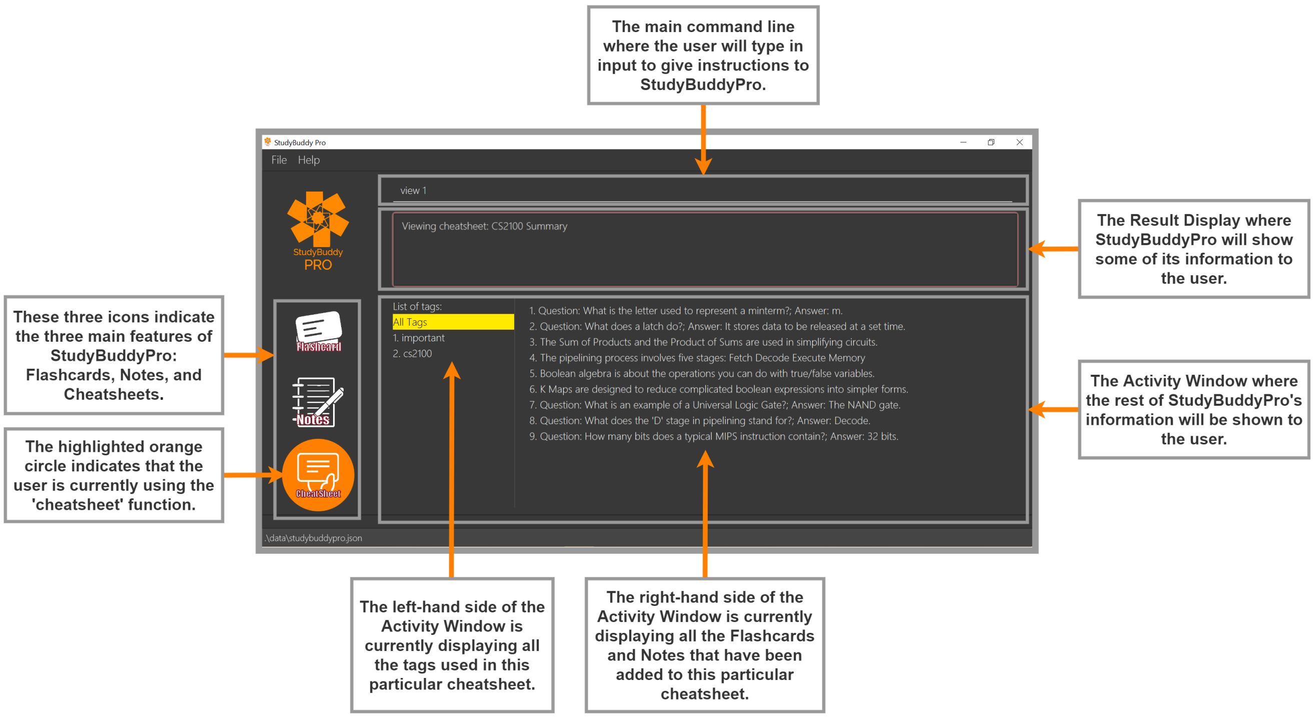
Samuel Lim - StudyBuddyPro Project Portfolio

About the Project

This Project Portfolio is intended to document and detail my personal contributions to my software engineering project for the module CS2103T Software Engineering. In this module, me and my team of four other software engineering coursemates were tasked with morphing a given codebase for a desktop app that primarily made use of typed input to receive instructions. We decided to morph the app into an all-in-one study aid application named **StudyBuddyPro**, targeted specifically at NUS Computer Science students who preferred visual learning. Our app supports the use of flashcards and notes, provides a timetrial and reminder feature for flashcards, and includes an automatic cheatsheet-generation tool.

