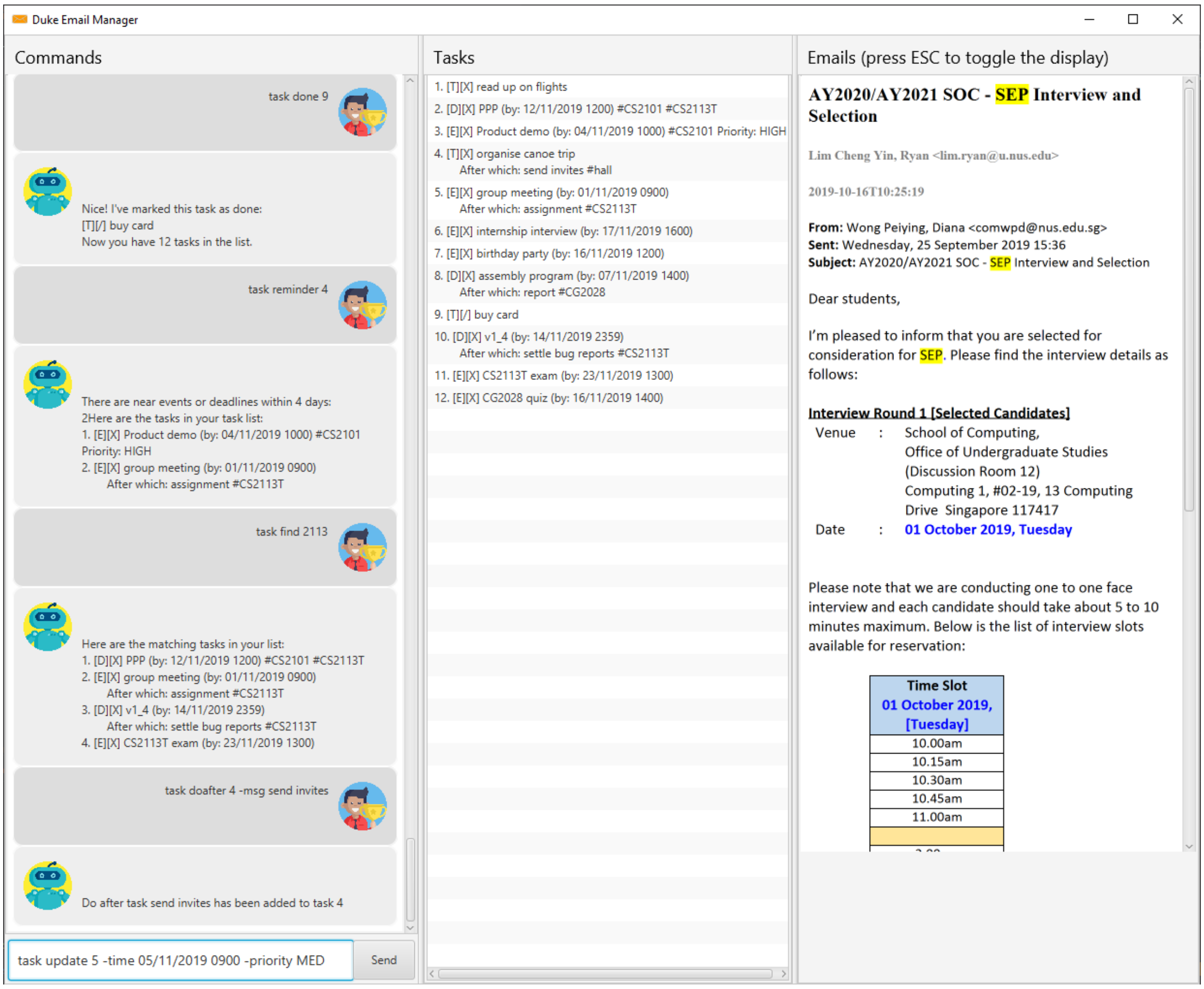
Email Manager - Developer Guide

By: `Team AY1920S1-CS2113T-F11-3` Since: `Nov 2019` Licence: `MIT`

1. Introduction

Welcome to the **Email Manager** Developer Guide!



```
task done 9

Nice! I've marked this task as done:
[T][/] buy card
Now you have 12 tasks in the list.

task reminder 4

There are near events or deadlines within 4 days:
2Here are the tasks in your task list:
1. [E][X] Product demo (by: 04/11/2019 1000) #CS2101
Priority: HIGH
2. [E][X] group meeting (by: 01/11/2019 0900)
      After which: assignment #CS2113T

task find 2113

Here are the matching tasks in your list:
1. [D][X] PPP (by: 12/11/2019 1200) #CS2101 #CS2113T
2. [E][X] group meeting (by: 01/11/2019 0900)
      After which: assignment #CS2113T
3. [D][X] v1_4 (by: 14/11/2019 2359)
      After which: settle bug reports #CS2113T
4. [E][X] CS2113T exam (by: 23/11/2019 1300)

task doafter 4 -msg send invites

Do after task send invites has been added to task 4

task update 5 -time 05/11/2019 0900 -priority MED
```

```
1. [T][X] read up on flights
2. [D][X] PPP (by: 12/11/2019 1200) #CS2101 #CS2113T
3. [E][X] Product demo (by: 04/11/2019 1000) #CS2101 Priority: HIGH
4. [T][X] organise canoe trip
      After which: send invites #hall
5. [E][X] group meeting (by: 01/11/2019 0900)
      After which: assignment #CS2113T
6. [E][X] internship interview (by: 17/11/2019 1600)
7. [E][X] birthday party (by: 16/11/2019 1200)
8. [D][X] assembly program (by: 07/11/2019 1400)
      After which: report #CG2028
9. [T][/] buy card
10. [D][X] v1_4 (by: 14/11/2019 2359)
      After which: settle bug reports #CS2113T
11. [E][X] CS2113T exam (by: 23/11/2019 1300)
12. [E][X] CG2028 quiz (by: 16/11/2019 1400)
```

