tp

Developer Guide

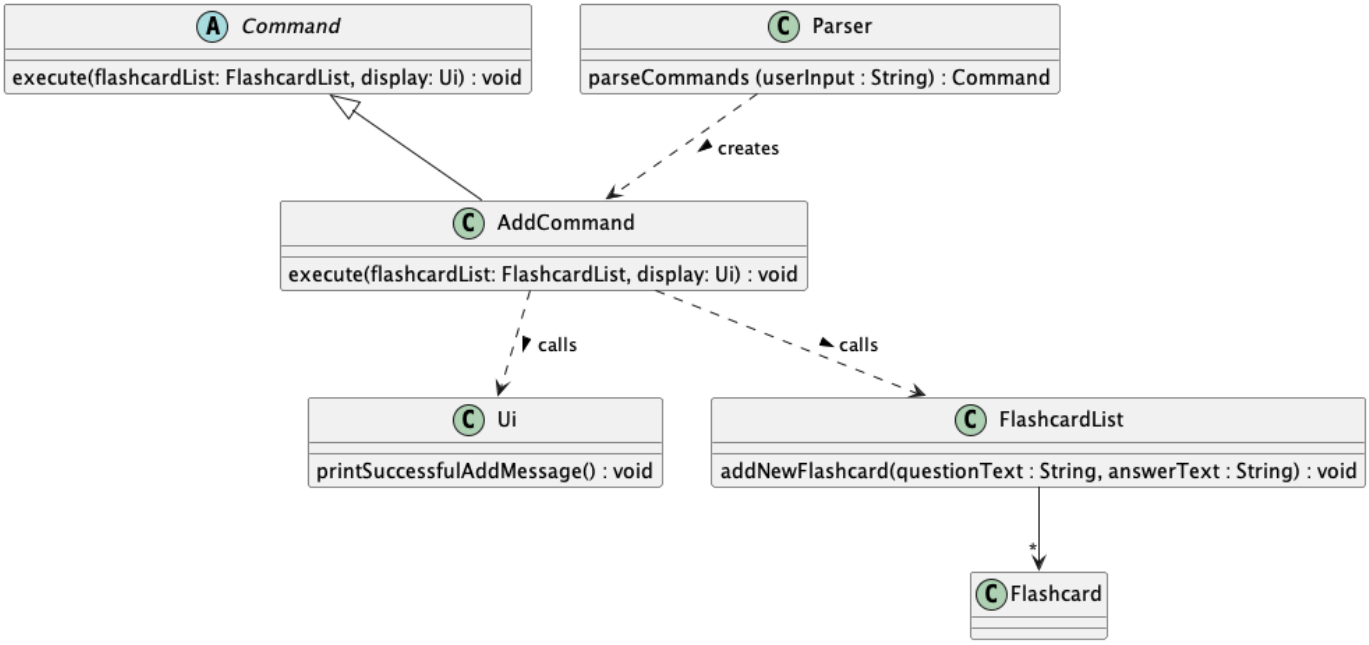
Acknowledgements

- addressbook-level2
- Song Zijin's IP

Design & Implementation

Add Flashcard Feature

The image below shows a partial class diagram involving only the relevant classes when an AddCommand is created and executed:



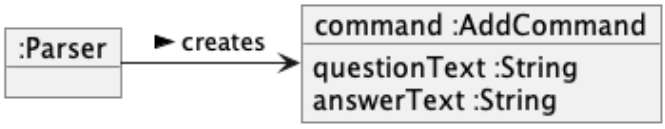
Current Implementation

The current add flashcard allows the user to add a flashcard to the list of flashcards, it is implemented through the following steps:

Step 1: The input of user is collected by `getUserCommand()` inside class `Ui`.

Step 2: The input string will be converted into a `Command` object by being passed through `parseCommand(String userInput)` inside `Parser`.

In this case, an `AddCommand` will be created and returned, as shown in the object diagram below:

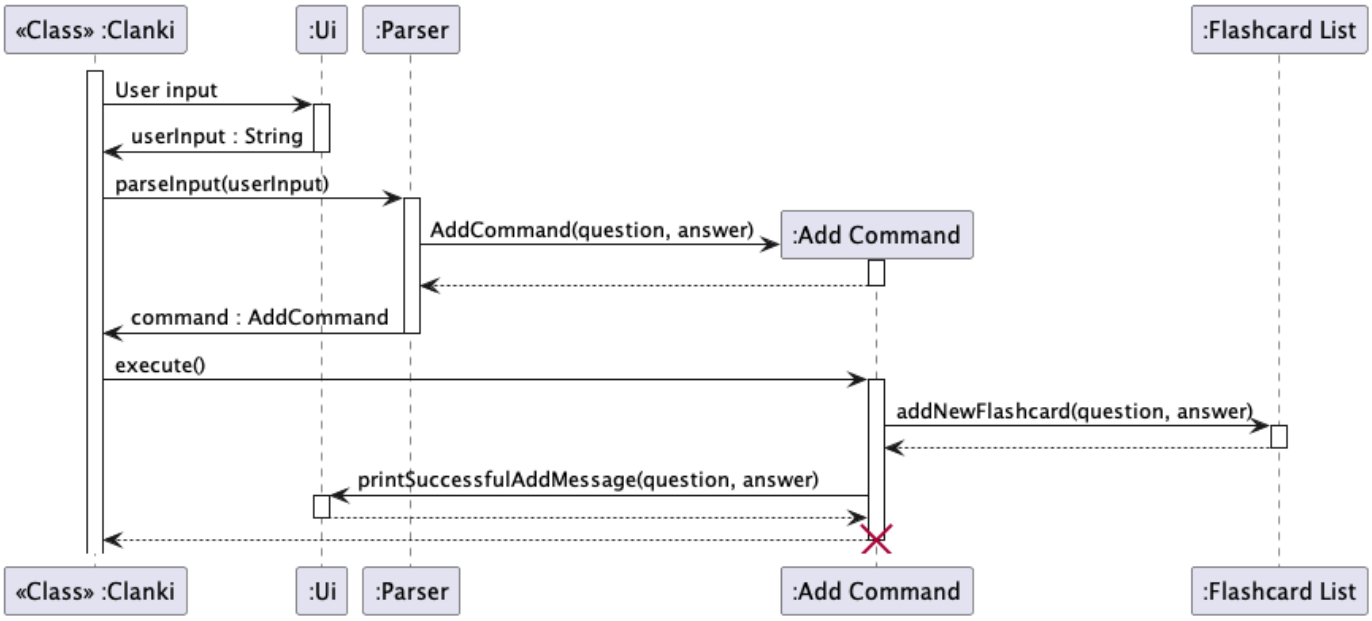


Step 3: The `execute()` function of `AddCommand` will run, calling `addNewFlashcard(questionText, answerText)` of class `FlashcardList` to create and add the new flashcard to the list.

Then it will also call `printSuccessfulAddMessage(questionText, answerText)` of class `Ui` to display text indicating the successful adding function to the user.

At this point, the adding process is completed and the program is ready to take another command.

The following sequence diagram show how the add operation works:



Reason for Current Implementation

Through using `AddCommand` class, which extends `Command` class it increases the level of abstraction as the code can now perform the various commands on a class level.

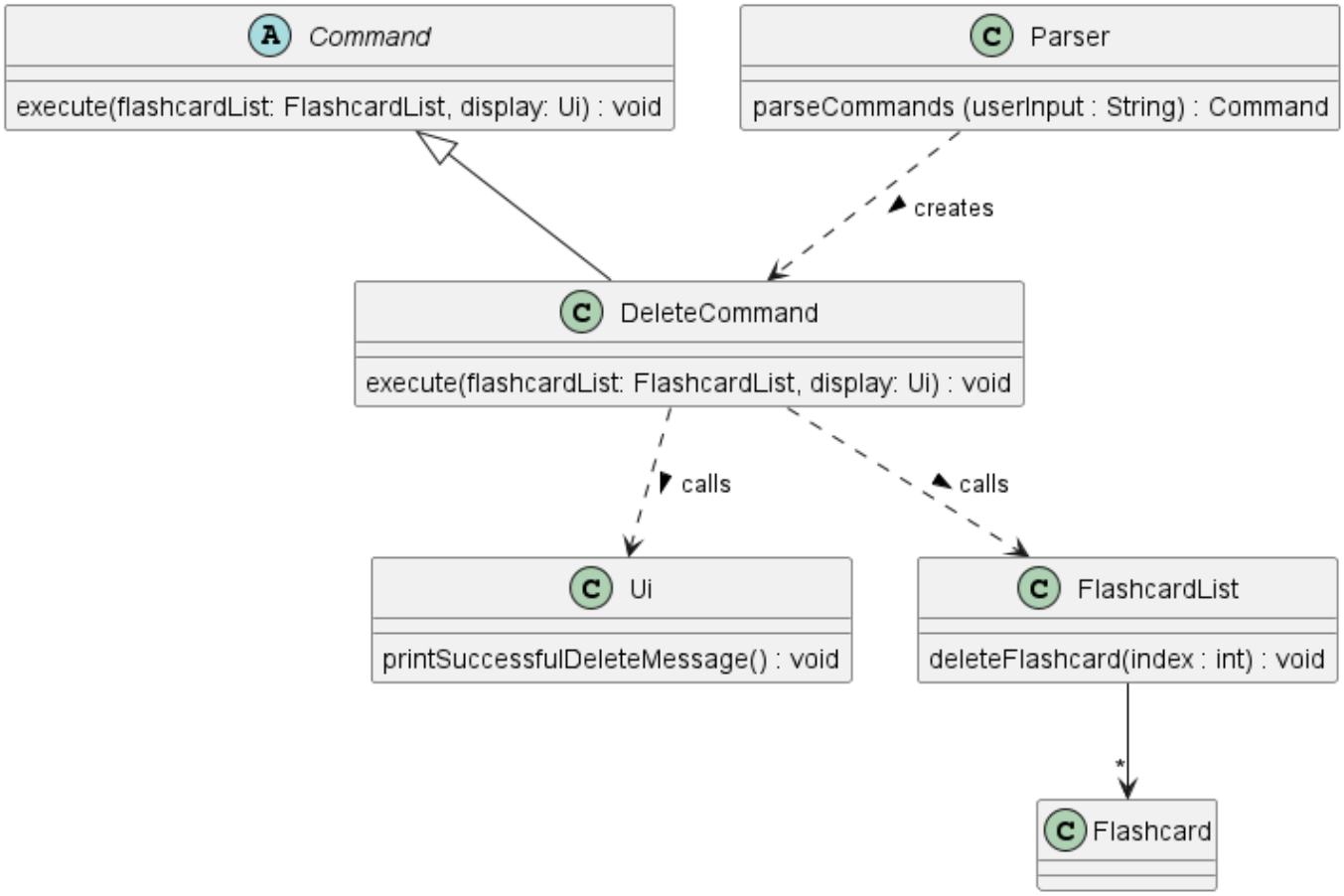
Moreover, since the creating of new `Flashcard` of object and adding of the newly created flashcard are both done in the same class as where the flashcards are stored, this reduces coupling in the program as the `AddCommand` will not have access to the inner structure of `FlashcardList`, which stores the list of flashcards.

