CardiBuddy - Developer Guide

By: `Team T10-2`   Since: `Jan 2020`   Licence: `MIT`

1. Setting up

Refer to the guide here.

2. Design

2.1. Architecture

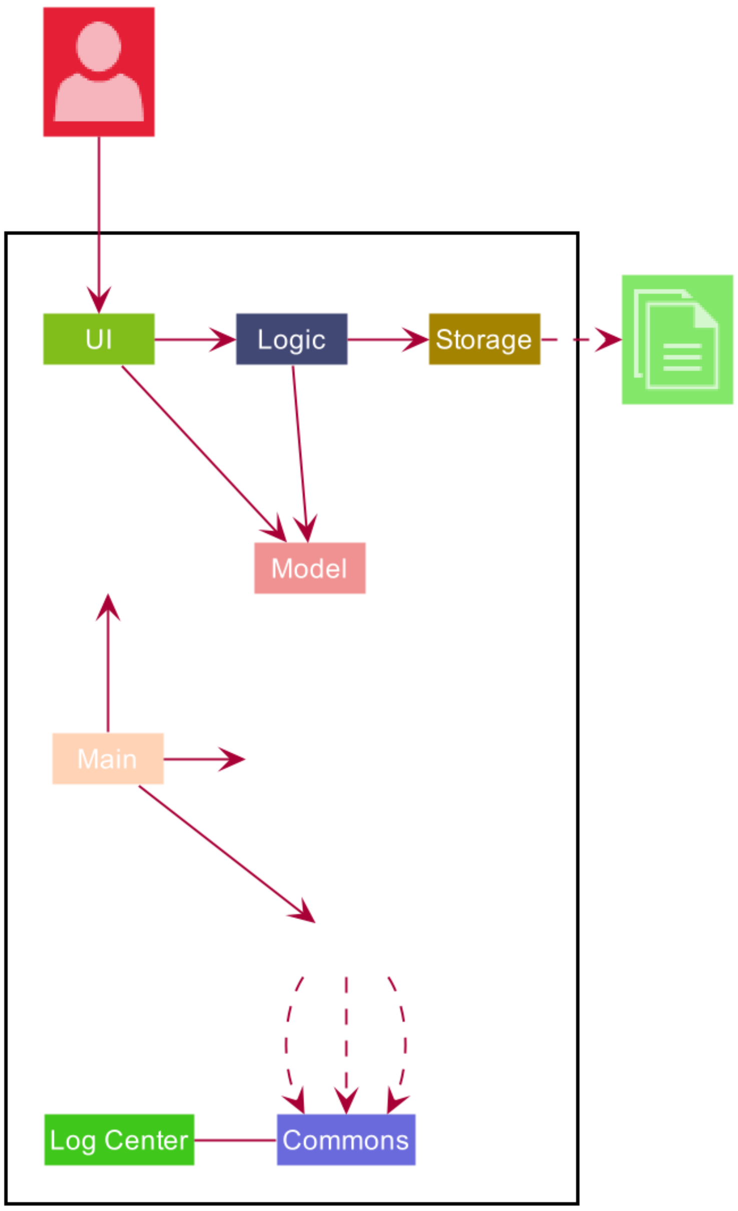


Figure 1. Architecture Diagram

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

TIP	The `.puml` files used to create diagrams in this document can be found in the diagrams folder. Refer to the Using PlantUML guide to learn how to create and edit diagrams.

`Main` has two classes called `Main` and `MainApp`. It is responsible for,

- At app launch: Initializes the components in the correct sequence, and connects them up with each other.
- At shut down: Shuts down the components and invokes cleanup method where necessary.

`Commons` represents a collection of classes used by multiple other components. The following class plays an important role at the architecture level:

- `LogsCenter` : Used by many classes to write log messages to the App's log file.

The rest of the App consists of four components.

- **`UI`**: The UI of the App.
- **`Logic`**: The command executor.
- **`Model`**: Holds the data of the App in-memory.
- **`Storage`**: Reads data from, and writes data to, the hard disk.

Each of the four components

- Defines its *API* in an `interface` with the same name as the Component.
- Exposes its functionality using a `{Component Name}Manager` class.

For example, the `Logic` component (see the class diagram given below) defines it's API in the `Logic.java` interface and exposes its functionality using the `LogicManager.java` class.