AY2020/AY2021 SOC - SEP Interview and Selection

Lim Cheng Yin, Ryan <lim.ryan@u.nus.edu>

2019-10-16T10:25:19

From: Wong Peiying, Diana <comwpd@nus.edu.sg>
Sent: Wednesday, 25 September 2019 15:36
Subject: AY2020/AY2021 SOC - SEP Interview and Selection

Dear students,

I'm pleased to inform that you are selected for consideration for SEP. Please find the interview details as follows:

Interview Round 1 [Selected Candidates]

Venue	:	School of Computing, Office of Undergraduate Studies (Discussion Room 12) Computing 1, #02-19, 13 Computing Drive Singapore 117417
Date	:	**01 October 2019, Tuesday**

Please note that we are conducting one to one face interview and each candidate should take about 5 to 10 minutes maximum. Below is the list of interview slots available for reservation:

Time Slot 01 October 2019, [Tuesday]
10.00am
10.15am
10.30am
10.45am
11.00am

Email Manager is an email and task manager 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 Command Line Interface (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.

2. About the Developer Guide

This developer guide provides detailed documentation on the implementation of all the various features Email Manager offers. It also suggests methods for you to modify and build upon it.
Throughout this developer guide, there will be various icons used, as shown below:

□	**This is a tip. Follow these tips to aid your development of Email Manager.**

□□	**This is a note. Read these for additional information.**

□□	**This is a warning. Heed these warnings to avoid making mistakes that will hamper your development efforts.**

3. Setting up

This section shows how to set up Email Manager on your desktop and begin your development journey.

4. Design

4.1. Architecture

4.1.1. How the architecture components interact with each other

The *Architecture Diagram* given above explains the high-level design of the App. Given below is a quick overview of each component.

4.1.2. UI component

4.1.3. Logic component

4.1.4. Model component

4.1.5. Common classes

5. Implementation

This section describes some noteworthy details on how certain features in **Email Manager** are implemented.

5.1. Task Management

There are three main types of tasks that Email Manager works with: `todo`, `deadline` and `event`.

5.1.1. Natural Dates Support

The email manager aims to help computing students handle their tasks efficiently. Therefore, one of its main goals is to speed up the process at which students enter their task details so that their task can be added into the task list quickly. The benefits of having this Natural Dates support are:

- Reduce the time and effort needed to key in the date and time for deadline and event tasks.

The Natural Dates support is facilitated by two main classes, namely `TaskCommandParseHelper` and `TaskParseNaturalDateHelper`.

`TaskParseNaturalDateHelper` is an element of the will retrieve the parsed time string from `TaskCommandParseHelper` and convert the extracted string to LocalDateTime format. It implements the following operations:

- `TaskParseNaturalDateHelper#isCorrectNaturalDate(input)` - Checks if input contains natural date format.
- `TaskParseNaturalDateHelper#convertNaturalDate(day, time)` - Converts string day and time to local date in LocalTimeDate format.
- `TaskParseNaturalDateHelper#getDate(input)` - Returns a date and time(if applicable) after checking if natural date input contains a time element.

`TaskCommandParseHelper` is an element of the Command component. It handles all parsing of inputs when the input type is set to `task`.

Given below is an example usage scenario and how Natural Dates Support behaves at each step.

Step 1: The user launches the application. The input type is currently in `email` mode. The user wishes to add a task and key in `flip` to switch input type to `task` mode.

Step 2: The user executes `deadline homework -time Mon 1200` to add a new deadline task.

- `TaskCommandParseHelper` takes in the command, parses and extracts the date and time information of the task and saves it inside a list of type `ArrayList<Command.Option>`.
- The extracted date and time will go through `TaskParseNaturalDateHelper#getDate()`, which calls the relevant methods in the class to process the date and time.

Step 3: The user wishes to update the date and time for the task above, let the task above be task 1 in the task list. The user executes `update 1 -time Tue` to change the task date from Mon to Tue.

- `TaskParseNaturalDateHelper` will be called to process the time information and update the task accordingly. When no time is entered, the time is set to 0000 (HHmm).

5.1.2. Updating of Tags

Tasks have a variety of attributes that a user may want to add or change. The current attributes that can be modified are: `time`, `doafter`, `priority` and `tags`. The updating of task details is facilitated by `TaskUpdateCommand`. It extends `Command`, overriding the `execute` method.

The details of the update mechanism are as follows:

Step 1: `TaskCommandParseHelper` takes in the command from the user input text field, and parses it. If the command starts with `update` than it will separate all the parameters into an ArrayList, passing it into the `TaskUpdateCommand`.

Step 2: `TaskUpdateCommand.execute()` will then go through the ArrayList, calling the appropriate method in `TaskList`.

Step 3: The method in `TaskList` will retrieve the specified task and call the related setter to change the value.

5.2. Email Management

5.2.1. Email Auto Parsing

The emails fetched or stored locally will be automatically parsed to extract important information for tagging, task creation and reminder purposes. The parsing consists of two stages, the **format parsing** and **content parsing**. Email format parsing is to parse the email components like subject, sender and body from the raw string fetched from the server or stored in local file. The content parsing is to parse the keyword included all components of email.