The image below displays the current Graphical User Interface (GUI) of StudyBuddyPro:


The main command line where the user will type in input to give instructions to StudyBuddyPro.
The Result Display where StudyBuddyPro will show some of its information to the user.
These three icons indicate the three main features of StudyBuddyPro: Flashcards, Notes, and Cheatsheets.
The highlighted orange circle indicates that the user is currently using the 'cheatsheet' function.
The Activity Window where the rest of StudyBuddyPro's information will be shown to the user.
The left-hand side of the Activity Window is currently displaying all the tags used in this particular cheatsheet.
The right-hand side of the Activity Window is currently displaying all the Flashcards and Notes that have been added to this particular cheatsheet.
StudyBuddy Pro
File
Help
StudyBuddy PRO
Flashcard
Notes
CheatSheet

```
view 1
```

Viewing cheatsheet: CS2100 Summary

List of tags:
All Tags
1. important
2. cs2100

1. Question: What is the letter used to represent a minterm?; Answer: m.
2. Question: What does a latch do?; Answer: It stores data to be released at a set time.
3. The Sum of Products and the Product of Sums are used in simplifying circuits.
4. The pipelining process involves five stages: Fetch Decode Execute Memory
5. Boolean algebra is about the operations you can do with true/false variables.
6. K Maps are designed to reduce complicated boolean expressions into simpler forms.
7. Question: What is an example of a Universal Logic Gate?; Answer: The NAND gate.
8. Question: What does the 'D' stage in pipelining stand for?; Answer: Decode.
9. Question: How many bits does a typical MIPS instruction contain?; Answer: 32 bits.

.\data\studybuddypro.json

Figure 1. The current GUI of StudyBuddyPro, with annotations

My role in this project was to write the code for Notes, implement their basic functions, and implement the ability to add tags within the content of notes. Additionally, I was tasked with completing the test cases, user guide, and developer guide portions for the entire Notes feature, and any other relevant documentation. This Project Portfolio thus details how I implemented these changes.

Informative Callouts

The following symbols will be used throughout this document to indicate more important pieces of information in this Portfolio:

WARNING	Information that, if not followed, will result in StudyBuddyPro not functioning correctly.
IMPORTANT	Information that is crucial for you to understand the details of this Portfolio.
NOTE	Information that will enrich your understanding of this Portfolio.

`colored words` --- Refers either to a command word or a part of StudyBuddyPro’s architecture.

Summary of Contributions

As mentioned earlier, these are my contributions to my software engineering project:

- **Major Contribution 1**: Implemented **Notes**, along with their basic features.
  - **What**: Notes are pieces of information that contain a title, some content, and some tags. They can be added, deleted, viewed, and filtered.
  - **Why**: Notes can be used by students to store information. They form one of the two main sources of information for cheatsheet creation in StudyBuddyPro, and are therefore essential to the studying process.
- **Major Contribution 2**: Implemented the **Add Tags Within Notes** feature.
  - **What**: This feature allows the user to use a pre-defined syntax to tag specific portions of a note with tags (these tags are henceforth referred to as 'note fragment tags'). Note fragment tags are independent of the note’s original tags, and can be filtered just like normal notes can be.
  - **Why**: This feature can be used by students to add information to cheatsheets much more precisely. It can also be used to highlight specific important parts of a note. This would be very helpful for students to revise more efficiently.
- **Minor Contribution 1**: Implemented the `view` and `viewraw` commands for Notes.
  - **What**: Both commands view a given note, but the `view` command also cleans up the syntax of the note to remove the leftover syntax from the addition of note fragment tags.
  - **Why**: Users will not want the GUI of the app to be cluttered up by note fragment tag syntax. It will be more aesthetically pleasing to have the option to not view the syntax when viewing a note.
- **Code Contributed**: This link contains the code that I wrote for this project: My code
- **Other Contributions**:
  - Project management:
    - Created specific GitHub issues for the final release of version 1.4. (#221, #223, #226, #229)
  - Enhancements to existing features:
    - Wrote test code for features relevant to Notes to improve test coverage. (#332, #336,

#339)

    - Created the GUI for the Note feature. (#173, #182, #212)
    - Connected the Note feature to the Storage component. (#167)
        - Assistance was provided by one of my teammates, Sahil. (#158)

- Documentation:
    - Standardized formatting of User Guide to improve overall consistency. (#349)
- Community assistance:
    - Tested bugs for another group's project during a Practical Exam dry run.

Contributions to User Guide

In this project, I was tasked with writing all portions of the User Guide related to Notes (specifically Section 6 of the User Guide). The sections below hence document my contributions to the User Guide of StudyBuddyPro. In the following 3 pages, I will describe how you can add and view notes from StudyBuddyPro, in an easy-to-follow manner.