Alternative Implementation

- Alternative 1: Have the add command function directly in `FlashcardList`
    - Pros: Easy to implement
    - Cons: Will require another function in another program to differentiate it from other commands
- Alternative 2: Have the constructor of `Flashcard` include adding the card to list of flashcards
    - Pros: Simplifies code
    - Cons: Will cause trouble when temporary flashcard (that need not be stored) are created

Delete Flashcard Feature

The figure below shows a simple class diagram for the Delete Command.



Current Implementation

The current delete flashcard allows the user to remove a flashcard from the list of flashcards, it is implemented through the following steps:

Step 1: The input of user is collected by `getUserCommand()` inside class `Ui`.

Step 2: The input string will be converted into a `Command` object by being passed through `parseCommand(String userInput)` inside `Parser`.

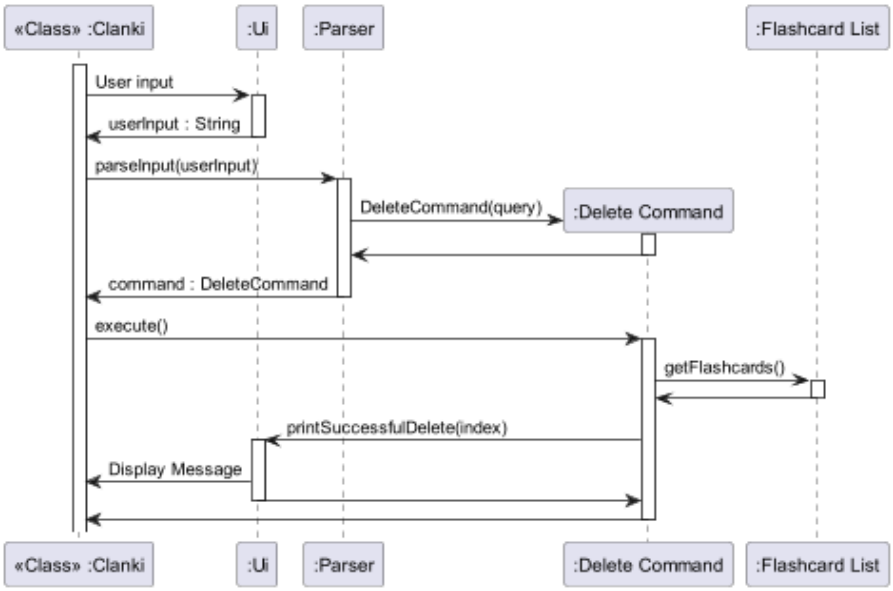
In this case, a `DeleteCommnad` will be created and returned.