Email Format Parsing

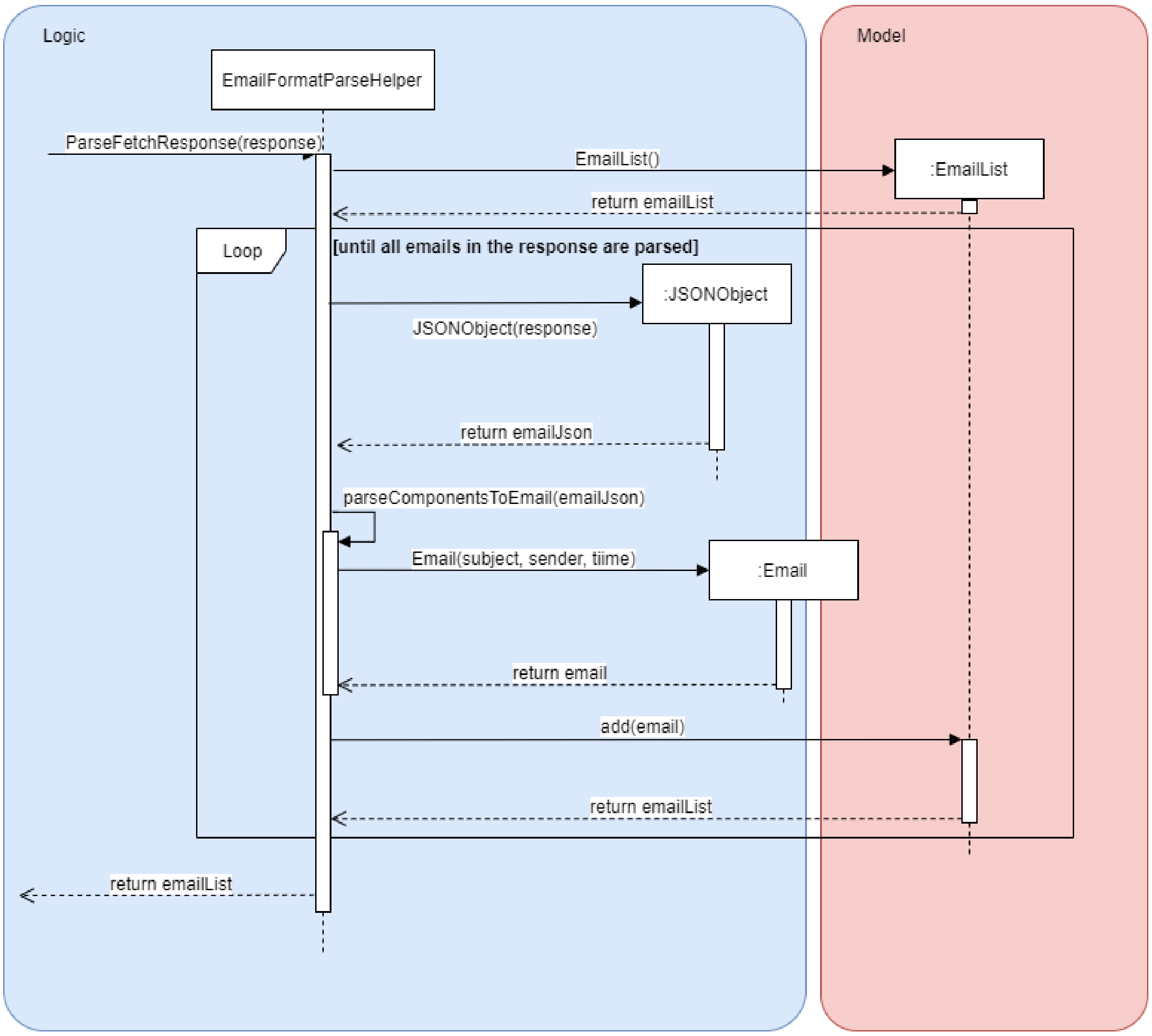


Figure N: Email Format Parsing

The email format parsing starts at the call of `ParseFetchResponse(response)`. The variable `response` here is a string of the http response from the Outlook server to the fetch API call.

It first creates an `EmailList` to store all the emails parsed from the response. This function only returns the list instead of directly adding the email parsed to the model because the storage or network component will decide whether and how the emails are to added to the model.

Then each email contained in the response of is parsed to a `JSONObject` called `emailJson` for easier manipulation.

With this `emailJson`, `parseComponentsToEmail(emailJson)` is called to extract different components of the json and instantiate an `Email` object to be added to the `emailList` created earlier.

This process repeats until all the email information in the response is processed.