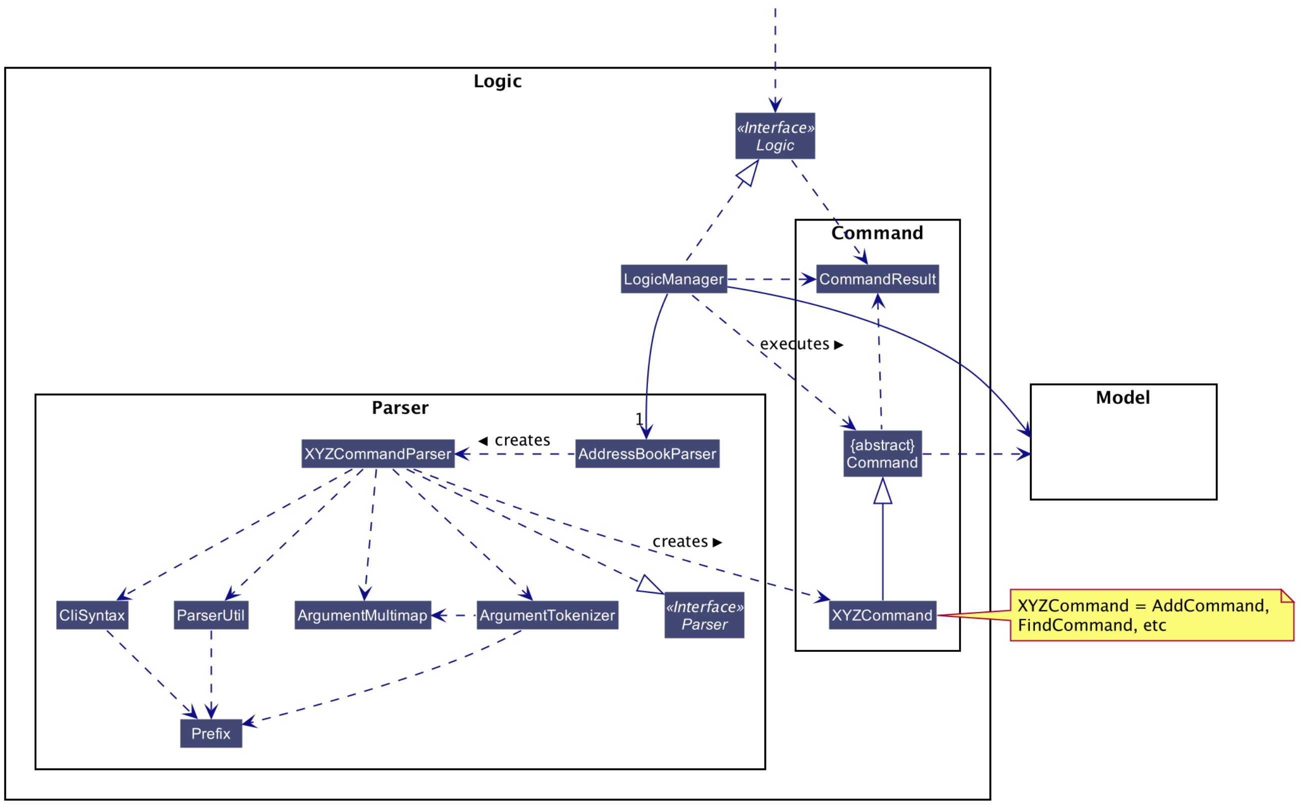


Figure 2. Class Diagram of the Logic Component

How the architecture components interact with each other

The *Sequence Diagram* below shows how the components interact with each other for the scenario where the user issues the command `delete 1`.

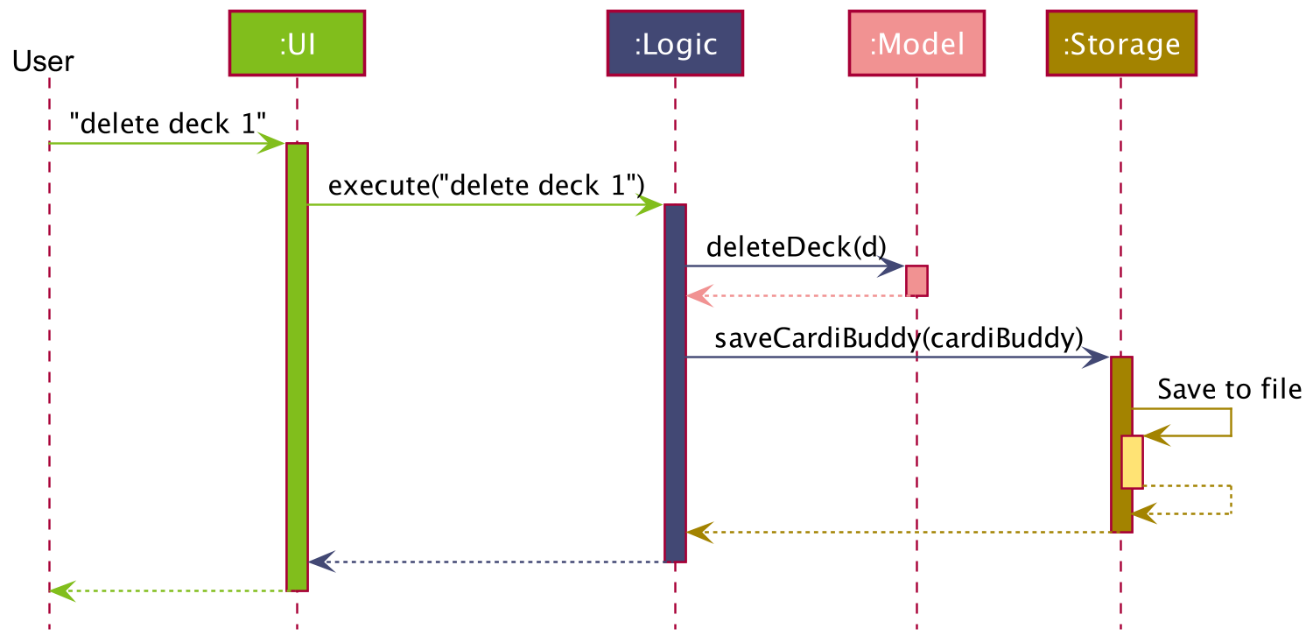


Figure 3. Component interactions for `delete 1` *command*

The sections below give more details of each component.