IMPORTANT All the operations in this section assume that you are in the *notes* mode. To be sure you are in the *notes* mode, please ensure that you have used the `switch notes` command before this.

Creating a note: `add`

Adds a note from user input with title `TITLE` and content `CONTENT`. The title of the note cannot be a duplicate of an existing note title.

```
Format: add t/TITLE c/CONTENT [tag/TAG]...
```

Example usage:

```
add t/Pipelining Definition c/Pipelining is a process where a processor executes
multiple processes simultaneously. tag/cs2100
```

Expected output:

```
New note added:
    Title: Pipelining Definition
    Content: Pipelining is a process where a processor executes multiple processes
simultaneously.
    Tags: [cs2100]

The added Note has no detected note fragment tags!
```

More advanced usage: Tagging of note fragments is also supported. The note fragment tagging is added at the same time as the note is created.

Note fragment tags are added with content `FRAGMENT_CONTENT`, at least one tag `FRAGMENT_TAG`, and any number of additional tags `ADDITIONAL_FRAGMENT_TAG`:

```
Format (within CONTENT): /* C/FRAGMENT_CONTENT TAG/FRAGMENT_TAG
[TAG/ADDITIONAL_FRAGMENT_TAG]... */
```

IMPORTANT The format for note fragment content is 'C/', not 'c/', and the format for note fragment tags is 'TAG/', not 'tag/'.

WARNING If the format is not followed correctly, the note fragment tag will simply not be added (but the note will still be added). StudyBuddyPro will assume that the user has typed the tags correctly.

In the following example, two note fragment tags are added to the same note fragment:

Example usage:

```
add t/About Notes c/Notes can be /* C/highlighted TAG/highlight TAG/important */ if
needed. tag/about
```

Expected output:

```
New note added:
    Title: About Notes
    Content: Notes can be /* C/highlighted TAG/highlight TAG/important */ if needed.
    Tags: [about]

Note fragment tags detected:
    Title: About Notes
    Content: highlighted
    Tags: [important][highlight]
```

This adds a note with content "Notes can be highlighted if needed.", and a note fragment tag with content "highlighted" and two tags "cs2100" and "important".

NOTE The spaces around the syntax elements of '/*', 'C/' etc are part of the syntax. For example, if a note fragment tag looks like this: '/* C/highlighted TAG/important */if needed', then the resultant Note will look like this: 'highlightedif needed'.

Multiple note fragment tags are allowed. These do not interfere with the other tags of the Note.

IMPORTANT Overlapping note fragment tags are not allowed.

Viewing a note: `view`

Views the note of index `NOTE_INDEX`. If the note contains any note fragment tags, those tags will be hidden.

NOTE To view the note with its note fragment tags, use the `viewraw` command instead (see Section 6.4).

```
Format: view (index)
```

Example usage:

```
view 3
```

Expected output:

```
Viewing note:
    Title: About Notes
    Content: Notes can be highlighted if needed.
    Tags: [about]
```

Viewing a raw note: `viewraw`

Views the note of index `NOTE_INDEX`. The note is shown exactly as written, including all note fragment tags.

```
Format: viewraw (index)
```

Example usage:

```
viewraw 3
```

Expected output:

```
Viewing raw note:
    Title: About Notes
    Content: Notes can be /* C/highlighted TAG/cs2100 TAG/important */ if needed.
    Tags: [about]
```

Contributions to Developer Guide

In this project, I was tasked with writing all portions of the Developer Guide related to Notes (specifically Sections 3.4 and 4.7 of the Developer Guide). The sections below hence document my contributions to the Developer Guide of StudyBuddyPro. In the following 5 pages, I will detail out how I decided on and created my feature. I will also describe the architecture of my feature using various types of diagrams, so that you can more clearly understand how the feature works.