Email Content Parsing

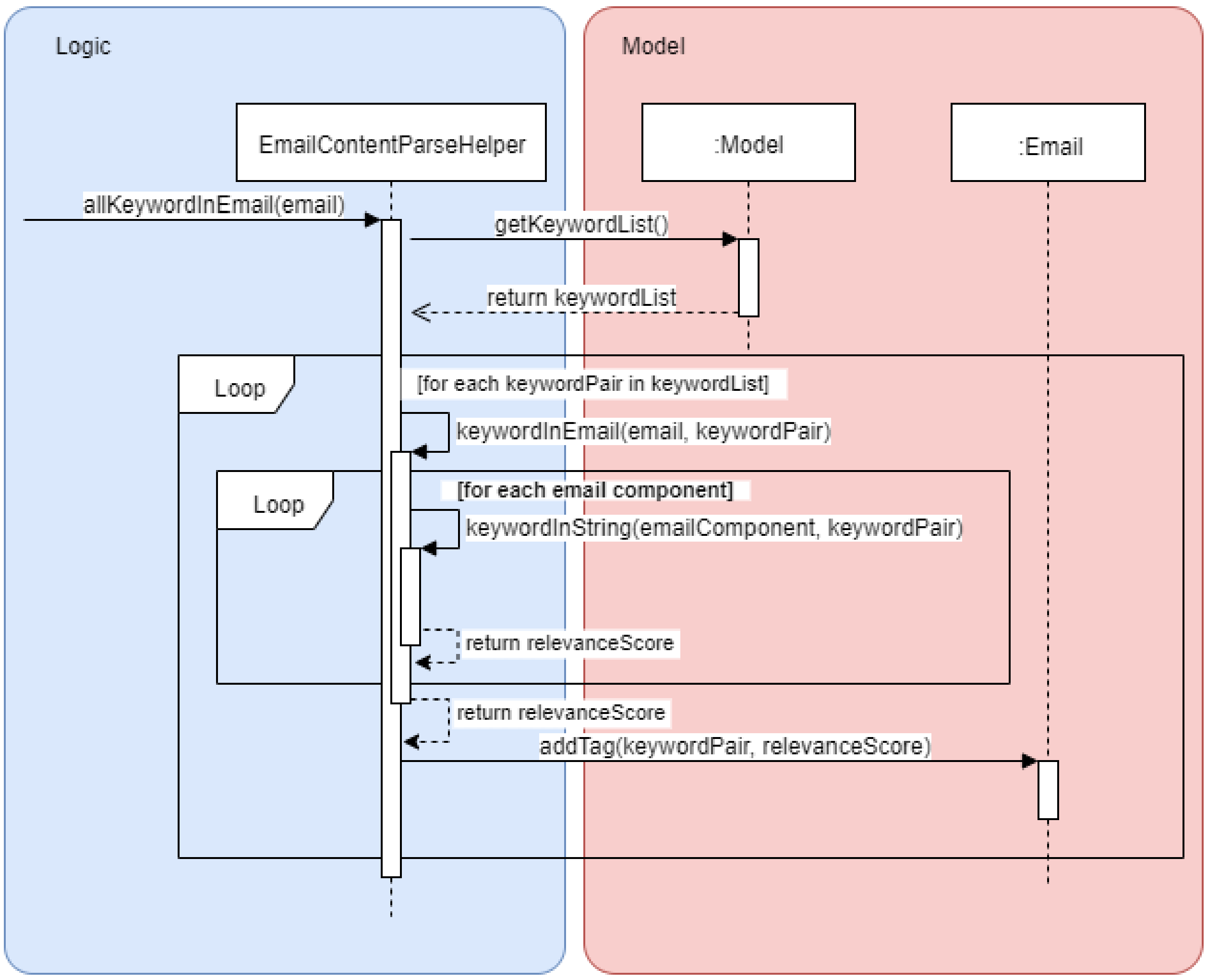

Logic
Model
EmailContentParseHelper
:Model
:Email
allKeywordInEmail(email)
getKeywordList()
return keywordList
Loop
[for each keywordPair in keywordList]
keywordInEmail(email, keywordPair)
Loop
[for each email component]
keywordInString(emailComponent, keywordPair)
return relevanceScore
return relevanceScore
addTag(keywordPair, relevanceScore)

Figure N+1: Email Content Parsing

Email content parsing is to parse the keyword from different components of an email.

Email content parsing starts with the calling of `allKeywordInEmail(email)` and parse in the email in `Email` type.

It then gets all the `KeywordPair` from the `KeywordList`. A `KeywordPair` contains a human readable `keyword` signifying the "meaning" of the keyword. It also contains multiple `expressions` which will be looked through the email for matches.

For each email component (subject, sender, body etc), the `keywordInString(emailComponent,` `keywordPair)` is called to search for matches across these components. It will then return a relevance score. Higher relevance score means a more occurrence. Keyword with all its expressions absent in the email will get a 0 relevance score. Also, subject and sender will have a higher weights compared to the email body.

Each keyword searched in the email with a relevance score higher than 0 (at least 1 occurrence), will be added to the email as a tag by calling `addTag(keywordPair, relevanceScore)`.

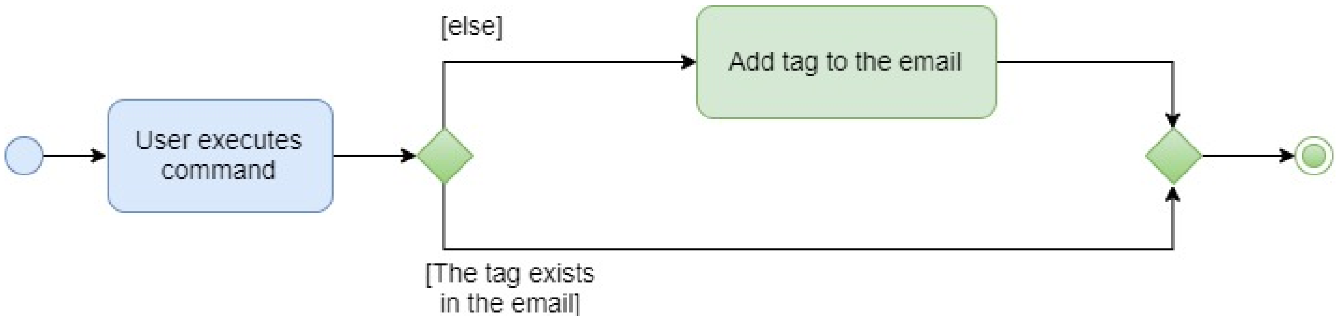
5.2.2. Email Tagging

Email Manager allows user to tag emails by tags.