Step 3: The `execute()` function of `DeleteCommand` will run, creating a copy of the list of flashcards. Then `findFlashcards(flashcards, query)` is called to find the flashcards with questions matching the query, before calling `printFlashcardList(matchingFlashcards)` to display the found flashcards.

User input is taken to get the index of the flashcard to be removed. `deleteFlashcard` is called from class `flashcardList` to remove the flashcard from the original list of flashcards. Finally `printSuccessfulDelete` is called from class `Ui` to indicate a successful removal of the flashcard.

The deletion process is now completed and the program will await another command.

An overview of how the Delete operation works is shown with the following sequence diagram



Reason for Current Implementation

Through using `DeleteCommand` class, which extends `Command` class it increases the level of abstraction as the code can now perform the various commands on a class level.

In order to minimise the time for users to search for the flashcard to delete, they are able to first search for a sub-list of flashcards with matching questions as the query. This method makes the deletion process simple even if the user does not remember the index of the flashcard.

Alternative Implementation

- Alternative 1: Delete flashcard by index from the start
    - Pros: Easy to implement and simplifies code
    - Cons: Cumbersome to delete if user forgets the flashcard's index and has to search through the whole list of flashcards.

Update Flashcard Feature

Current Implementation

The current update flashcard feature allows users search for a specific flashcard and update the contents of this flashcard. It is implemented through the following steps:

Step 1: The input of user is collected by `getUserCommand()` inside class `Ui`.

Step 2: The input string will be converted into a `Command` object by being passed through `parseCommand(String userInput)` inside `Parser`.

In this case, an `UpdateCommand` will be created and returned.

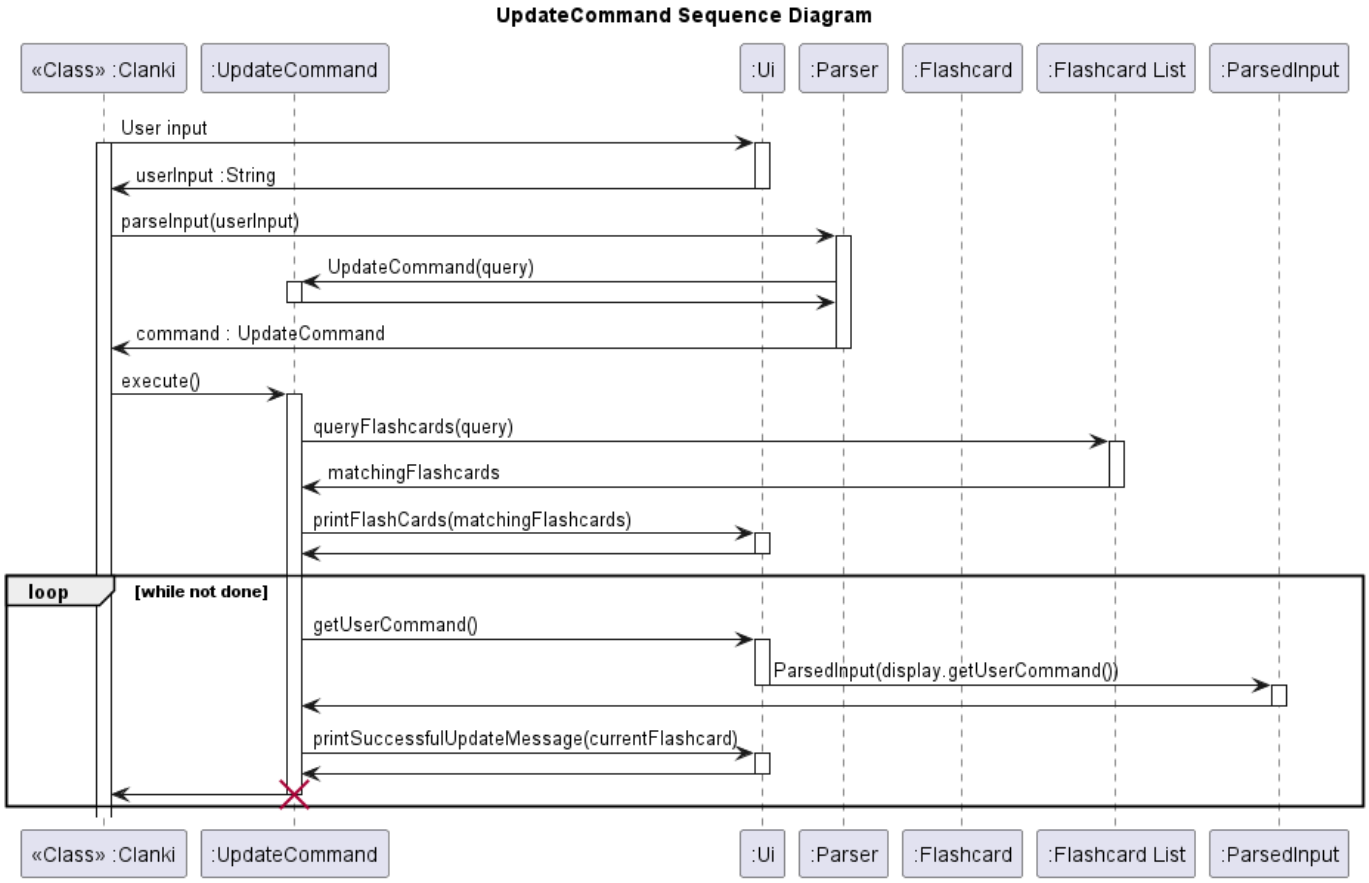
Step 3: The `execute()` function of `UpdateCommand` will run `queryFlashcards(query)` of `FlashcardList`, which will query for flashcards in the current deck that matches the query inside either the question or answer and return an ArrayList of `Flashcard` called `matchingFlashcards`.

