NUMBER`

Show the email content of the email at the index number in the email list.

Example:

`show 3`: shows content of the 3rd email in the email list.

> **TIP** You can press `Esc` key on your keyboard any time to switch display between the list and content view of emails.

Listing all keywords: `listKeyword`

Format: `listKeyword`

Gives a list of all keywords with the relevant expressions.

Tagging an email: `update`

Format: `update ITEM_NUMBER -tag TAG1 [-tag TAG2]`

Tags the specified item with the tag(s) minimum number of tags is 1. Tags without duplication will

be added.

Examples:

`update 1 -tag CS2113T`

`update 2 -tag Tutorial -tag Spam`

Listing all tags: `listTag`

Format: `listTag`

Gives a list of all existing tags in the list of emails.

Filtering email by tags: `list`

Format: `list [-tag TAG1] [-tag TAG2]`

Gives a list of emails with the tags. Minimum number of tags is 1, and the maximum number of tags is 2.

Examples:

`list -tag CS2113T -tag Tutorial`

`list -tag Spam`

Deleting an email: `delete`

Format: `delete ITEM_NUMBER`

Deletes the item specified.

Examples:

`delete 1`: deletes the first email in the email list.

NOTE	If you enter `show ITEM_NUMBER`, then followed with `delete ITEM_NUMBER`, the content of email at `ITEM_NUMBER` will still remains displayed although the email has been deleted.

Clear email list: `clear`

Format: `clear`

This command deletes all emails in the list.

WARNING	Once executed, you will not be able to undo this command.

TIP	After clearing all the emails, you can enter `fetch` to retrieve latest emails from server.

Contributions to the Developer Guide

Given below are sections I contributed to the Developer Guide. They showcase my ability to write technical documentation and the technical depth of my contributions to the project.

Email Management

Email Tagging

Email Manager allows user to tag emails by tags.

Current Implementation

- Format: `email update ITEM_NUMBER [-tag TAG1] [-tag TAG2]`
- Note: Tags the specified item with the tag(s) minimum number of tags is 1.
- Eg: `email update 2 -tag Fun -tag Project` : tags 2nd email in the list with the tags "Fun" and "Project".

Following is the activity diagram when the command is executed:

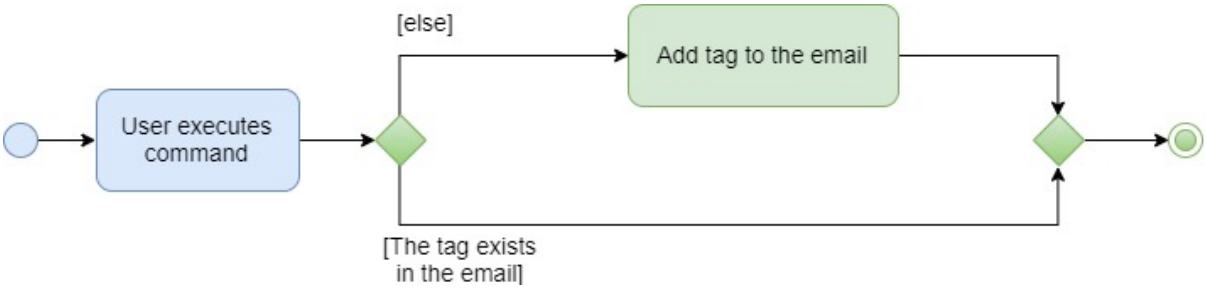

Figure 2. Activity diagram for email tagging

The following sequence diagram below will explain how the `email update` command works in detail:

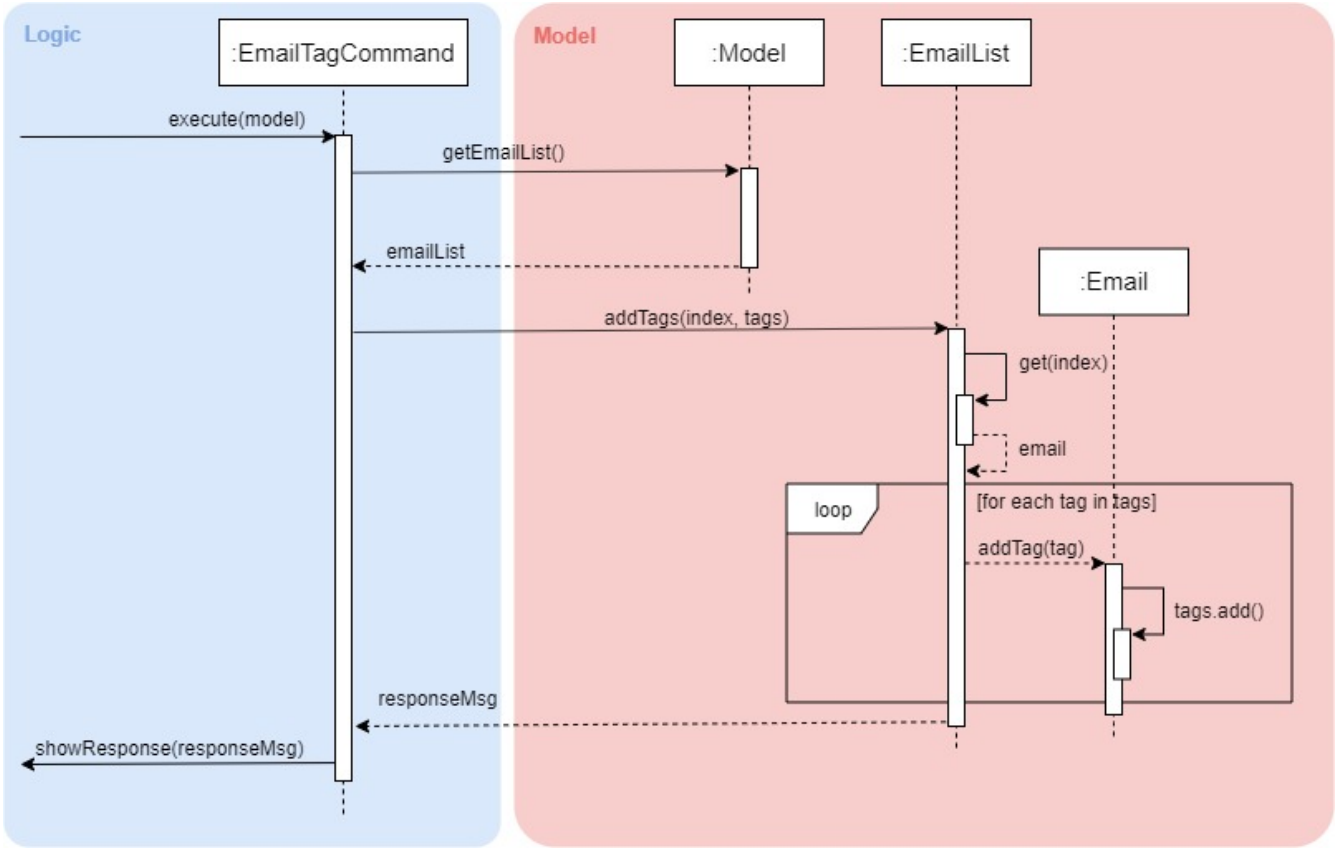

Figure 3. Sequence diagram for email tagging

An example usage of the command is as follows:

Step 1: The user launches the application. The user inputs `email update 2 -tag Fun -tag Project`

Step 2: `UI` component captures the input and passes to `Logic` component to parse the input. Section below explains how `Logic` component parse the input.