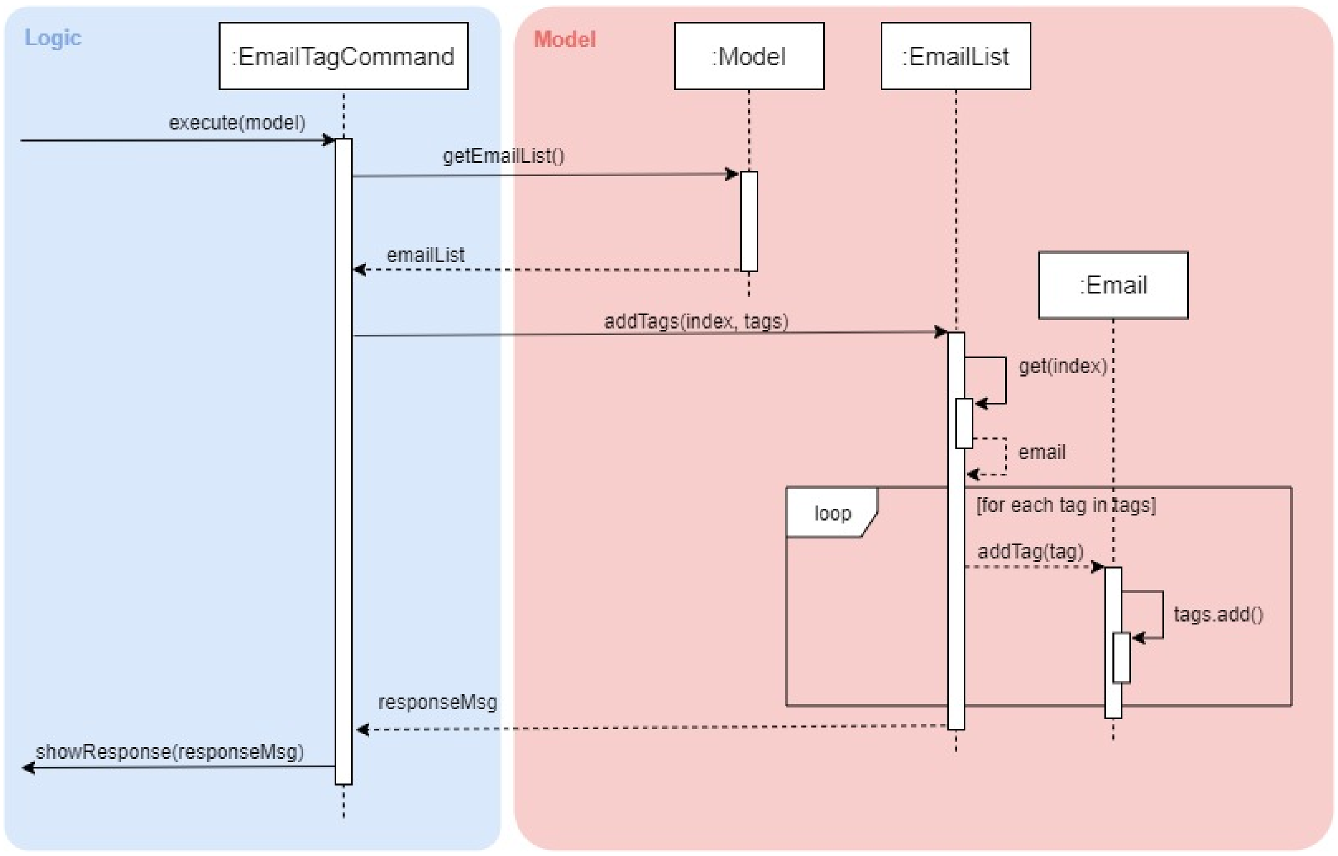
Current Implementation

- 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:



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



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

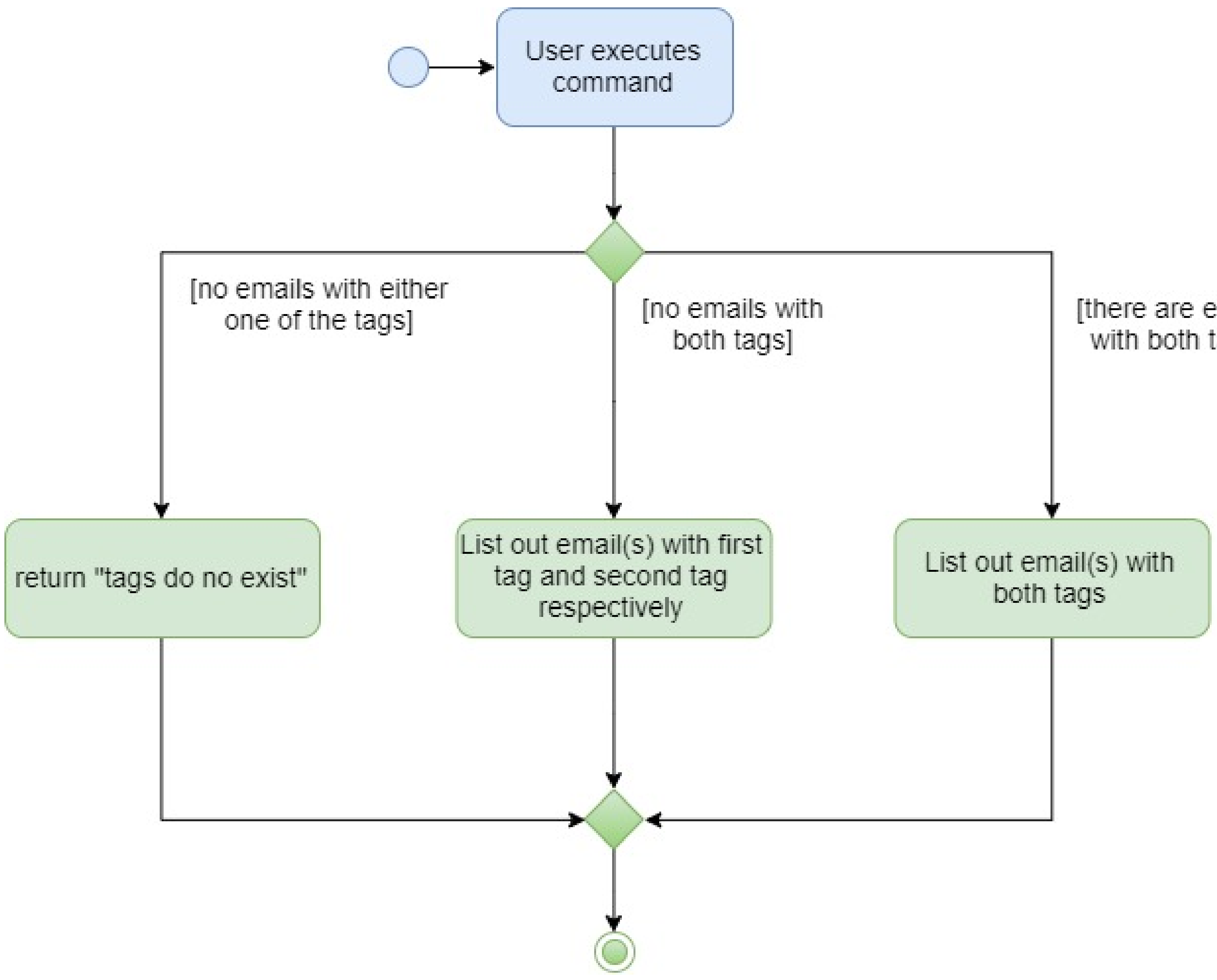
5.2.3. Email Filtering by Tag(s)