Step 4: Then, `printFlashCards(matchingFlashcards)` inside class `Ui` is called, which prints all questions and answers of the list of flashcards, that matches the query, to the console

Step 5: Lastly, `runUpdateFlashcard(display)` is executed. This method prompts the user for input and updates the specified flashcard based on that input.

At this point, the update flashcard process is completed and the program is read to take another command.

An overview of how the Update command works is shown with the following sequence diagram



Reason for Current Implementation

Implementing the update flashcard in an `UpdateCommand` class makes it easier during the debugging process related to update flashcard feature alone as most of the methods and attributes are within this `UpdateCommand` class.

Furthermore, the UpdateCommand has a dedicated function, runUpdateFlashcard(display), which handles the updating of the flashcard. This helps to ensure that the code remains organized and easy to read, with the updating process separated from other code.

Alternative Implementation

- Alternative 1: Instead of creating a new arrayList `matchingFlashcards` that store flashcards containing the `query` and then printing the list of flashcards, directly print the flashcards when there is a match with the query
    - Pros: Easier to implement

    - Cons: Harder to track the total number of flashcards that has `query` and will need to have another way to track the index of the matching flashcards. it will also be more confusing as the index of the user input is not aligned with the index of the arrayList that contains all the flashcards

- Alternative 2: An alternative implementation could be to have the updating of the flashcard handled directly in FlashcardList.
    - Pros: Simple implementation and no need for another function in another program to differentiate it from other commands
    - Cons: May lead to increased coupling in the program, as the `UpdateCommand` will have access to the inner structure of `FlashcardList` and this may make the code more difficult to read and debug, as the updating process will be combined with other code

Review Flashcard Feature

Current Implementation

The current review flashcard allows the user to review all the flashcards that are due today or before, it is implemented through the following steps:

Step 1: The input of user is collected by `getUserCommand()` inside class `Ui`.

Step 2: The input string will be converted into a `Command` object by being passed through `parseCommand(String userInput)` inside `Paser`.

In this case, an `ReviewCommand` will be created and returned.

Step 3: The `execute()` function of `ReviewCommand` will run, calling `getFlashCards()` of class `FlashcardList` to get the list of the flashcards.

Then it will iterate through the `FlashcardList` and call the function `isDueBeforeToday()` of class `Flashcard` to check if the flashcard is due by today.