2.2. UI component

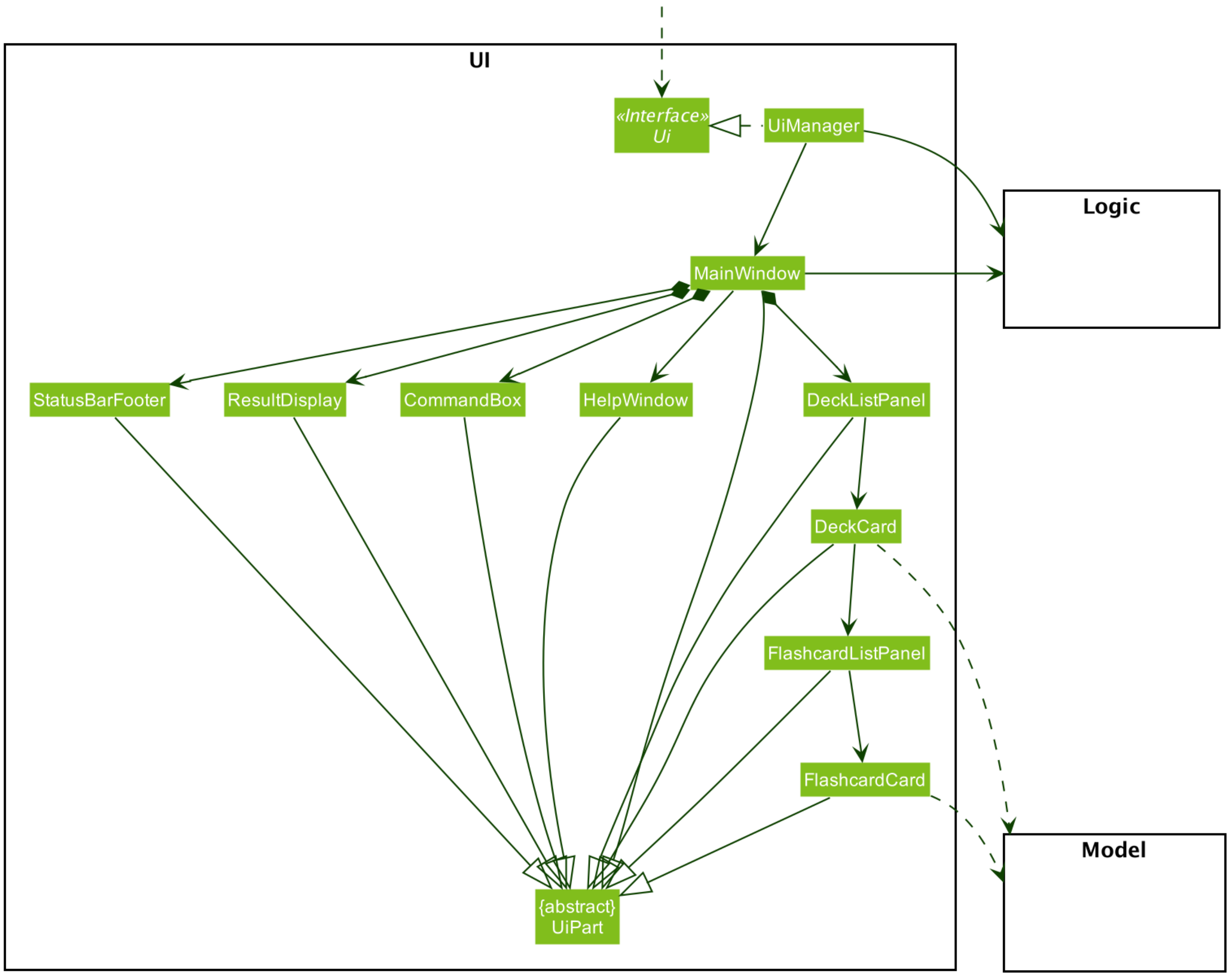


Figure 4. Structure of the UI Component

API : `Ui.java`

The UI consists of a `MainWindow` that is made up of parts e.g.`CommandBox`, `ResultDisplay`, `DeckListPanel`, `FlashcardListPanel`, `StatusBarFooter` etc. All these, including the `MainWindow`, inherit from the abstract `UiPart` class.

The `UI` component uses JavaFx UI framework. The layout of these UI parts are defined in matching `.fxml` files that are in the `src/main/resources/view` folder. For example, the layout of the `MainWindow` is specified in `MainWindow.fxml`

The `UI` component,

- Executes user commands using the `Logic` component.
- Listens for changes to `Model` data so that the UI can be updated with the modified data.

2.3. Logic component

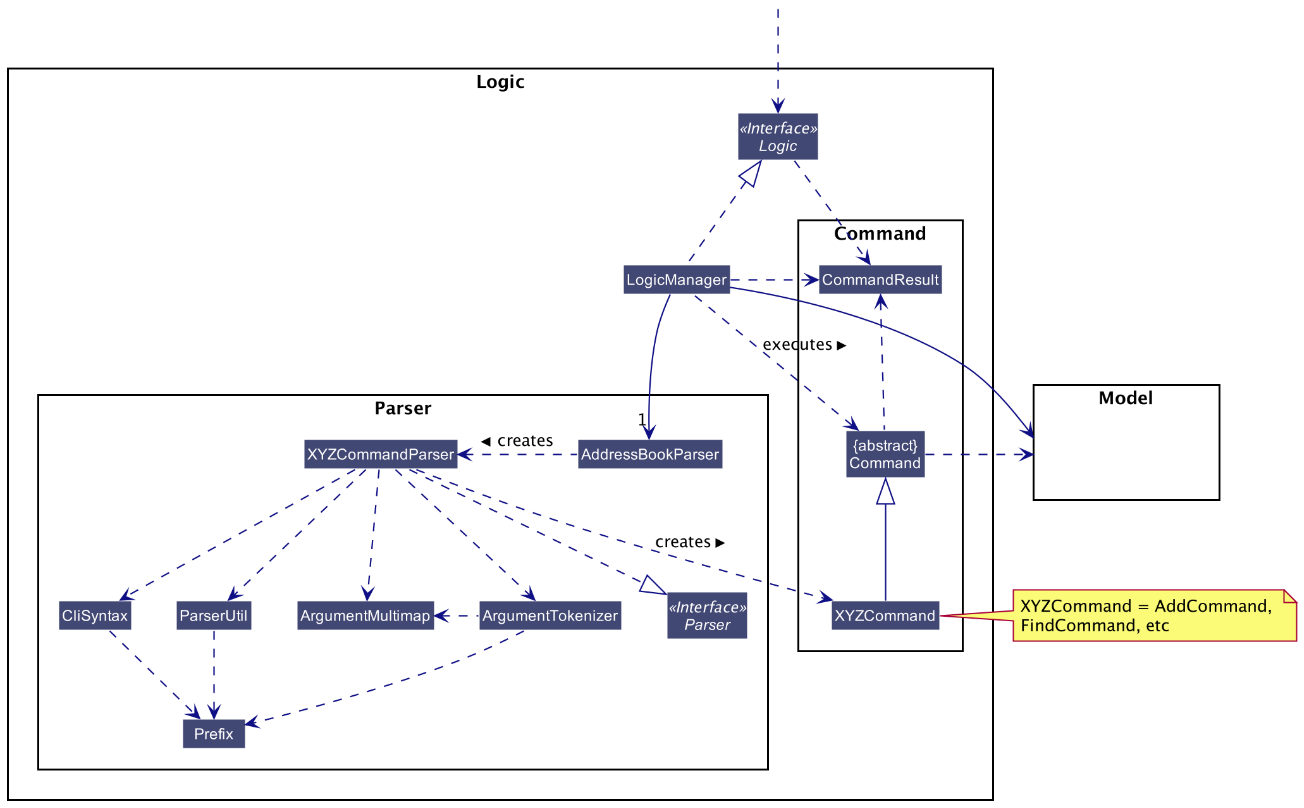


Figure 5. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `CardiBuddyParser` class to parse the user command.
2. This results in a `Command` object which is executed by the `LogicManager`.
3. The command execution can affect the `Model` (e.g. adding a deck).
4. The result of the command execution is encapsulated as a `CommandResult` object which is passed back to the `Ui`.
5. In addition, the `CommandResult` object can also instruct the `Ui` to perform certain actions, such as displaying help to the user.

Given below is the Sequence Diagram for interactions within the `Logic` component for the `execute("delete deck 1")` API call.

