Liao Li Xin - Project Portfolio

PROJECT: FlashSpeed

Overview

FlashSpeed is a text-based flashcard application specifically designed for university students who are learning a foreign language. FlashSpeed was created to allow students to be able to study and revise foreign vocabulary on the fly.

If you are currently learning a new language or interested to do so, FlashSpeed will be yor best companion and helper along the learning journey.

Summary of contributions

- **Major enhancement 1**: added the **play view** of the app.
  - What it does: allows users to start a new game session to test their memory of the vocabulary.
  - Justification: This feature is one of the essence of the app and it improves the product significantly in terms of functionality because users not only can store and memorize vocabulary, but also can test their memory by playing a game.
  - Highlights: This enhancement affects other commands in different view since some commands can only be executed in one view. It requires more validations to be introduced to other commands as well as changes to the Model component to ensure isolation in different views.
- **Major enhancement 2**: implemented the **model manager** class of the app.
  - What it does: enables the Model component to handle all possible commands to be made by the users.
  - Justification: This enhancement allows every operation to be executed at the Model level and increases the maintainability and scalability of the code since all operations are handled at one same place.
  - Highlight: This enhancement requires an in-depth understanding of the overall architecture of the product and changes to be made to other related classes.
- **Minor enhancement**: added create/remove/rename deck command and add/delete card commands as the core functionality of the app.
- **Code contributed**: [Functional and Test code]
- **Other contributions**:
  - Enhancements to existing features:

    - added stop command to allow users to end the game session halfway (Pull request #173)
    - fixed bugs during testing (Pull requests #198, #276, #286)
  - Documentation:
    - Refine Play View section of User Guide to be more user friendly. (Pull request #255)
  - Community:
    - Reported bugs and provided suggestions for other teams in the class (examples: #1 #2 #3)

Contributions to the User Guide

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

This is what you will see when you test yourself. Strengthen your memory by frequently reviewing your cards! When playing a deck, each card will initially only show its front face to allow you to recall its associated back face.

Flipping a card : `flip`

Format: `flip`

Are you ready to reveal the back face of the card? Let's flip it to check if your memory is spot on. To flip a card, simply type `flip` into the input box and press `Enter`.

Answering : `yes`/`no`

Format: `yes` or `no`

Were you able to recall the correct back value?
If you could, type `yes` into the input box and press `Enter`. Congratulations!
If you could not or your guess was incorrect, type `no` into the input box and press `Enter`. Don't give up!

Stopping a session: `stop`

Format: `stop`

A play session will end automatically when there are no more cards to review. However, you can also stop an ongoing session immediately by typing `stop` into the input box and pressing `Enter`. But of course, try your best and don't use this too often!

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.