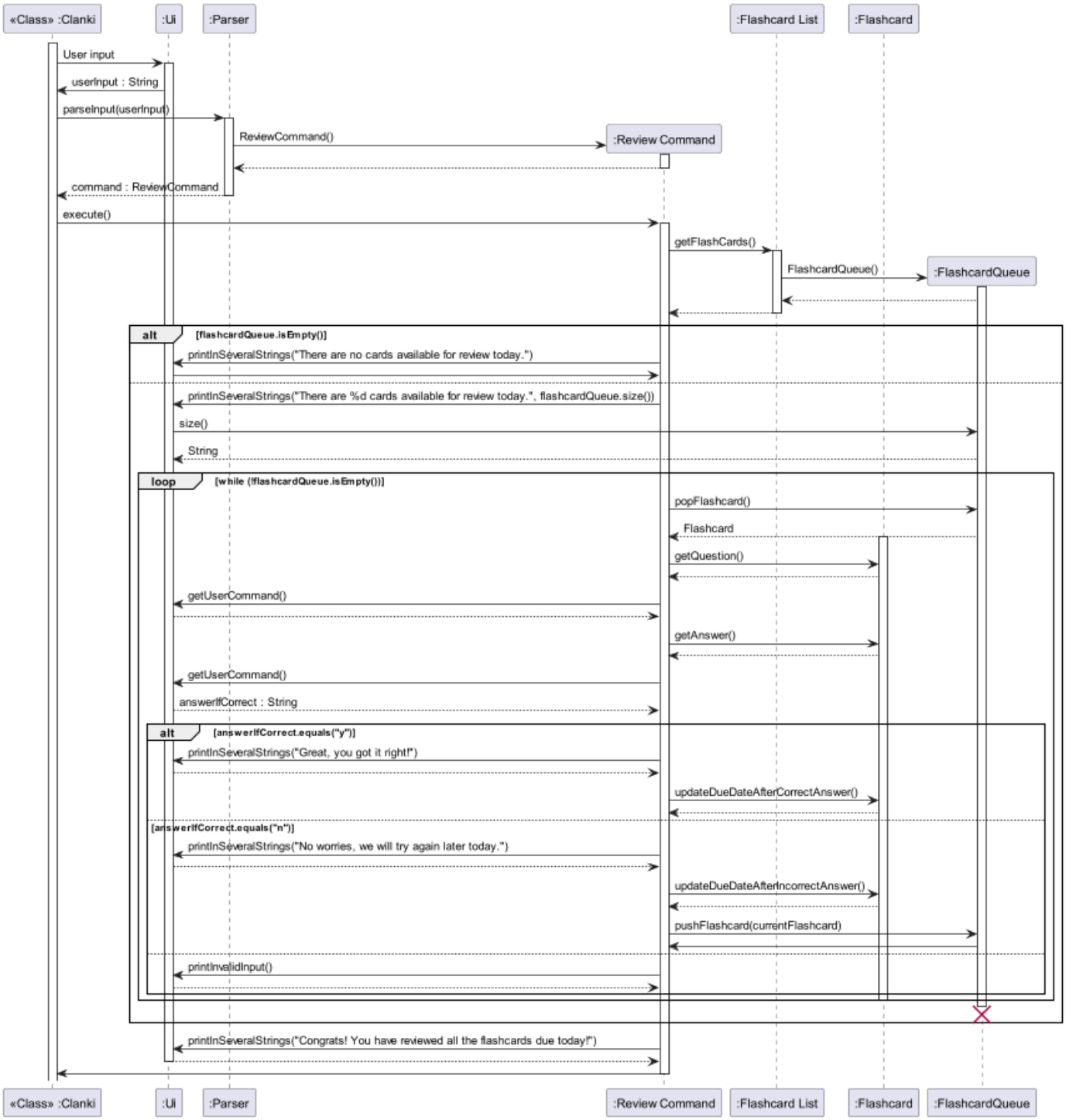
If the flashcard is due by today, `reviewCurrentFlashcard(Ui display, Flashcard flashcard)` of class `ReviewCommand` will be called to review the card.

First, the `Ui` will display the question of the current card by calling the `getQuestion()` method of class `Flashcard`, and ask user if user is ready to view the answer. After user enters any keyboard input, the answer of the current card will be shown by calling the `getAnswer()` method of class `Flashcard`, and `Ui` will ask the user if he/she has got the card correct. If the user inputs “y”, then the current `Flashcard` is considered to be cleared and `updateDueDateAfterCorrectAnswer()` of `Flashcard` will be called to update its `dueDate`. Then Clanki will let user review the next `Flashcard`. If the user inputs “n”, then the card is considered to be not cleared and `updateDueDateAfterIncorrectAnswer()` will be called to update its `dueDate`. Then Clanki will let user review the next `Flashcard`. This process will repeat until all the `Flashcards` in the `FlashcardList` are iterated.

After the whole `FlashcardList` has been iterated through, a message congratulating the user that he/she has completed the reviewing task will be displayed.

At this point, the reviewing process is completed and the program is ready to take another command.

The following sequence diagram show how the review operation work:



Reason for Current Implementation

Through using `ReivewCommand` class, which extends `Command` class it increases the level of abstraction as the code can now perform the various commands on a class level.

Moreover, `ReviewCommand` only has access to the public methods of `FlashcardList` and `Flashcard`, this reduces coupling in the program as the `ReviewCommand` will not have access to the

inner structure of `FlashcardList` and `Flashcard`.

Alternative Implementation

- Alternative 1: Have the review command function directly in `FlashcardList`
  - Pros: Easy to implement
  - Cons: Will require another function in another program to differentiate it from other commands
- Alternative 2: After entering the `ReviewCommand`, go back to `Clanki.run()` and take further commands for review process
  - Pros: Simplifies code in `ReviewCommand`
  - Cons: Will have to pass around a lot of parameters and variables

Parser

Current Implementation

The parser mostly relies on the `ParsedInput` class, which can parse any user provided string input in the format of Windows command prompt commands (`command body /opt-key opt-value`).