- `CommandParseHelper` takes in the `input`, parses and extracts tags information and stores it inside ArrayList<Option> `optionList`, then passes the `input` and `optionList` to `EmailCommandParseHelper`.
  - `input` here is `email update 2`
  - `optionList` here is `[tag=Fun, tag=Project]`
- `EmailCommandParseHelper` parses the `index` of email and extract tags information `optionList` and stores it in ArrayList<String> `tags`.
  - `index` here is `2`
  - `tags` here is `[Fun, Project]`
- `EmailCommandParseHelper` creates a new `EmailTagCommand` by passing in `index` and `tags`, then return the `EmailTagCommand` to `CommandParseHelper` and then to `UI`

Step 3 : `EmailTagCommand#execute(model)` is called by `UI`.

Step 4: `EmailTagCommand` calls `Model#getEmailList()`, then `emailList` is returned by `Model`.

Step 5: `EmailTagCommand` calls `EmailList#addTags(index, tags)`

- `EmailList` calls `get(index)` to get the email of the index number in the emailList.

    - Gets the 2nd email in the emailList.
- For each `tag` in `tags`, `EmailList` calls `Email#addTag(tag)`. `Email` calls `tags.add()` to add the tag to the email.
    - Each `tag` here is `Fun` and `Project`.

Step 6: `EmailList` returns a String `responseMsg` to `EmailTagCommand`.

- `responseMsg` here is:
  "Tags added: [Project, Fun] to email: <title of email at index 2>"

Design Considerations

Aspect: Handling updating of tags

- Alternative 1 (current choice): if at least one tag is entered as part of the command, it will overwrite all current tags of the task being modified.
    - Pros: makes it consistent with other parameters of the update command and gives users an option to replace/remove tags
    - Cons: if there are many tags, and the user only wants to add on an extra tag, the user will need to retype all existing tags into the command.
- Alternative 2: if a tag is entered as part of the command, it will amend on top of existing tags.
    - Pros: this will save users time if they only want to add on tags
    - Cons: no option to remove tags
- Alternative 3 (proposed): there will be an option to add tags and an option to remove tags.
    - Pros: gives users highest amount of flexibility and control over the tags they want to keep.
    - Cons: added complexity in commands

Email Filtering by Tag(s)

Email Manager allows user to filter emails by tag(s).

Current Implementation

- Format: `list [-tag TAG1] [-tag TAG2]`
- Note: Gives a list of emails with the tags. Minimum number of tags is 1, and the maximum is 2.
- Eg: `email list -tag Fun -tag Project`

Following is the activity diagram when the command is executed:

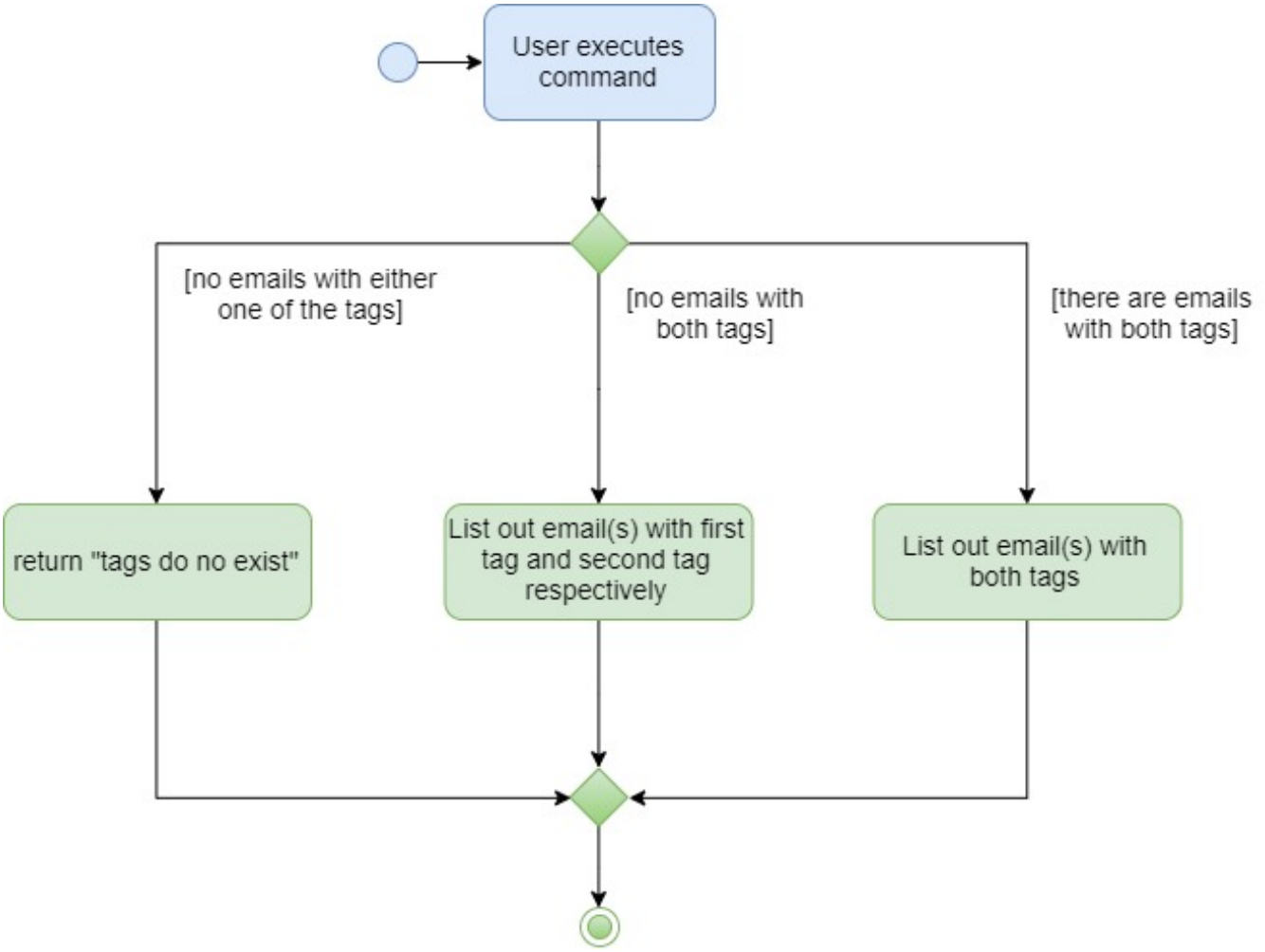

Figure 4. Activity diagram for email filtering by tags

The following sequence diagram below will explain how the `email update` command works in detail:

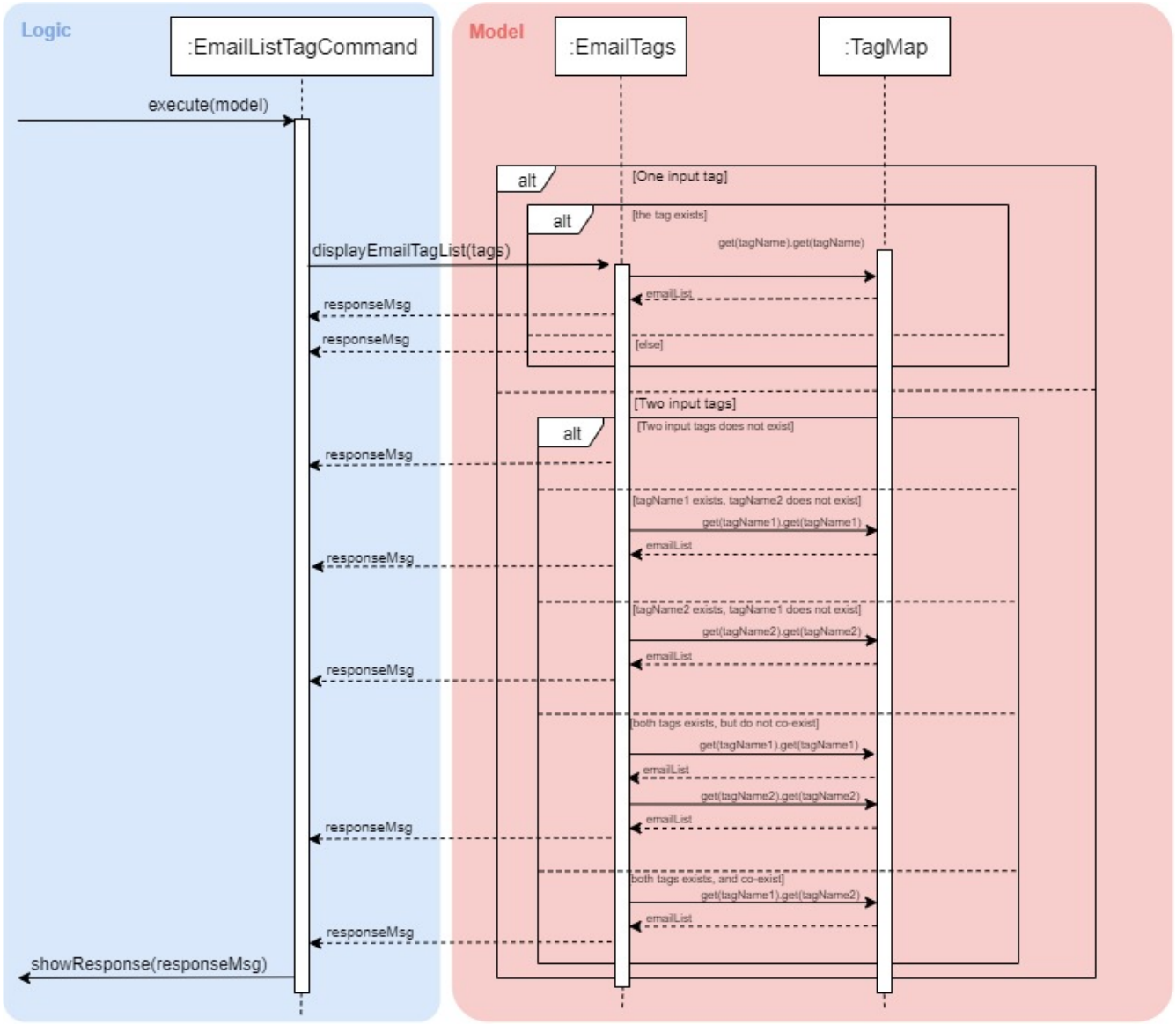

Figure 5. Sequence diagram for email filtering by tags

An example usage of the command is as follows:

Step 1 : The user launches the application. The user wishes to tag the 2nd email in the list with "Fun" and "Project" (Implementation of part is explained in Section 5.3.2). After tagging the email, the user wishes to view the list of emails with these tags, hence the user inputs `email list -tag Fun -tag Project`.

Step 2 : `UI` component captures the input and passes to `Logic` component to parse the input. Section below explains how `Logic` component parse the input.

- `CommandParseHelper` takes in the `input`, parses and extracts tags information and stores it inside ArrayList<Option> `optionList`, then passes the `input` and `optionList` to `EmailCommandParseHelper`.
  - `input` here is `email list`
  - `optionList` here is `[tag=Fun, tag=Project]`
- `EmailCommandParseHelper` parses the `input` and extract tags information `optionList` and stores it in ArrayList<String> `tags`.
  - `tags` here is `[Fun, Project]`

- `EmailCommandParseHelper` creates a new `EmailTagListCommand` by passing in `tags`, then return the `EmailTagListCommand` to `CommandParseHelper` and then to `UI`

Step 3 : `EmailTagListCommand#execute(model)` is called by `UI`.

Step 4: `EmailTagListCommand` calls `EmailTags#displayEmailTagList(tags)`.