Implementation of Notes

`Notes` contain a `Title` and a `Content`, with optional `Tags` and `NoteFragments`, as shown in the class diagram below:

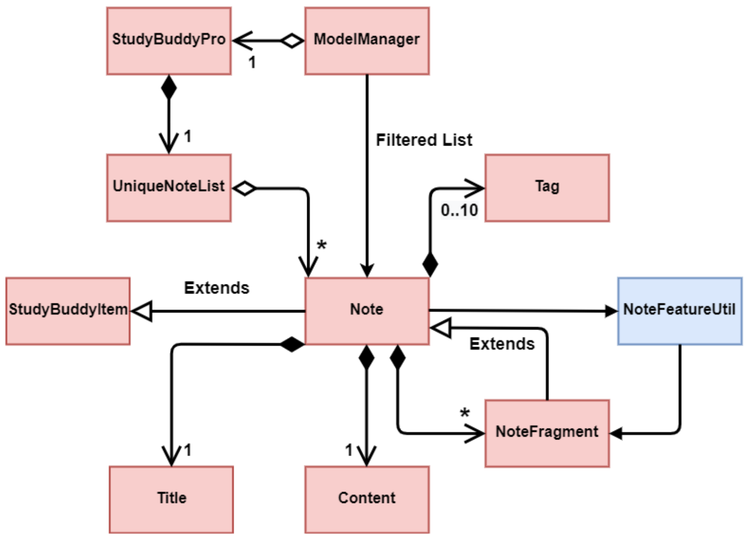


Figure 2. An overview of the Note class and its associated classes

NOTE As is standard in this Developer Guide, red classes are part of the `Model` package, and blue classes are part of the `Logic` package.

Implementation of the Add Tags Within Notes feature

In the following section, I will describe in general how I implemented the Add Tags Within Notes feature, which allows users to store specific sections of their notes with tags.

The Add Tags Within Notes feature was implemented in the following way:

- Creation of a `NoteFragment` class that represents one tag within a `Note`
- Have each `Note` contain any number of `NoteFragments` in a list
- Upon addition of a `Note`, parse its `Content` to check for any note fragment tags within it
- Create any required `NoteFragment` objects and add them to the list in their parent `Note`
- Since `NoteFragments` are contained within `Notes`, when a `Note` is deleted, its `NoteFragments` will be deleted as well.

The following class diagram shows a more specific view of the relationships between the `Note`, `NoteFragment`, and `NoteFeatureUtil` classes.