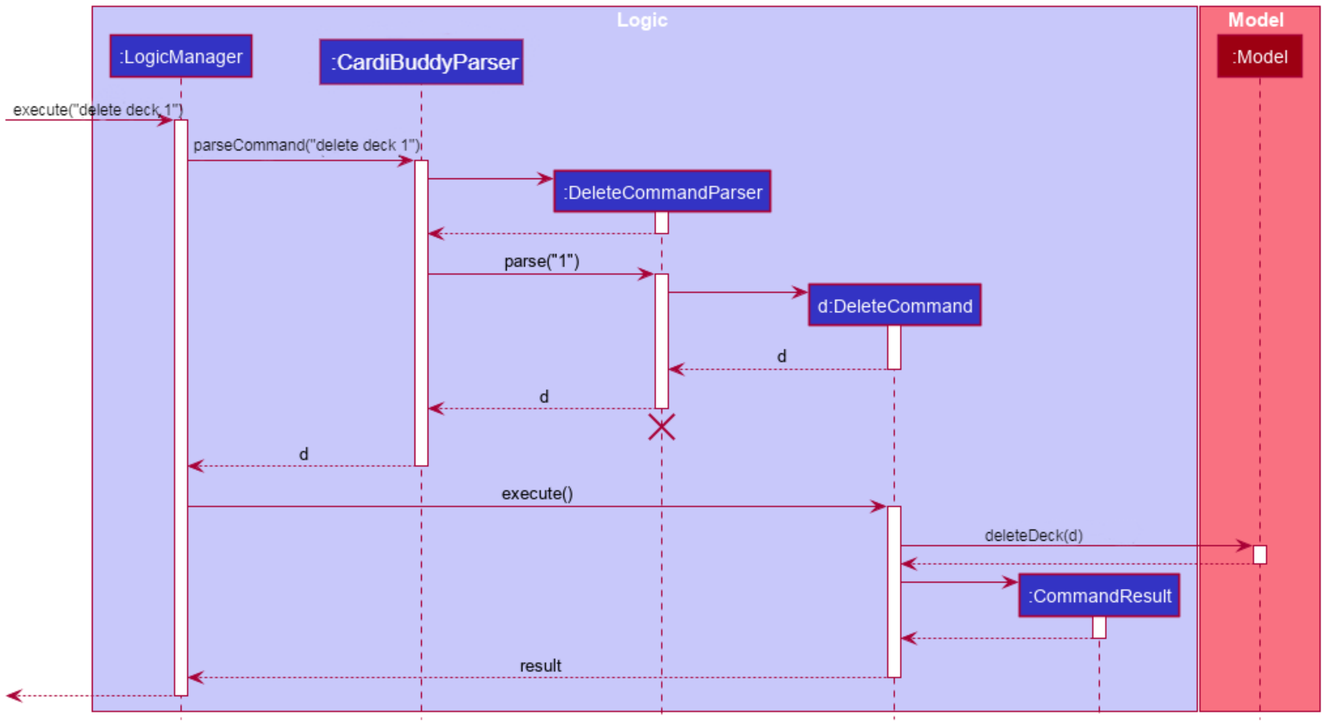


Figure 6. Interactions Inside the Logic Component for the `delete deck 1` Command

NOTE The lifeline for `DeleteCommandParser` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

3. LogicToUiManager

This class serves as a bridge between relevant `Command` classes and the `Ui`, for use cases that do not update the `ObservableList`. It calls specific methods in the `MainWindow` class to modify the display to the user according to the `Command` executed.

The `LogicToUiManager` object is created upon startup in the MainApp class, and there can exist only one instance of it throughout the use of CardiBuddy.

Commands that make use of `LogicToUiManager`:

1. `OpenCommand`
2. `AddCommand`
3. `DeleteDeckCommand`
4. `DeleteCardCommand`
5. `TestCommand`
6. `AnswerCommand`
7. `NextCommand`
8. `QuitCommand`
9. `SkipCommand`
10. `SearchCardCommand`

11. `StatisticsCommand`
12. `ListCommand`

The following sequence diagram illustrates how the `LogicToUiManager` can be used to modify the `MainWindow`, when CardiBuddy needs to display the flashcard question to the user during a `Test Session`. More details on `Test Session` in the section: Test Session.

This diagram is also an extension of the sequence diagram found in that section.

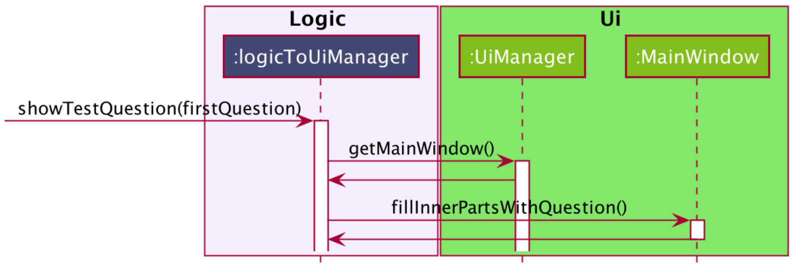


Figure 7. Sequence diagram showing how LogicToUiManager accesses the Ui to display a question during a test session.