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

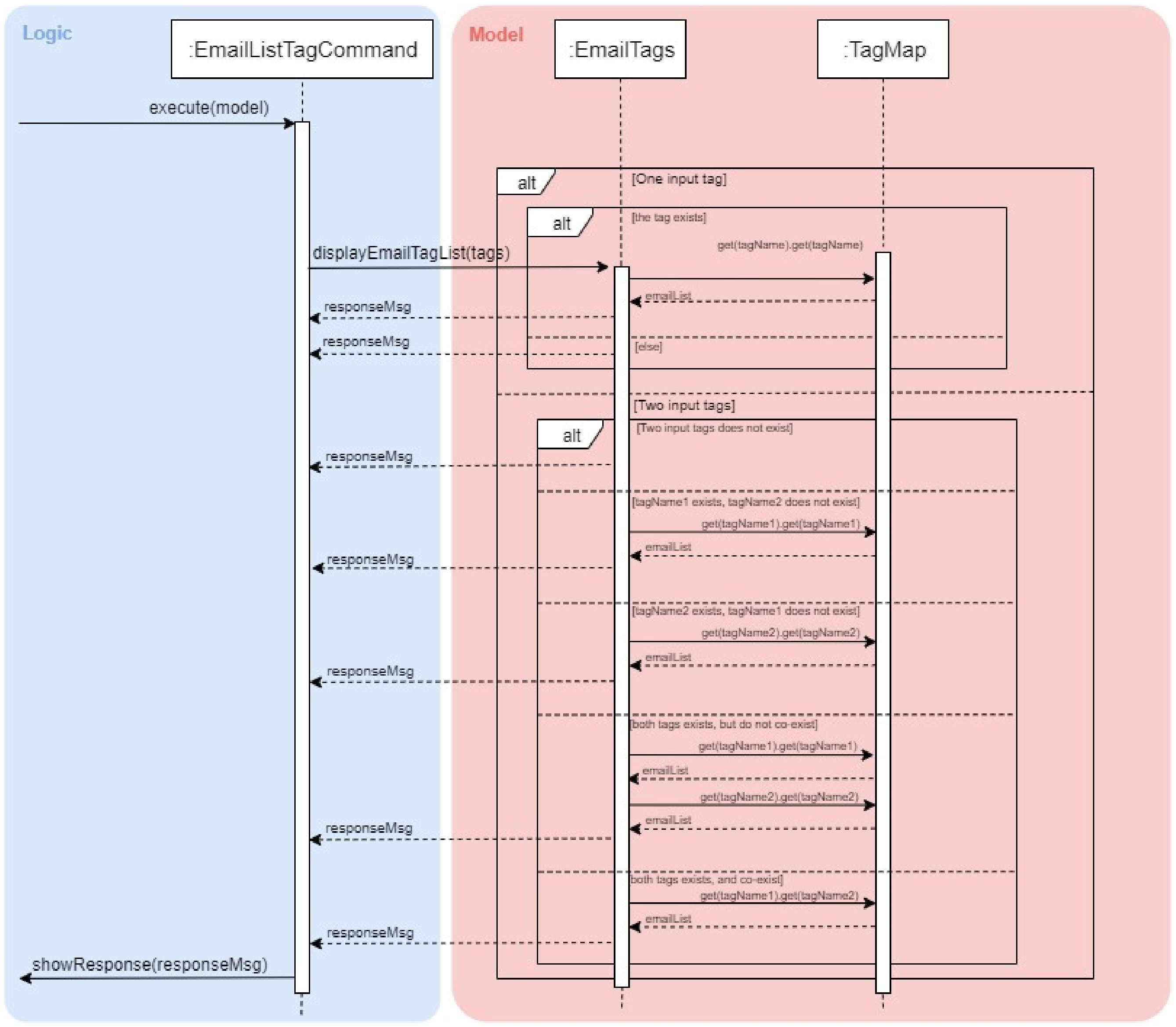
Current Implementation

- 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:


User executes command
[no emails with either one of the tags]
[no emails with both tags]
[there are emails with both tags]
return "tags do no exist"
List out email(s) with first tag and second tag respectively
List out email(s) with both tags

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


Logic
:EmailListTagCommand
Model
:EmailTags
:TagMap
execute(model)
alt
[One input tag]
alt
[the tag exists]
displayEmailTagList(tags)
get(tagName).get(tagName)
emailList
responseMsg
responseMsg
[else]
[Two input tags]
alt
[Two input tags does not exist]
responseMsg
[tagName1 exists, tagName2 does not exist]
get(tagName1).get(tagName1)
emailList
responseMsg
[tagName2 exists, tagName1 does not exist]
get(tagName2).get(tagName2)
emailList
responseMsg
[both tags exists, but do not co-exist]
get(tagName1).get(tagName1)
emailList
get(tagName2).get(tagName2)
emailList
responseMsg
[both tags exists, and co-exist]
get(tagName1).get(tagName2)
emailList
responseMsg
showResponse(responseMsg)

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.

5.3. 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

6. Documentation

7. Testing

8. Dev Ops

8.1. Build Automation

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

8.2. Continuous Integration

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

8.3. Coverage Reporting

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

8.4. 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.

8.5. 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)

Appendix A: Suggested Programming Tasks to Get Started

Appendix B: Product Scope

Target user profile:

1. National University of Singapore (NUS) School of Computing Students.
2. Busy computing student who is tired of receiving too many emails.
3. Busy computing student who has a lot of todos, deadlines and events.
4. Students who prefer desktop apps over other types.
5. Students who prefer typing over other means of input.

Value proposition:

1. Helps busy computing student to manage their emails.
2. Helps busy computing student to manage their tasks and schedules.
3. Reminds busy computing students of their important emails and tasks.

Appendix C: User Stories

Priorities: High (must have) - * * *, Medium (nice to have) - * *, Low (unlikely to have) - *

As a/an	I can	So that…	Priority
active student	get emails filtered out on student-life activities	I can get the interesting event info immediately	* * *
advanced user	tag the emails	I can search for them efficiently	* * *
busy student	prioritize my tasks by setting priority levels	I can work on more pressing task first	* * *
busy student	assign emails with color codes according to priority	I can have a clear view of priorities	* * *
computing student	filter out different types of emails by specifying the type we want to filter	I can access the type of emails we want easily	* * *
computing student	sort email according to module code	I can easily access the information related to my project	* * *
computing student	set auto delete function to delete emails from a specific address	I can keep my mailbox clean	* * *
computing student	filter out emails by specifying a keyword	I can access the email I am interested easily	* * *
computing student	auto-categorize the emails	my mailbox is not messy	* * *
email sender	send emails with tags	recipients can filter emails easily	* * *
email user	get the list of unread emails	I can attend the unread easily	* * *
job hunting student	add alarm to job/internship application deadlines	I will not miss any important application deadlines	* * *

As a/an	I can	So that...	Priority
advanced user	advance search based on Regular expression	I can search with complex filters	* *
advanced user	use shorter versions of command and auto-completion of command	I can type faster command	* *
busy student	check my calendar to see if there are new added	I can keep track of my task efficiently	* *
busy student	highlight tasks that are due soon (<24 hours)	I can work on things that are more pressing	* *
busy student	set alarm to review some important emails	I can remember to attend to some important emails that I don't have time to handle now	* *
busy student	sync with NUSMODS to automatically set deadlines for homework	I can my deadlines or homework assigned to a specific time	* *
busy student	undo my previous command	recover to the previous state	* *
computing student	set important emails to reply by a specific date	I won't miss any important deadlines	* *
computing student	update my calendar if the email contains a date	I won't miss out important deadline	* *
computing student	download all uploaded files sent through emails	I can get the latest version of the file	* *
computing student	get connected with list on contacts on email	I can easily send to or find the email user	* *
computing student	filter out competition/hackathon emails	I won't miss any interesting competition	* *
computing student	find teams for competition/hackathon	I can quickly find teams after the competition email is sent out	* *

As a/an	I can	So that…	Priority
email user	send, forward or reply to email	I do not need to switch to email app after reading from this app	* *
email user	highlights links, action items	I can take action and quickly get to a website	* *
email user	automatically restore emails that were thrown to the junk mail by the system	I won’t miss any important emails just because they are in the junk mail without me knowing it	* *
job hunting student	put away all outdated emails on internships/jobs	I can focus on the newest and valid ones	* *
job hunting student	get the jobs and internship emails sorted	I can find a job	* *
team member	tag emails with project stages	I can access emails from different stages of our project	* *
team member	sort all emails from my team members together	I can easily access the information related to my project	* *
team member	send progress tracking emails to other team members periodically	The team can be always updating each other’s progress	* *

Appendix D: Use Cases

System: `Email Manager`

Actor: `User` (SoC student)

Use Case: Fetch email from account

MSS

1. User starts system or enters the fetch command.
2. System retrieves account key from file, connects to Microsoft and logs in. New emails are retrieved, combined with those from local storage and displayed.

Use case ends.

Extensions

- 2a. System is unable to retrieve a valid account key, receives error from Microsoft.
  - 2a1. Opens Microsoft portal in browser.
  - 2a2. The user types in the username and password into Microsoft portal.
  - 2a3. System saves the account key from Microsoft for future logins, downloads new emails, and displays email.

    Use case ends.
- 2b. System does not receive a response from Microsoft server.
  - 2b1. System notifies user of failure to log in and loads email from local file.

    Use case ends.

Use Case: Set priority to different keyword

MSS

1. The user selects the "Keyword Priority" from the menu
2. System presents the user with all current priority settings
3. The user selects "New" from the menu
4. System displays a text box for input of keyword/regular expression.
5. The user types in the keyword/regular expression, selects the priority level of this keyword and selects "Save" and confirm

   Use case ends.