`ParsedInput`

Initiated with a string `input`, it splits the input to sections that are of use. From there it splits each section further to a "title" (denoted with `=` below) and a "body" (denoted with `-` below) part.

```
command blah blah /opt1 hello /opt2 world blah bleh
|    Part 1     |  | Part 2 |  |      Part 3      |
|=====| |-------|  |==| |---|  |==| |-------------|
```

Then these small subparts are grouped together to a format where the command part of the command, the body part and the options can be retrieved programmatically.

The command and body can be read with `getCommand()` and `getBody()` respectively. `getCommand()` is guaranteed to be non-null.

The options can be read with `getOptionByName(optionKeyName)`. The reason we don't have specific `getDate` or `getQuestion` command is because we don't know what the user will input and what options we will require for each command. So depending on the command, we retrieve the option accordingly with e.g.

```
"command blah blah /opt1 hello /opt2 world blah bleh"
        getOptionByName("opt2") // -> "world blah bleh"
        getOptionByName("opt3") // -> null
```

`Parser`

This is now just a matter of wrapping `ParsedInput` with suitable error handling and logic such that each command will be used to initiate a corresponding command class (e.g. `AddCommand`), while errors are handled gracefully.

Reason for Current Implementation

We need an intuitive, safe and declarative way to parse the user input. Alternative implementations that can only parse specific commands with specific options are more imperative, less readable, less maintainable and overall just a pain to handle. That's why the two classes are here.

Alternative Implementation

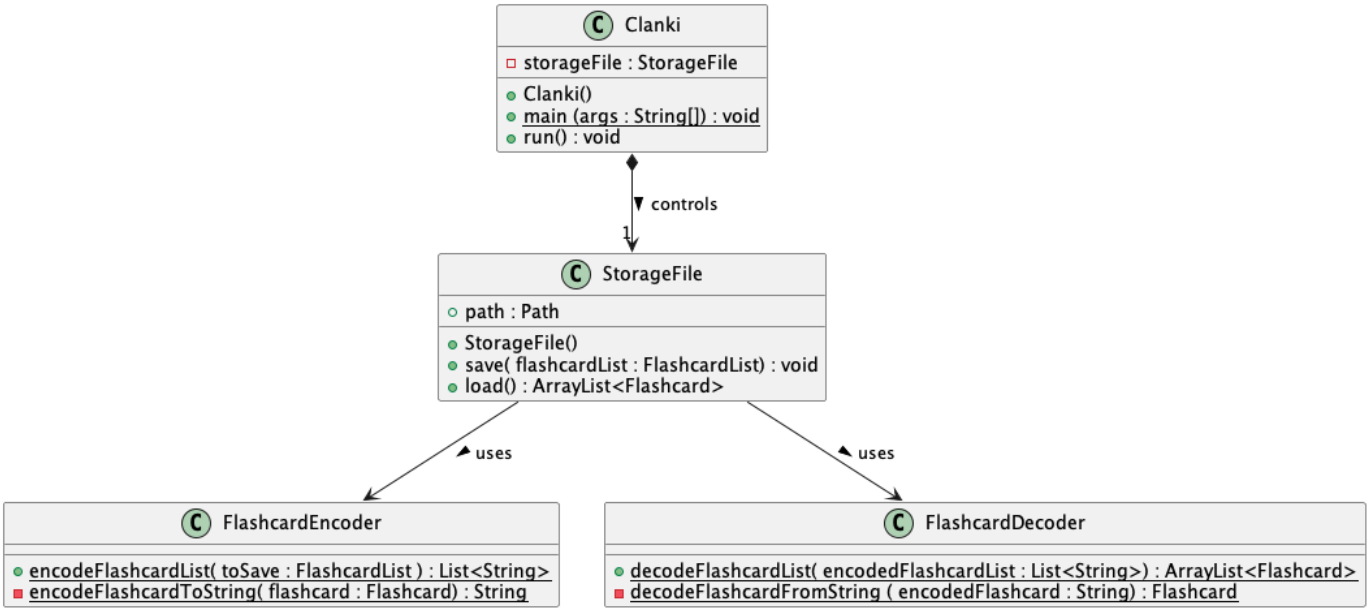
No.