3.1. Model component

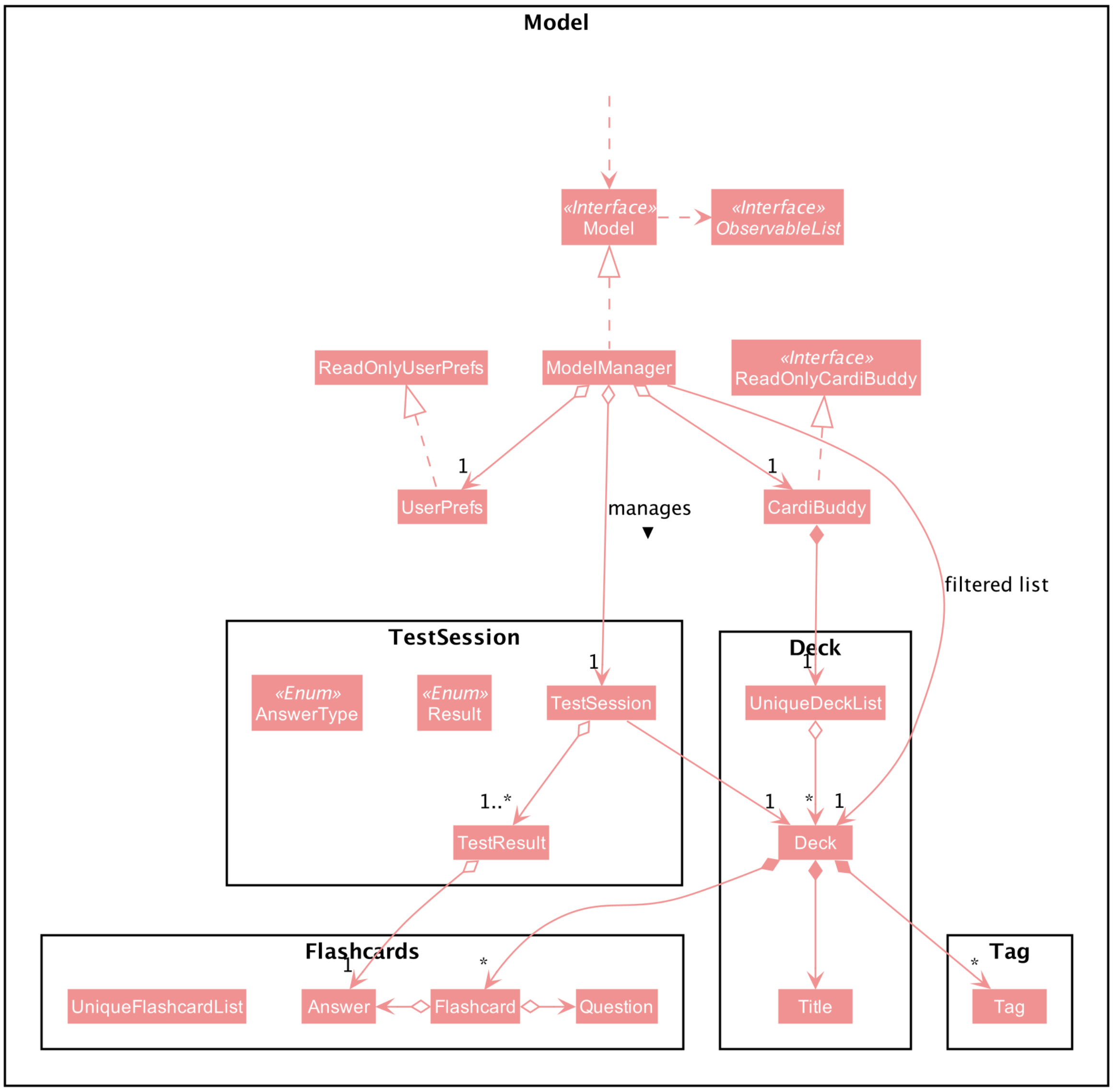


Figure 8. Structure of the Model Component

API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user's preferences.
- stores the CardiBuddy data.
- 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 changes.
- does not depend on any of the other three components.

3.2. Storage component

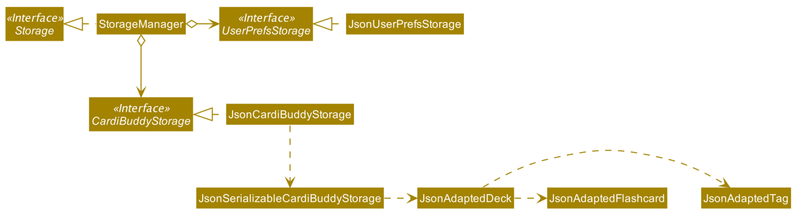


Figure 9. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

- can save `UserPref` objects in json format and read it back.
- can save the CardiBuddy data in json format and read it back.

3.3. Common classes

Classes used by multiple components are in the `cardibuddy.commons` package.

4. Implementation

This section describes some noteworthy details on how certain features are implemented.

4.1. Undo/Redo feature

4.1.1. Design

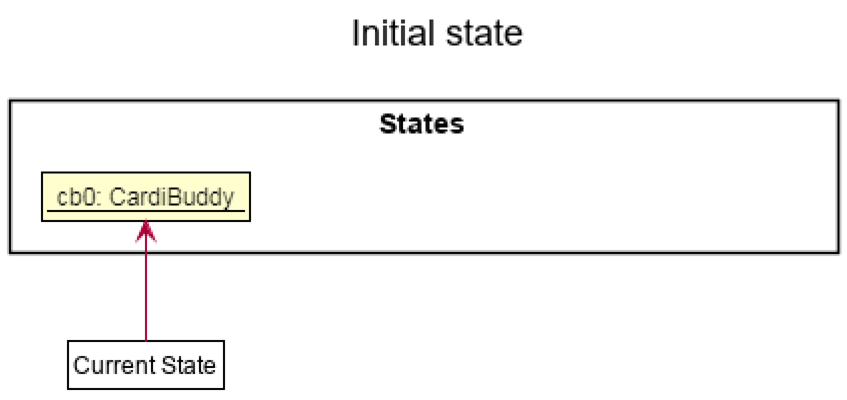
The undo/redo mechanism is facilitated by `VersionedCardiBuddy`. It extends `CardiBuddy` with an undo/redo history, stored internally as an `cardiBuddyStateList` and `currentStatePointer`. Additionally, it implements the following operations:

- `VersionedCardiBuddy#commit()` — Saves the current CardiBuddy state in its history.
- `VersionedCardiBuddy#undo()` — Restores the previous CardiBuddy state from its history.
- `VersionedCardiBuddy#redo()` — Restores a previously undone CardiBuddy state from its history.

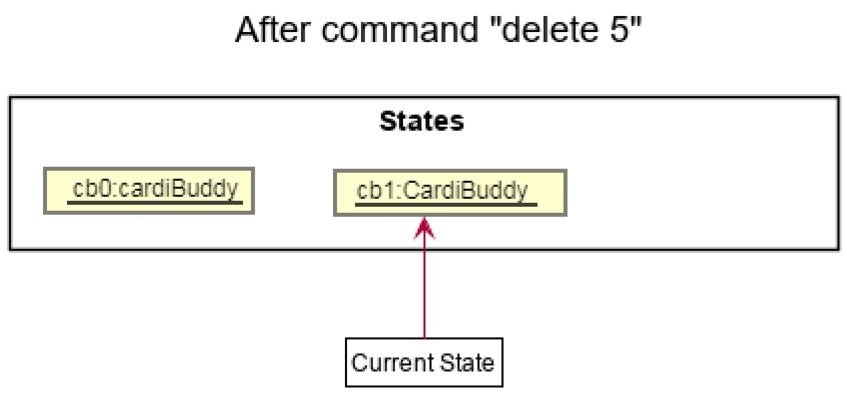
These operations are exposed in the `Model` interface as `Model#commitCardiBuddy()`, `Model#undoCardiBuddy()` and `Model#redoCardiBuddy()` respectively.

Given below is an example usage scenario and how the undo/redo mechanism behaves at each step.

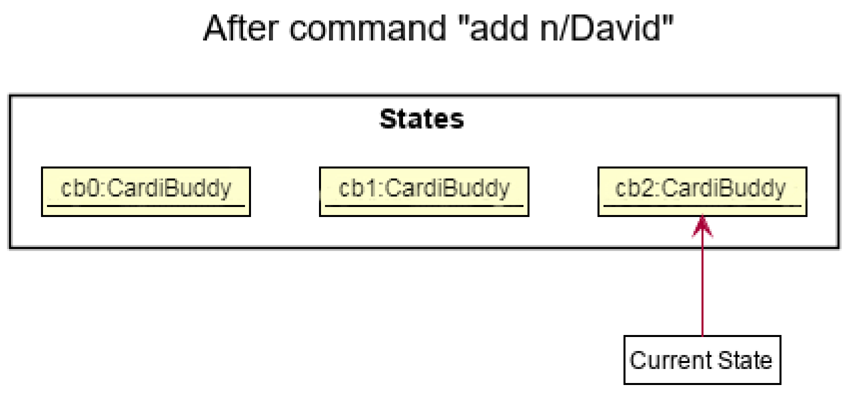
Step 1. The user launches the application for the first time. The `VersionedCardiBuddy` will be initialized with the initial CardiBuddy state, and the `currentStatePointer` pointing to that single CardiBuddy state.