Model component

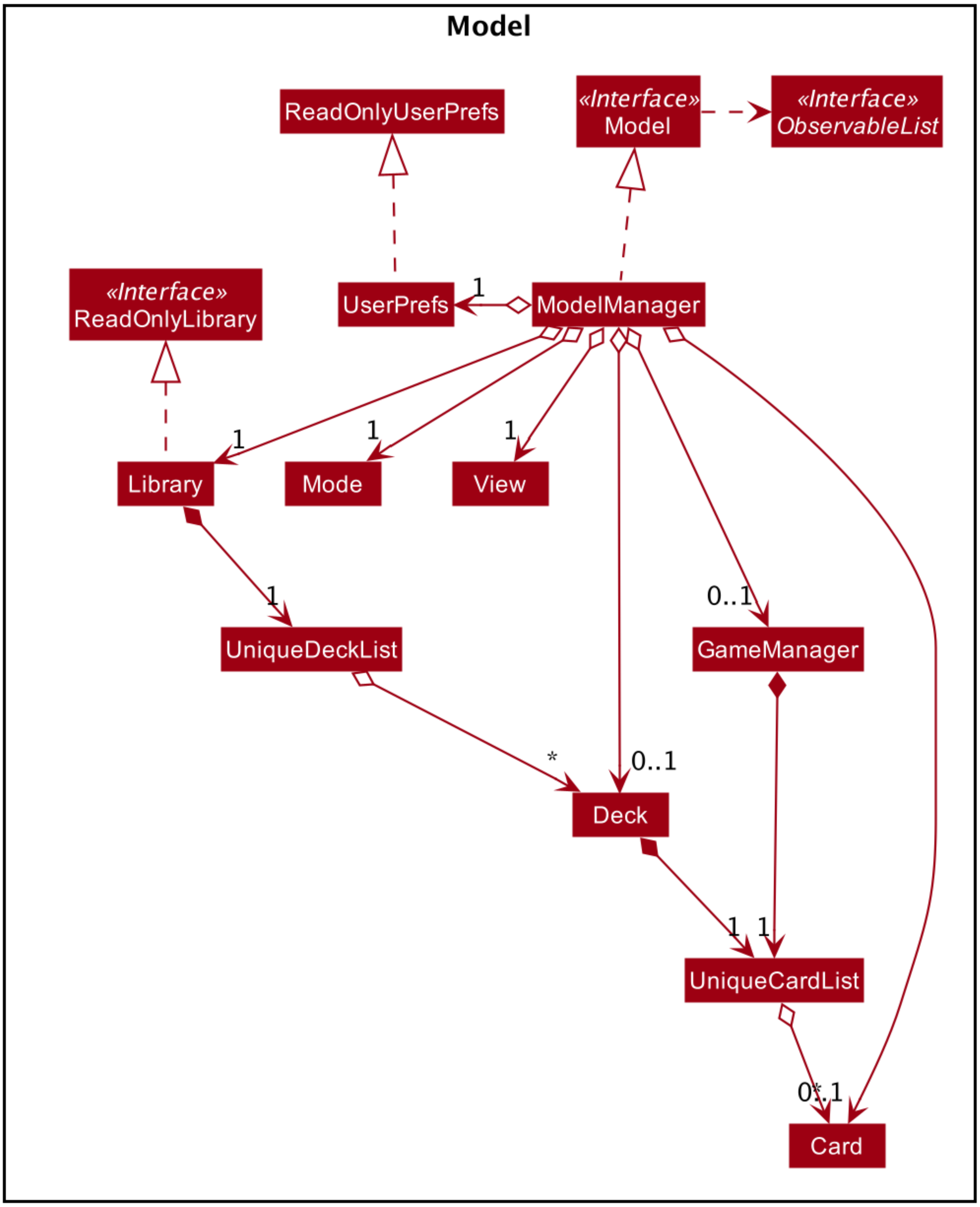

Figure 1. Structure of the Model Component

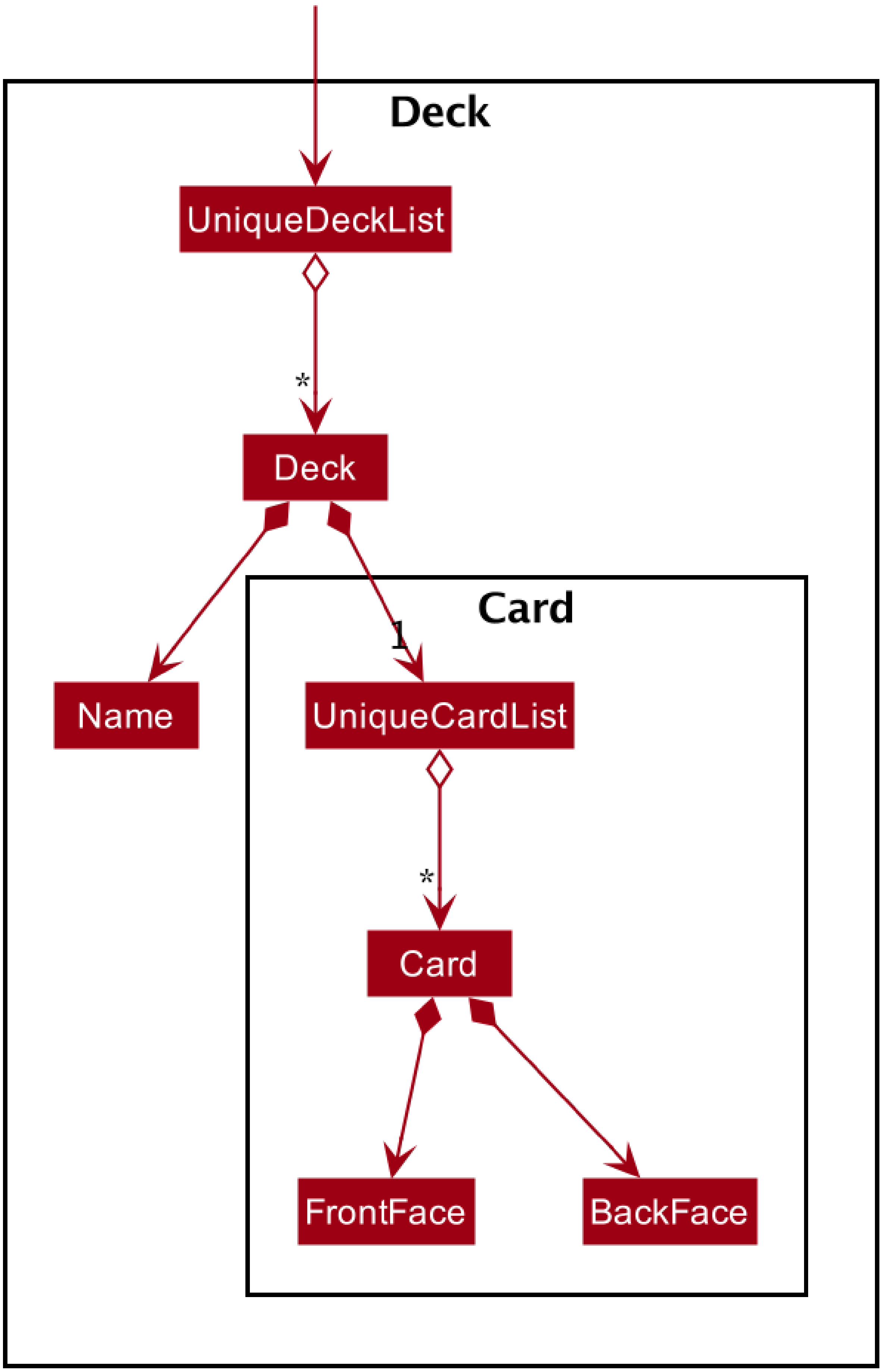
Deck
UniqueDeckList
*
Deck
Card
1
Name
UniqueCardList
*
Card
FrontFace
BackFace

Figure 2. Structure of the Deck Component within the Model Component

API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user’s preferences.
- stores the Library’s current state and data.
- stores and manipulates a `GameManager` object that represents one game session.
- stores and manipulates a `Deck` object that represents the deck that the user is viewing when user is in deck view.
- stores and manipulates a `Card` object that represents the card that the user is playing with when user is in play view.
- stores and manipulates `View` object that represents the view that the user is currently in.
- exposes an unmodifiable `ObservableList<Deck>` that can be 'observed' e.g. the UI can be bound to this list so that the UI automatically updates when the data in the list change.
- does not depend on any of the other three components.

Starting a play session

Current Implementation

The `play` command creates a new session to play with a specific deck.

Accepted syntax: `play INDEX`

The play command changes the mode of the application to `PLAY` mode and creates a new session with the Deck at the given `INDEX`. The value of the `FRONT` of the selected `Deck` will be displayed to the user.

Validation and extraction of input in parser

The first validation of the `play` command is performed in `PlayCommandParser#parse()`. The validation only checks that the `play` command has the correct format as the `INDEX` argument is given by the user and it is performed on the login level.

In `PlayCommandParser#parse()`, the `INDEX` of the deck is extracted from the arguments in the `play` command. The `INDEX` is converted to an Index object. An `PlayCommand` object is then constructed with the Index.