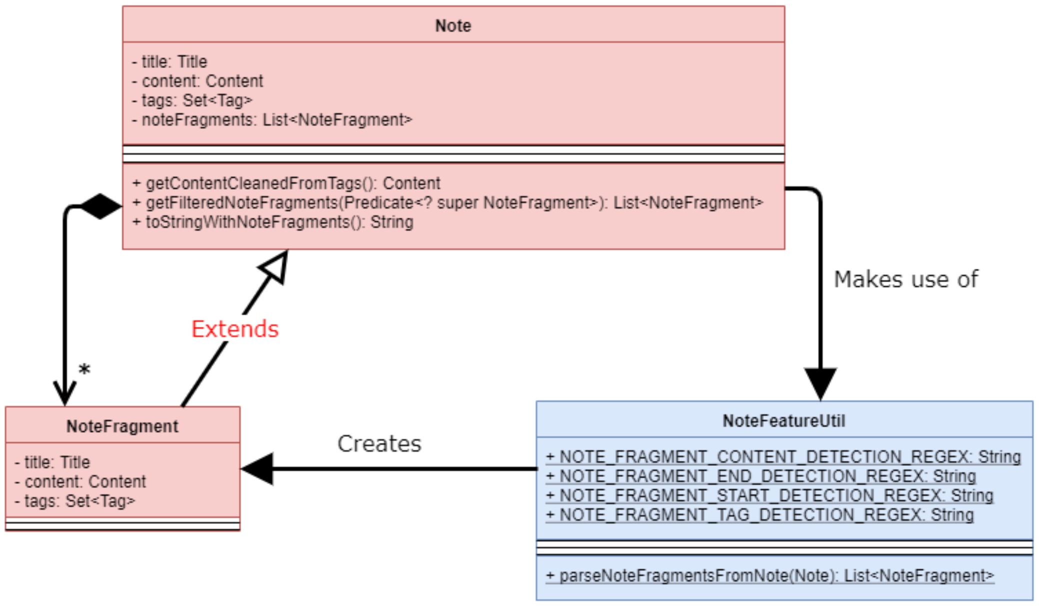

Note
- title: Title
- content: Content
- tags: Set<Tag>
- noteFragments: List<NoteFragment>
+ getContentCleanedFromTags(): Content
+ getFilteredNoteFragments(Predicate<? super NoteFragment>): List<NoteFragment>
+ toStringWithNoteFragments(): String
Makes use of
Extends
*
NoteFragment
- title: Title
- content: Content
- tags: Set<Tag>
Creates
NoteFeatureUtil
+ NOTE_FRAGMENT_CONTENT_DETECTION_REGEX: String
+ NOTE_FRAGMENT_END_DETECTION_REGEX: String
+ NOTE_FRAGMENT_START_DETECTION_REGEX: String
+ NOTE_FRAGMENT_TAG_DETECTION_REGEX: String
+ parseNoteFragmentsFromNote(Note): List<NoteFragment>

Figure 3. A detailed view of the Note, NoteFragment, and NoteFeatureUtil classes

As shown in the diagram above, `NoteFragments` contain:

- A `Title` that is the same as their parent `Note`
- A `Content` that is a substring of their parent `Note`
- Any indicated `Tags` that are independent of their parent `Note`

Implementation of NoteFragment creation

In the following section, I will describe how I implemented the creation of `NoteFragments`. While `Notes` themselves are quite useful, I felt that implementing `NoteFragments` would allow users to gain even more value out of the automatic cheatsheet-generation feature in StudyBuddyPro.

The method of parsing `Content` in `NoteFeatureUtil#parseNoteFragmentsFromNote()` relies on the use of `Prefixes` around each note fragment tag. Each note fragment tag is specified with a start and end marker (represented by the `Prefixes` '/*' and '*/'), and its `Content` and `Tags` are also represented with the `Prefixes` 'C/' and 'TAG/'.

Example usage:

```
add t/About Notes c/Notes can be /* C/highlighted TAG/highlight TAG/important */ if
needed. tag/about
```

NOTE 'C/' and 'TAG/' have to be used instead of the default 'c/' and 'tag/' because otherwise the `Note` would not be parsed correctly (since only the most recent 'c/' tag is used for each command).

A Note has been added with the Content of 'Notes can be /* C/highlighted TAG/highlight TAG/important */ if needed.', and a note fragment tag with Content 'highlighted' and two Tags 'cs2100' and 'important'. The Note itself is instead tagged with the Tag 'about'.

NOTE You can refer to the earlier pages of this Portfolio to recall the specific syntax used in note fragment tags.

The following sequence diagram elaborates upon the creation process of Notes and NoteFragments:

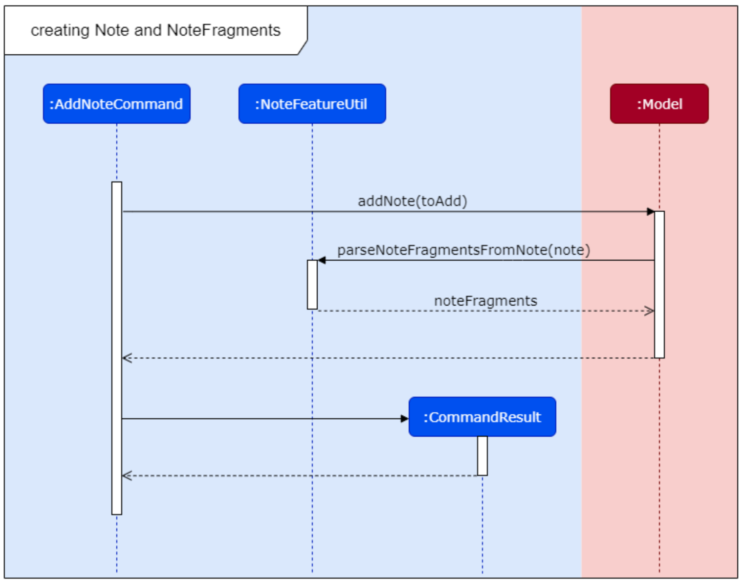


Figure 4. Illustration of how Notes and NoteFragments are created

As can be seen from the zoomed-in diagram above, when an AddNoteCommand is created:

- A Note is created with the desired Title, Content, and Tags.
- The Content of the Note is parsed for the Prefixes '/*', '*/', 'C/', and 'TAG/', using NoteFeatureUtil.
- A List<NoteFragment> is returned to the Note.
- A CommandResult is returned to describe the AddNoteCommand's result.