Use Case: View Email List with Priority

MSS

1. The user selects "All Email" from the menu
2. The user selects "by Priority" from the menu
3. System displays all the priorities and keywords under each priority
4. The user selects the priority range that he/she wants to view
5. System leads the user back to the email list page

   Use case ends.

Extensions * 3a. The user can include or exclude a particular keyword from that priority

+ Use case ends.

Use Case: Auto categorisation of emails

MSS

1. User creates a new categorize name.
2. User specifies the keyword for this category.
3. The app will look through the emails and put the related-emails under the category.

   Use case ends.

Appendix E: Non-Functional Requirements

Email Manager meets the following non-functional requirements:

- Security of user login credentials (user enters details directly into Microsoft portal)
- Data Accessibility (efficient storage)
- Time-out
- Clean layout
- Works with common operating systems

Appendix F: Glossary

Mainstream OS

- Windows
- macOS
- Linux

Appendix G: Instructions for Manual Testing

Please follow the following feature testing tables to do the manual testing. The actual output and error message are to be checked against the expected output in the table. If Internet access is not available, please copy the content in the `/data/test_data` folder to the `/data` folder and overwrite all. These two folders will be automatically generated once the program is started.
Please ensure the program is copied to an empty folder before starting the testing to avoid any data contamination. Any manual changes to the data folder might affect the program from function normally and is not recommended.

NOTE As we are using key mapping for certain hotkeys, you will not be able to use `Ctrl + V` for pasting. However, you can right click on the input box and choose paste to paste the test input.

NOTE These instructions only provide a starting point for testers to work on; testers are expected to do more exploratory testing.

G.1. Feature Testing

G.1.1. Email Auto Parsing and AddKeyword Command

Since the email content parsing is automatic, it is easier to be tested together with the addKeyword command.

NOTE All the expression match is case insensitive and full match, meaning that 'cs' can be matched to 'CS' but not 'CS2101'.

Mode	Input	Expected Result	Purpose of Testing
`email`	`addKeyword Notice -exp announcement`	All emails with the word 'announcement' in its subject, sender or body will have #Notice on the list	Normal use case of addKeyword and auto parsing
`email`	`addKeyword Project -exp project -exp demo`	All emails with the word 'project' or 'demo' in its subject, sender or body will have #Project on the list	Normal use case with multiple expressions in one keyword
`email`	`addKeyword Project Demo -exp project -exp demo`	All emails with the word 'project' or 'demo' in its subject, sender or body will have #Project Demo on the list	Normal use case with space in keyword
`email`	`addKeyword Demo -exp final demo`	All emails with the word 'final demo' in its subject, sender or body will have #Demo on the list	Normal use case with space in expression
`email`	`addKeyword Project Demo -exp luminus`	All emails with the word 'luminus' in its subject, sender, or body will have #Project Demo on the list. Those emails already have #Project Demo will not be affected	Normal use case with new expression but existing keyword
`email`	`addKeyword 46 -exp 46`	All emails with '46' in its subject, sender or body will have #46 on the list	Normal use case with numbers in keyword and expression

Mode	Input	Expected Result	Purpose of Testing
`email`	`addKeyword`	Invalid command reported asking for a keyword after `addKeyword`	Invalid input without keyword specified
`email`	`addKeyword -exp project`	Invalid command reported asking for a keyword after `addKeyword`	Invalid input with expression but without keyword specified
`email`	`addKeyword project`	Invalid command reported asking for expression options	Invalid input without expression specified
`email`	`addKeyword project -exp`	Invalid command format reported	Empty expression in expression option should trigger the invalid command format check as option must not be empty
`email`	`addKeyword project -exp !@#`	Invalid command format reported	Invalid character in input will trigger the invalid command format check

G.1.2. Email Fuzzy Search

NOTE For performance reasons, fuzzy search will only be done word by word. For example, if "project demo" is to be searched through a sentence "This is a project demo", both "project" and "demo" will be compared against "this", "is", "a", "project", "demo" and produce a relevance score.

NOTE Using short target string like "is" is not recommended, since it can be matched to many other words like "a", "I", "am" etc, which appears in almost every email.

Mode	Input	Expected Result	Purpose of Testing
`email`	`fuzzySearch Aggarwal`	A few emails sent by Divesh Aggarwal will be listed	Normal use case with exact match should be captured by fuzzy search
`email`	`fuzzySearch Aggarwa`	A few emails sent by Divesh Aggarwal will be listed	Normal use case with edit distance of 1 should be captured by fuzzy search

Mode	Input	Expected Result	Purpose of Testing
`email`	`fuzzySearch Aggarw`	A few emails sent by Divesh Aggarwal will be listed	Normal use case with edit distance of 2 should be captured by fuzzy search
`email`	`fuzzySearch Aggar`	No email will be listed	Normal use case with edit distance of 3 will not be captured by fuzzy search
`email`	`fuzzySearch Divesh Aggarwal`	A few emails sent by Divesh Aggarwal will be listed	Normal use case with more than two target words should be captured properly
`email`	`fuzzySearch CS2113`	Many emails about CS2113T, CS2101, CS2102 should be listed, with CS2113T emails generally listed first	Normal use case when multiple words can be matched but the most relevant should be listed first

G.2. Launch and Shutdown

1. Initial launch
   i. Download the jar file and copy into an empty folder
   ii. Double-click the jar file
   Expected: Shows the GUI with a set of sample tasks and emails. The window size may not be optimum.
2. Exiting the program
   i. Type exit into the user input box.
   ii. Expected: Application will shut down and close itself.