Step 2. The user executes `delete 5` command to delete the 5th deck in the address book. The `delete` command calls `Model#commitCardiBuddy()`, causing the modified state of CardiBuddy, after the `delete 5` command executes, to be saved in the `cardiBuddyStateList`, and the `currentStatePointer` is shifted to the newly inserted CardiBuddy state.

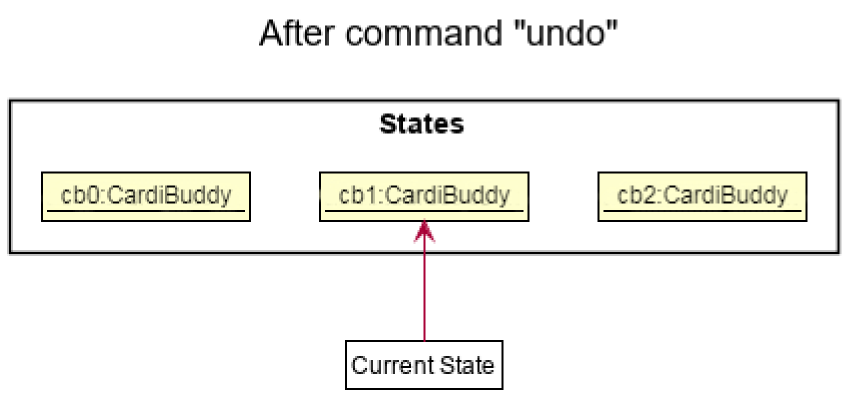


Step 3. The user executes `add d/cs2103T …` to add a new deck. The `add` command also calls `Model#commitCardiBuddy()`, causing another modified CardiBuddy state to be saved into the `cardiBuddyStateList`.



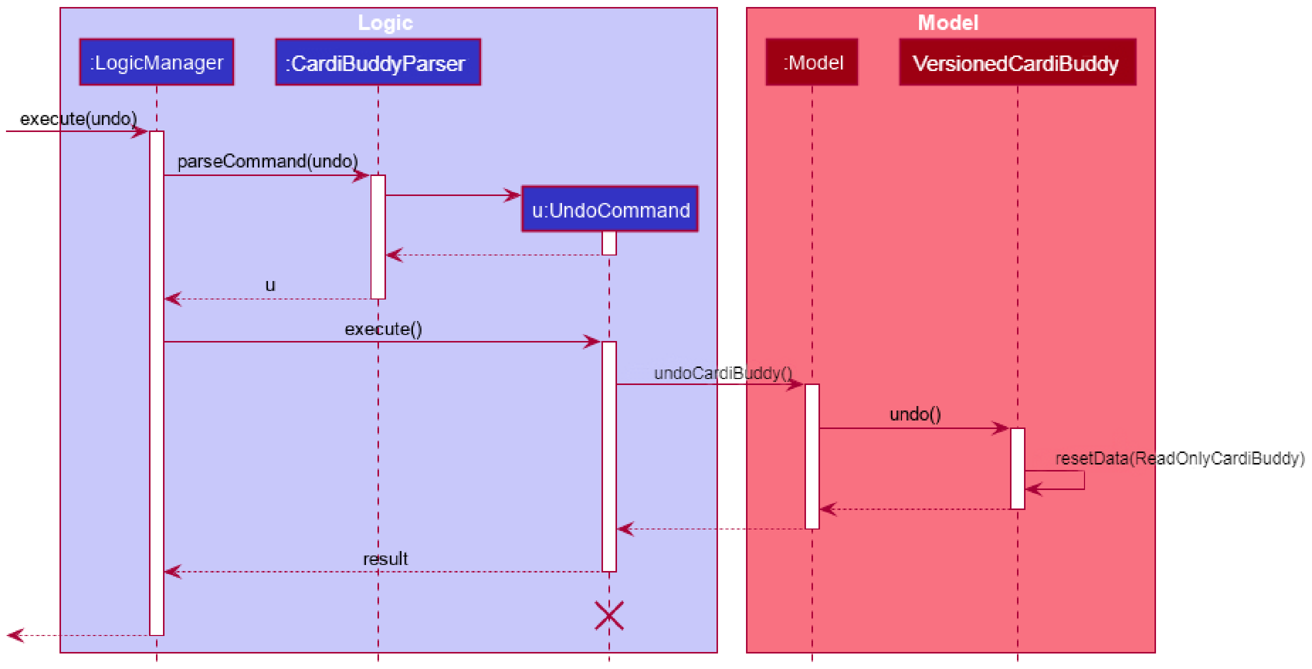
NOTE	If a command fails its execution, it will not call `Model#commitCardiBuddy()`, so the CardiBuddy state will not be saved into the `cardiBuddyStateList`.

Step 4. The user now decides that adding the deck was a mistake, and decides to undo that action by executing the `undo` command. The `undo` command will call `Model#undoCardiBuddy()`, which will shift the `currentStatePointer` once to the left, pointing it to the previous CardiBuddy state, and restores the CardiBuddy to that state.



NOTE If the currentStatePointer is at index 0, pointing to the initial address book state, then there are no previous CardiBuddy states to restore. The undo command uses Model#canUndoCardiBuddy() to check if this is the case. If so, it will return an error to the user rather than attempting to perform the undo.

The following sequence diagram shows how the undo operation works:



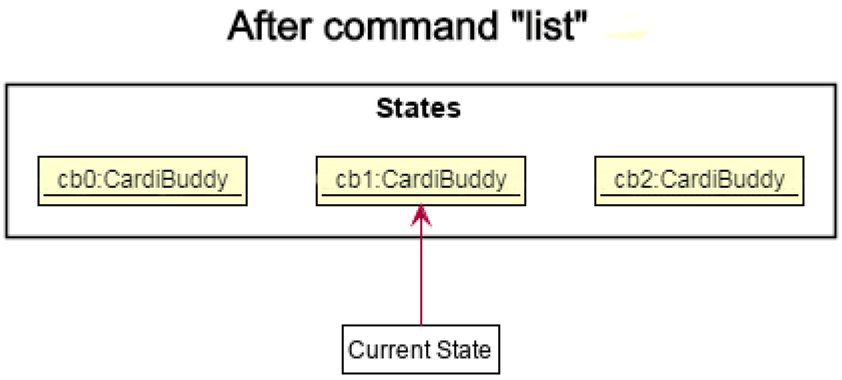
NOTE The lifeline for UndoCommand should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

The redo command does the opposite — it calls Model#redoCardiBuddy(), which shifts the currentStatePointer once to the right, pointing to the previously undone state, and restores the CardiBuddy to that state.