NOTE All four Prefixes must be present for the note fragment tag to be considered valid. Otherwise, an exception is thrown.

Implementation of ViewNoteCommand

In the following section, I will describe how I implemented ViewNoteCommand. I chose to implement this command because users might not want to see complicated syntax when they are viewing notes.

There are two Commands available if the user intends to view a particular Note: ViewNoteCommand and ViewRawNoteCommand. ViewNoteCommand displays the Note to the user while hiding any note fragment tag syntax from the content of the Note, while ViewRawNoteCommand displays the Note to the user exactly as it was inputted.

The main difference in the process of the two commands is shown in the activity diagram below:

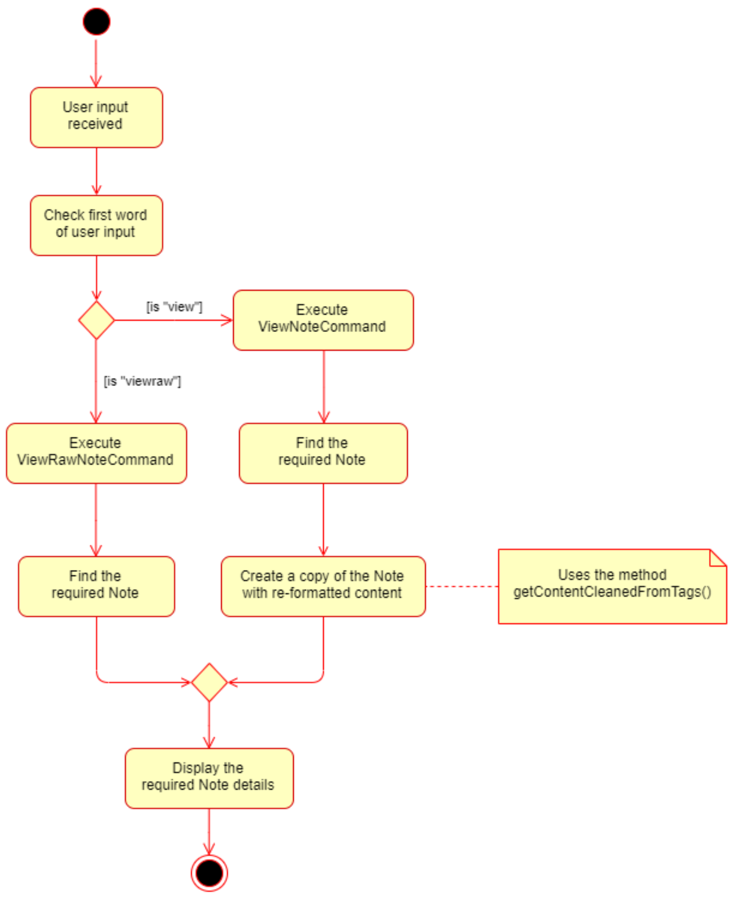

User input received
Check first word of user input
[is "view"]
Execute ViewNoteCommand
[is "viewraw"]
Execute ViewRawNoteCommand
Find the required Note
Find the required Note
Create a copy of the Note with re-formatted content
Uses the method getContentCleanedFromTags()
Display the required Note details

Figure 5. Difference between ViewNoteCommand and ViewRawNoteCommand

As can be seen in the diagram above, the method `Note#getContentCleanedFromTags()` is used to obtain a copy of the `Content` of the `Note`, and reformat it by replacing its note fragment tags with blank spaces. This is further elaborated on by the object diagram below:

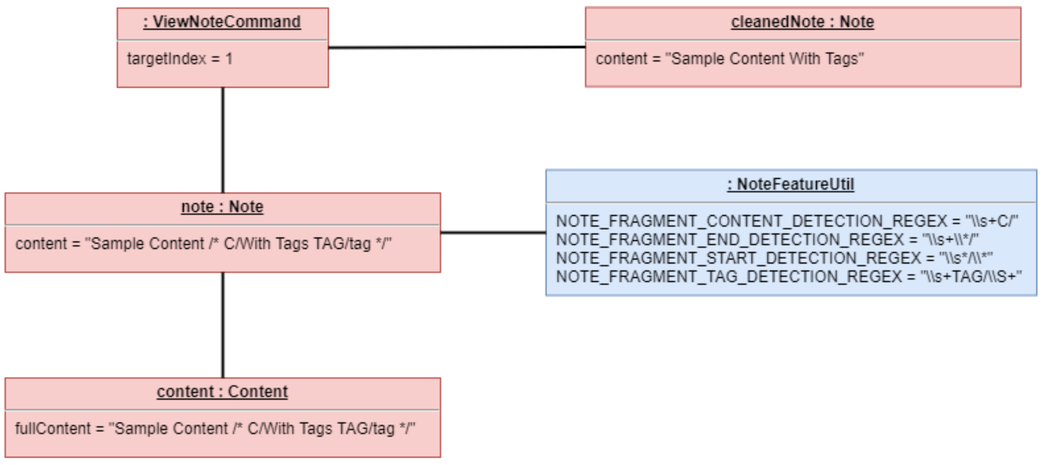

: ViewNoteCommand
targetIndex = 1
cleanedNote : Note
content = "Sample Content With Tags"
note : Note
content = "Sample Content /* C/With Tags TAG/tag */"
: NoteFeatureUtil
NOTE_FRAGMENT_CONTENT_DETECTION_REGEX = "\\s+C/"
NOTE_FRAGMENT_END_DETECTION_REGEX = "\\s+*/"
NOTE_FRAGMENT_START_DETECTION_REGEX = "\\s*/*"
NOTE_FRAGMENT_TAG_DETECTION_REGEX = "\\s+TAG/\\S+"
content : Content
fullContent = "Sample Content /* C/With Tags TAG/tag */"

Figure 6. A possible instance of a use of ViewNoteCommand

As can be seen from the diagram above, the `Note` object named 'note' makes use of the `NoteFeatureUtil` object and its own `Content` object to return the cleaned content String to the `ViewNoteCommand` object. The `ViewNoteCommand` object then creates a separate `Note` object named 'cleanedNote' to obtain the information to display.

This indicates that calling a `ViewNoteCommand` does not affect the original `Note`, as a separate `Note` object is created instead.

Design Considerations of the Add Tags Within Notes feature and ViewNoteCommand

In the following section, I will describe two of the decisions that I had to make while creating my feature, and will elaborate on the pros and cons of each choice.

NOTE Some of the architectural components mentioned in these alternatives (e.g. `UniqueNoteList`) are not mentioned in this Portfolio. Should you wish to read up more about them, you can refer to the full Developer Guide here.

Aspect: Container location for `NoteFragment`

- Alternative 1 (current choice): Each `Note` contains a `List<NoteFragment>`:
    - Pros: Much easier maintenance, since `NoteFragments` are automatically deleted with the deletion of their parent `Note`.
    - Cons: $O(n^2)$ search time for `filter` commands, as all `NoteFragments` in each `Note` in the `UniqueNoteList` must be searched through to filter them out.
- Alternative 2: Separate `UniqueNoteFragmentList` from `UniqueNoteList`:
    - Pros: O(n) search time for `filter` commands, as the `UniqueNoteFragmentList` exists separately from the `UniqueNoteList`.
    - Cons: Difficult to maintain; deletion of a parent `Note` requires searching the `UniqueNoteFragmentList` for any child `NoteFragments` to delete as well.

Aspect: Implementation of `ViewNoteCommand`

- Alternative 1 (current choice): Create a method `Note#getContentCleanedFromTags()` that also returns a Content:
    - Pros: Allows `Note` to remain as the sole point of interaction between the other `Note` classes (specifically `Content` and `Tags`).
    - Cons: Is more complicated as `Content` must be converted to String before manipulation.
- Alternative 2: Create a method `Content#getContentCleanedFromTags()` that returns a cleaned copy of itself.
    - Pros: Can manipulate the String of the `Content` directly without having to convert it to and from `Content`.
    - Cons: Breaks the Single Responsibility Principle, as `Content` will now be dependent on `Tags`, even though it does not use `Tags`.