Execution of Command object

After the object of the `PlayCommand' is constructed, `PlayCommand#execute()` will be executed and the second validation of the `play` command is performed. This validation firstly checks if the given `INDEX` argument is a non-negative integer and is within the number of cards in the selected Deck. Then the validation checks if there is any card currently in the selected deck by checking if the `FRONT` face and `BACK` face of the card returned by `ModelManager#play()` are both empty.

A valid `play` command will change the `MODE` of the `ModelManager` to `PLAY` mode and a `GameManager` object will be constructed in `ModelManager`. The first card of the selected deck is obtained using `deck#asUnmodifiableObservableList().get(0)` and returned to UI. The `FRONT` face of the first card will be displayed to the user.

The following sequence diagram shows how the `play` operation works.

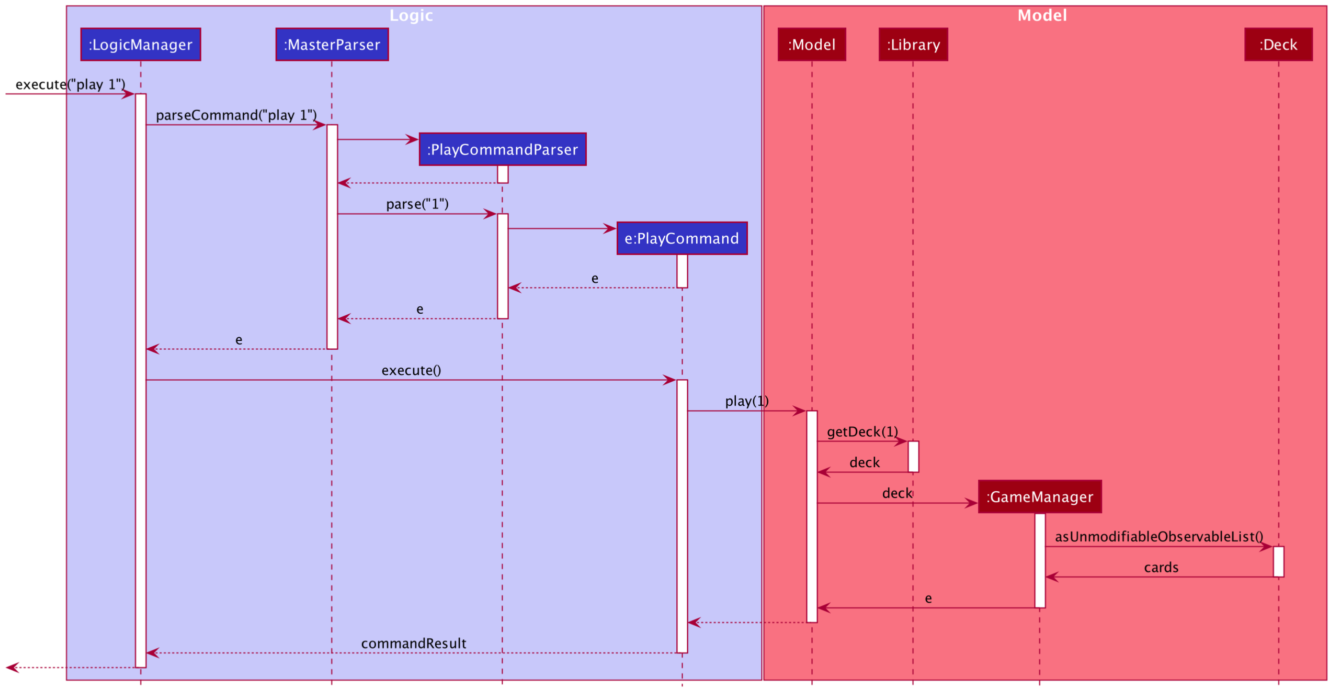

Figure 3. Interactions inside Logic and Model components when `play 1` is executed

Flipping a Card

Current Implementation

The `flip` command flips a card in the selected deck to view the `BACK` face of the card.