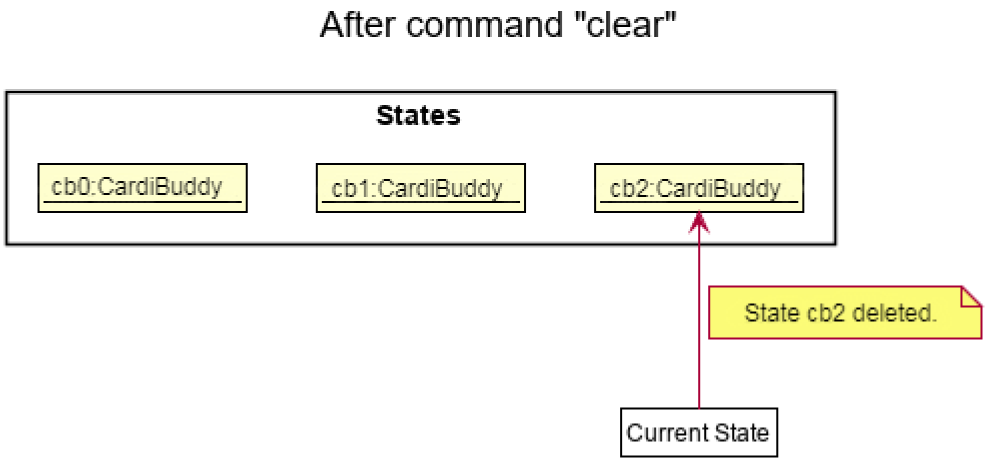
NOTE If the currentStatePointer is at index cardiBuddyStateList.size() - 1, pointing to the latest CardiBuddy state, then there are no undone CardiBuddy states to restore. The redo command uses Model#canRedoCardiBuddy() to check if this is the case. If so, it will return an error to the user rather than attempting to perform the redo.

Step 5. The user then decides to execute the command list. Commands that do not modify the

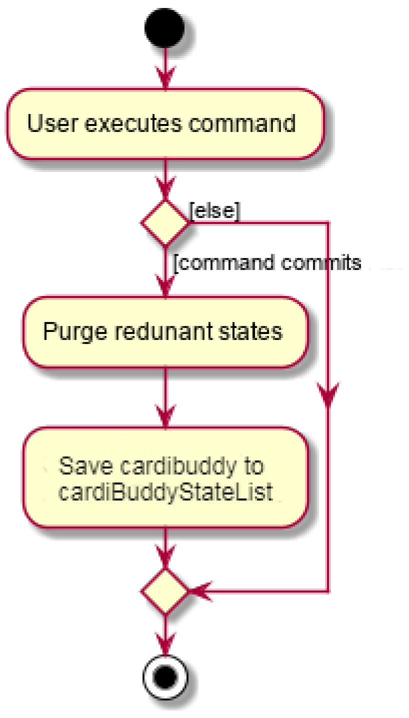
CardiBuddy, such as `list`, will usually not call `Model#commitCardiBuddy()`, `Model#undoCardiBuddy()` or `Model#redoCardiBuddy()`. Thus, the `cardiBuddyStateList` remains unchanged.



Step 6. The user executes `clear`, which calls `Model#commitCardiBuddy()`. Since the `currentStatePointer` is not pointing at the end of the `CardiBuddyStateList`, all CardiBuddy states after the `currentStatePointer` will be purged. We designed it this way because it no longer makes sense to redo the `add d/cs2103T …` command. This is the behavior that most modern desktop applications follow.



The following activity diagram summarizes what happens when a user executes a new command:



4.1.2. Design Considerations

Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Saves the entire CardiBuddy.
    - Pros: Easy to implement.
    - Cons: May have performance issues in terms of memory usage.
- **Alternative 2:** Individual command knows how to undo/redo by itself.
    - Pros: Will use less memory (e.g. for `delete`, just save the deck being deleted).
    - Cons: We must ensure that the implementation of each individual command is correct.

Aspect: Data structure to support the undo/redo commands

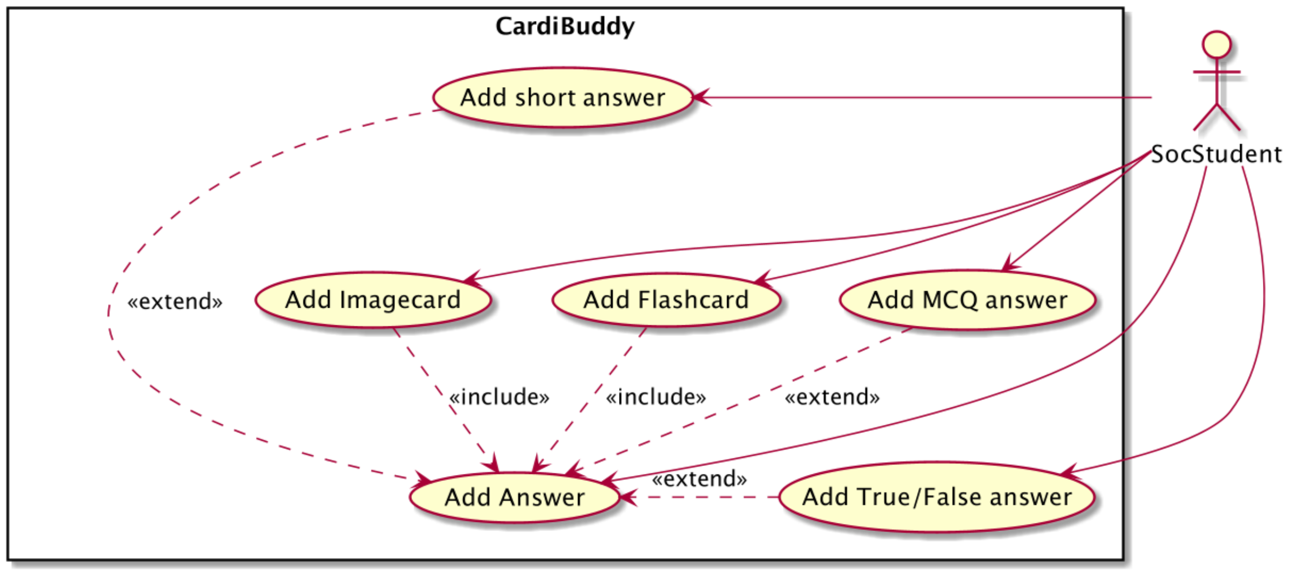
- **Alternative 1 (current choice):** Use a list to store the history of CardiBuddy states.
    - Pros: Easy for new Computer Science student undergraduates to understand, who are likely to be the new incoming developers of our project.
    - Cons: Logic is duplicated twice. For example, when a new command is executed, we must remember to update both `HistoryManager` and `VersionedCardiBuddy`.
- **Alternative 2:** Use `HistoryManager` for undo/redo
    - Pros: We do not need to maintain a separate list, and just reuse what is already in the codebase.
    - Cons: Requires dealing with commands that have already been undone: We must remember to skip these commands. Violates Single Responsibility Principle and Separation of Concerns as `HistoryManager` now needs to do two different things.