Step 5: `EmailTags` checks the conditions of the each tags in `tags`, we say that a tag exists if there is email with the tag. If none of the emails has the tag, we say that the tag does not exist. We say that both tags co-exist if there is email tagged with both tags.

- In this example, both tags `Fun` and `Project` co-exist.

Step 6: `EmailTags` call `TagMap.get("Fun").get("Project")`. `TagMap` returns `emailList` which is the email(s) tagged with both `Fun` and `Project`.

Step 7: `EmailTags` returns a String `responseMsg` to `EmailTagListCommand`.

- `responseMsg` here is:
  "Here is the email tagged with both #Project and #Fun: <list of title of email(s) with both tags>"

Design Considerations

- Alternative 1 (current choice):
  The tags associated with emails is stored in `TagMap`:
  - TagMap is a `HashMap<String, SubTagMap>`:
    - Each `key` in the HashMap is a tag name (we call it `root tag name` here) that exists in the email list.
    - The `value` associated with each `key` is a `SubTagMap`.
  - `SubTagMap` is a `HashMap<String, EmailList>`:
    - Each `key` in the HashMap is a tag name (we call it `sub tag name` here) that co-exists with the `root tag name` from the `TagMap`. We say that both tags co-exist if there is email tagged with both tags.
    - The `value` associated with each `key` is an `EmailList`, which is the list of emails tagged with both `root tag name` and `sub tag name`.
  - For example, let `emailOne` be an email tagged with `Tutorial` and `CS2113T`, `emailTwo` be an email tagged with `Tutorial` and `CG2271`.
    - After calling `EmailTags#updateEmailTagList`, the TagMap has the following structure:
      {
      Tutorial={Tutorial=emailOne emailTwo, CS2113T=emailOne, CG2271=emailTwo},
      CS2113T={CS2113T=emailOne, Tutorial=emailOne},
      CG2271={CG2271=emailTwo, Tutorial= emailTwo}
      }
  - Pros: Faster search when user invokes `EmailTagListCommand`, since `EmailTags#displayEmailTagList` is navigating in the HashMap.
  - Cons: Current implementation invokes the `EmailTags#updateEmailTagList` on every user input

to keep the tagMap and email list view in GUI updated, which increases the computational load.

- Alternative 2:
  Loop through each tag of each email in the list of emails, and check if the each tag equals to the tag requested by the user, if yes, add the email to the list, if no, continue with the loop. After finishing the loop, output the email(s) in the list.
  - Pros: This implementation does not have to maintain a TagMap structure to keep track of the emails with the tags, therefore does not requires update of the TagMap, this saves the space and computational load of the program.
  - Cons: Slower search when user invokes `EmailTagListCommand`, since it has to loop through each tag of each email in the list of emails.

Logging

We are using `java.util.logging` package for logging. The `LogsCenter` class is used to manage the logging levels and logging destinations.

Current Implementation

- The logging level can be controlled using the logLevel
- 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 in `data/logs` folder with the format `"log" + "yyyyMMdd_HHmm" + ".log"`.
- 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 is not usually noteworthy but may be useful in debugging e.g. print the actual list instead of just its size

Dev Ops

Build Automation

We use Gradle for _build automation. See Gradle Tutorial for more details.

Continuous Integration

We use Travis CI to perform *Continuous Integration* on our projects.

Coverage Reporting

We use Coveralls to track the code coverage of our projects.

Making a Release

Here are the steps to create a new release.

1. Update the archiveVersion number of shadowJar in `build.gradle`.
2. Generate a JAR file using Gradle.
3. Tag the repo with the version number. e.g. `v0.1`
4. Create a new release using GitHub and upload the JAR file you created.

Managing Dependencies

A project often depends on third-party libraries. For example, **Email manager** depends on the JavaFX for GUI support. Managing these *dependencies* can be automated using Gradle. For example, Gradle can download the dependencies automatically, which is better than these alternatives:

a. Include those libraries in the repo (this bloats the repo size)

b. Require developers to download those libraries manually (this creates extra work for developers)