Accepted syntax: `flip`

The `flip` command displays the `BACK` face of the card that the user is currently playing with to the user so that user is able to check if his or her answer is correct.

Validation and extraction of input in parser

No user parameter is required, hence a parser is not needed.

Execution of Command object

An `FlipCommand` object is constructed and `FlipCommand#execute()` is executed. In `FlipCommand#execute()`, validation for the `flip` command is performed. The validation will check if `ModelManager` is in `PLAY` mode using `ModelManager#getMode()`. if `ModelManager` is in `PLAY` mode, then the validation will check if the card has been flipped by checking if the returned `BACK` face of the card is empty since a card can only be flipped once.

After that, `ModelManager#flip()` will be executed. In `ModelManager#flip()`, `GameManager#flip()` will be executed and the `BACK` face of the card is obtained using `GameManager#cards.get(counter).getBackFace()` and returned to `ModelManager`.

A valid `flip` command returns the `BACK` face of the card that the user is currently playing to the UI and displays it to the user.

The following sequence diagram shows how the `flip` operation works.

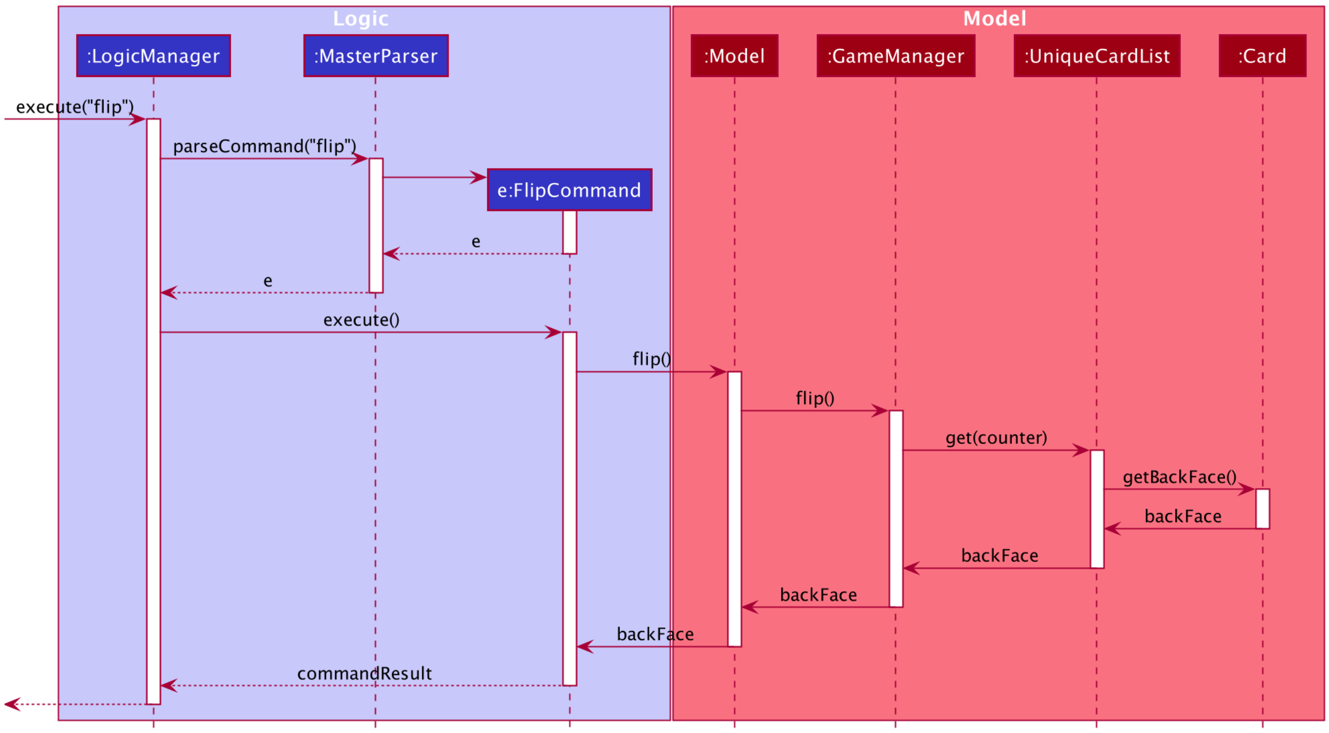

Figure 4. Interactions inside Logic and Model components when `flip` is executed

Answering in a play session

Current Implementation

User answers to the card that he or she is currently playing with using `yes` or `no` command.