Storage Feature

Current Implementation

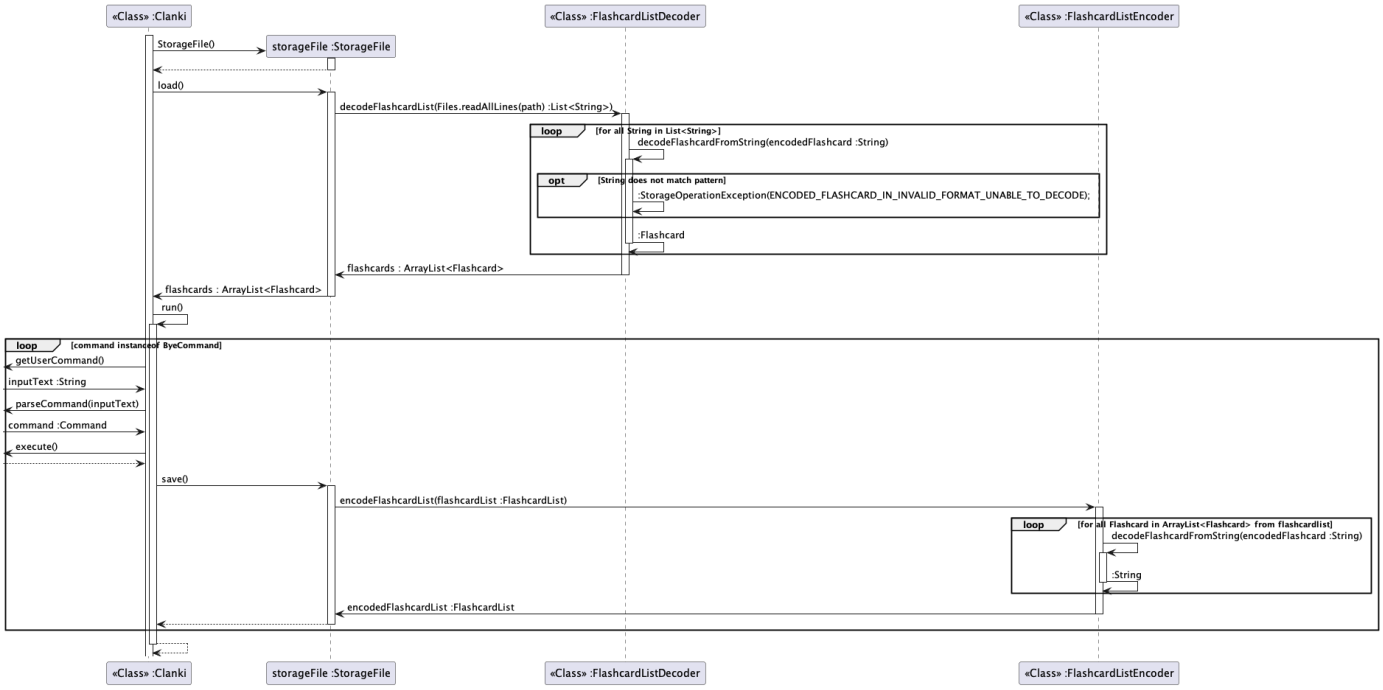
The current storage feature triggers after every execution of command, updating the flashcardList.txt file to be the came as what is stored in the `FlashcardList` object.

The entire feature consist of 3 parts, as shown in the class diagram below:



1. `FlashcardListEncoder` : takes the list of flashcards from `FlashcardList` and convert them to a list of strings, with heading to indicate the start of the question, answer and deadline portion of a flashcard.
2. `FlashcardListDecoder` : takes a list of string in specific format (as defined by `FLASHCARD_ARGS_FORMAT`) and decodes the string into an arrayList of flashcards, discarding any string of incorrect format.
3. `StorageFile` : uses the encoder or decoder to save or load the current state into or from a text file.

The following sequence diagram show how the add operation works:



Reason for Current Implementation

By separating the decoder and encoder as separate classes, it allows the code for the storage system to be more readable by others, allowing them to identify and find the chunk of code for each function more easily, and possibly reuse the functions if they deem necessary in future versions.

Alternative Implementation

- Alternative 1: Have all functions in one `Storage` class
    - Pros: Exceptions can be handled in the same place
    - Cons: Will cause the code be less organised and readable

Product Scope

Target User Profile

Students learning subjects that require a lot of memorisation (history, a new language, etc.)

Value Proposition

This application help users to better remember key points in their upcomming tests by providing them a platform to read through and practice answering those key learning points.

User Stories

Version	As a ...	I want to ...	So that I can ...
v1.0	user	add a card to my flashcard collection	study it later on
v1.0	user	delete any of my cards	prevent getting asked to review that card later when I am confident I have truly memorised the card
v1.0	user	review the cards that are due today	remember them better
v1.0	user	make changes to the q/a any cards I want	keep the info there always updated with what I want myself to memorise
v1.0	user seeking efficiency	review the cards at an appropriate pace that is most efficient for memorisation	not waste time reviewing when I still remember the cards well
v2.0	new user	see usage instructions	refer to them when I forget how to use the application
v2.0	busy user	store the cards somewhere	revisit them next time I open the app
v2.0	organised user	view a list of all currently stored flashcards	know what are the things I need to remember