Accepted syntax: `yes` or `no`

After flipping the card, users indicates if he or she gets the correct answer by using `yes` and `no` command.

Validation and extraction of input in parser

No user parameter is required, hence a parser is not needed.

Execution of Command object

An `AnswerYesCommand` or `AnswerNoCommand` object is constructed and `AnswerYesCommand#execute()` or `AnswerNoCommand#execute()` is executed accordingly. Validation for the `yes` and `no` command is performed to check if if `ModelManager` is in `PLAY` mode using `ModelManager#getMode()`. if `ModelManager` is in `PLAY` mode, then the validation will check if the card has been flipped using `ModelManager#getGame().isFlipped()` since a card should not have been flipped before user answers to the card.

After that, `ModelManager#answerYes()` or `ModelManager#answerNo()` will be executed accordingly. In `ModelManager#answerYes()` and `ModelManager#answerNo()`, `GameManager#answerYes()` and `GameManager#answerNo()` will be executed accordingly and the next card is obtained using `GameManager#cards.get(counter)` and returned to `ModelManager`. `ModelManager` will check if `ModelManager` will check if the session has ended as the user have run through every card in the deck by checking if the returned card is empty.

A valid `yes` or `no` command returns the next card to the UI and the `FRONT` face of the card is displayed to the user.

The following sequence diagrams show how the `yes` and `no` operation work.

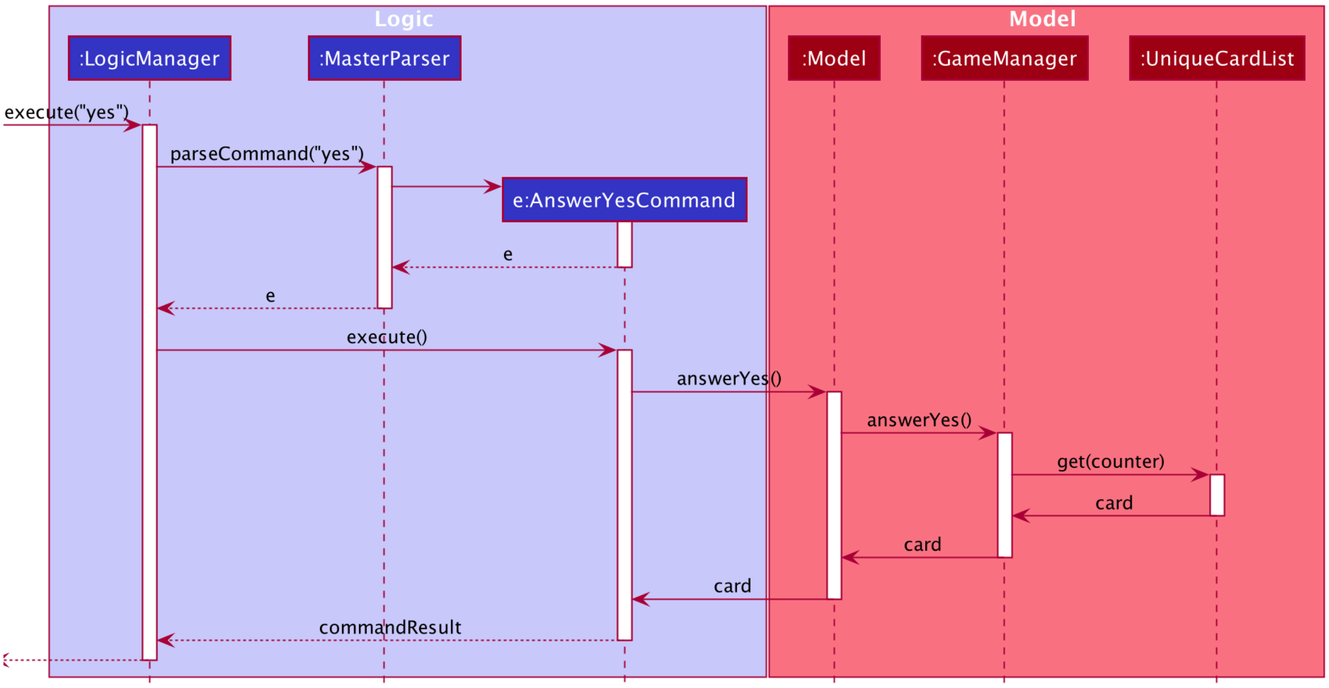

Figure 5. Interactions inside Logic and Model components when `yes` *is executed*

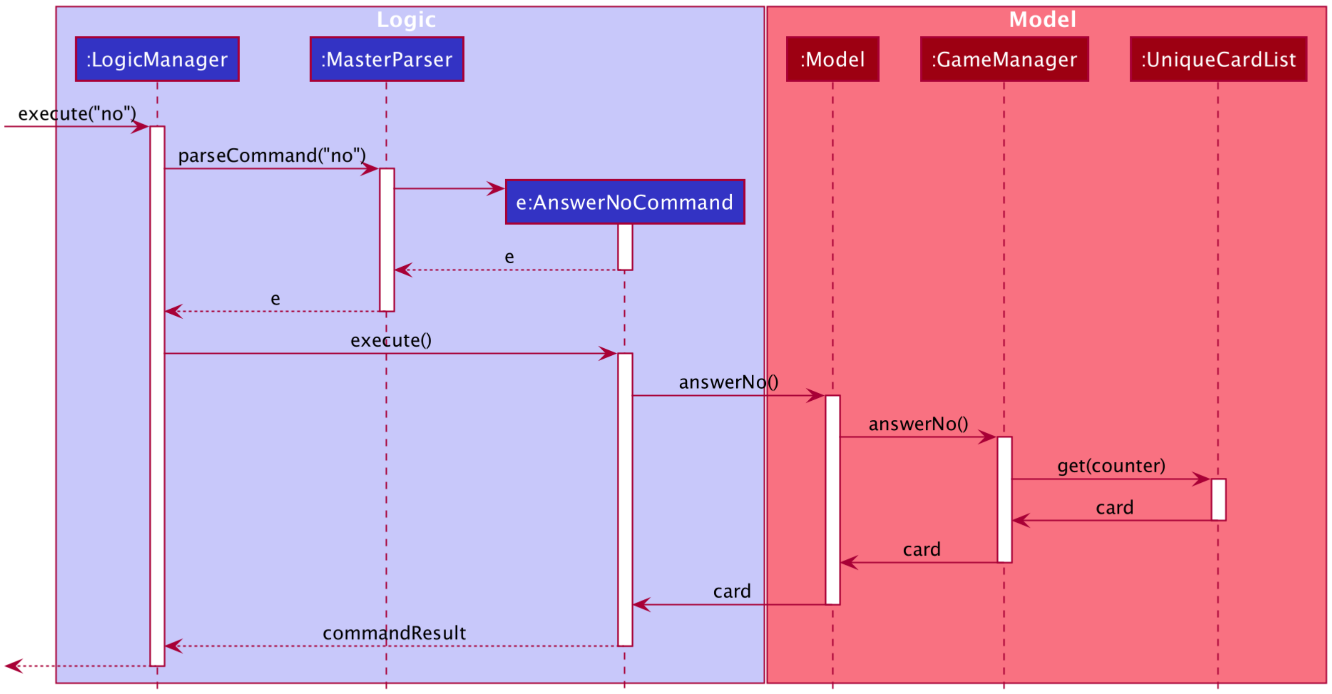

Figure 6. Interactions inside Logic and Model components when `no` *is executed*