4.2. Flashcards

4.2.1. Design

Users are able to add two different types of cards — cards with images and cards without. These cards have three types of answers — True/False, MCQ and short answers.

The following is a Use Case Diagram for this feature:



Model Component

The following classes can be found inside *cardibuddy/model/flashcard*.

The add feature revolves around 2 abstract classes: `Card` and `Answer`.

The `Card` class is extended by two card classes: `Flashcard` and `Imagecard`.

The `Answer` class is extended by three answer classes: `TfAnswer`, `McqAnswer` and `ShortAnswer`.

Logic Component

To add a card, a deck must first be opened. This can be checked from accessing the `LogicToUiManager` which stores the currently opened deck. Subsequently, The `Parser` classes will separate the relevant arguments from the user input and execute commands from the `Command` classes. These `Parser` and Command` classes are part of the **Logic** component of CardiBuddy, and can be found within the *cardibuddy/logic* package.

These commands allow the user to add the different types of flashcards and answers into a deck:

- The user will first open a deck.
- *Adding a Flashcard:* The user will enter `add` followed by `q/` with their question and `a/` with their answer.
- *Adding an Imagecard:* The user will enter `add` followed by `p/` with the filepath to the image, `q/`

with their question and `a/` with their answer.

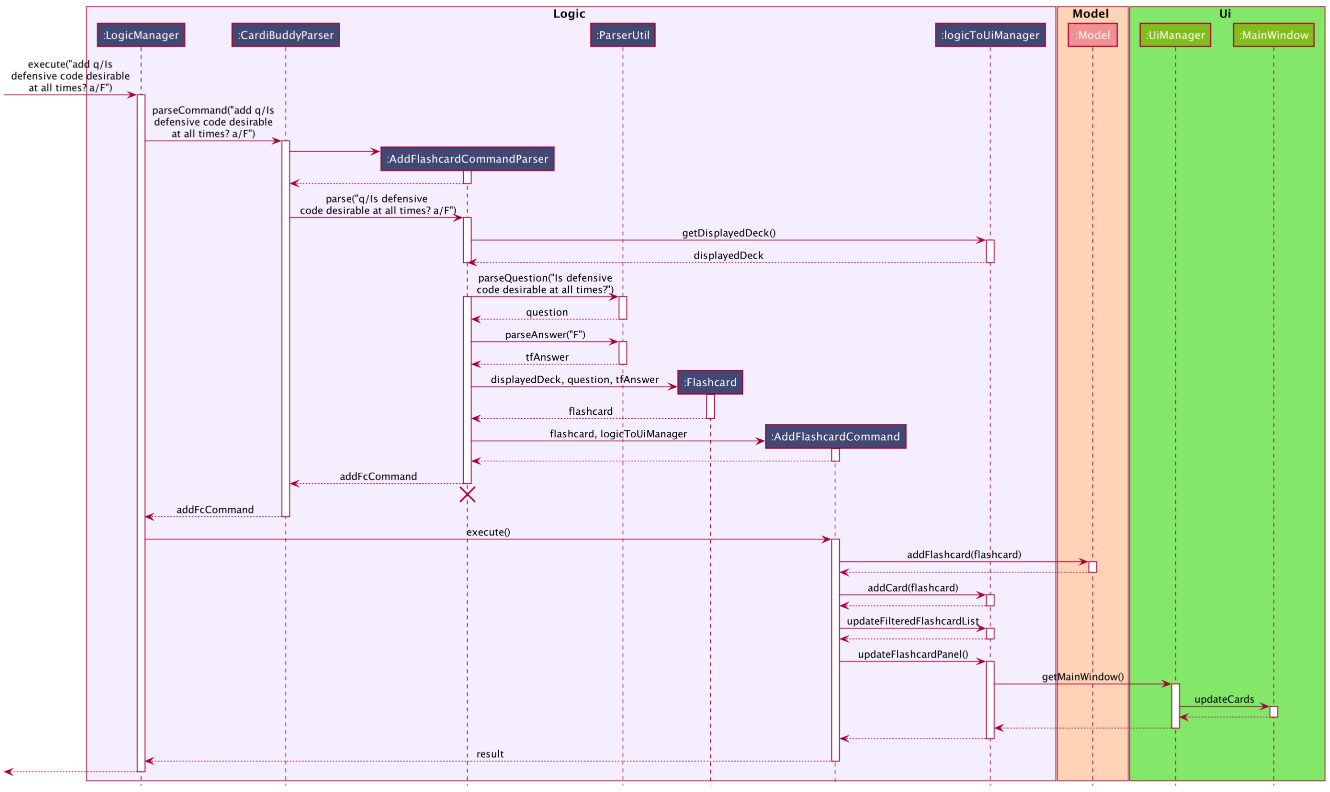
- *Adding a TfAnswer*: The user will enter either `T` or `F` for their answer after the `a/` prefix. Only capital, single-lettered answers are accepted and a `WrongTfException` will be thrown if the user enters `t`, `f`, `True` or `False`.
- *Adding an MCQAnswer*: The user will enter `A)CHOICE_A B)CHOICE_B C)CHOICE_C`, with the correct choice positioned first, for their answer after the `a/` prefix. In other words, if `C)CHOICE_C B)CHOICE_B A)CHOICE_A` is entered by the user, `C` will be taken as the correct answer. A `WrongMcqAnswerException` will be thrown if the user input does not have all three options in capital letters with parentheses.

4.2.2. Types of Cards

Flashcard

To add a `Flashcard` in an opened deck, the user will enter `add q/QUESTION a/ANSWER`. A sample command would be `add q/Is defensive code desirable at all times? a/F`.

The following sequence diagram shows how a `Flashcard` is created from the above command and displayed immediately in the flashcard panel to the user:



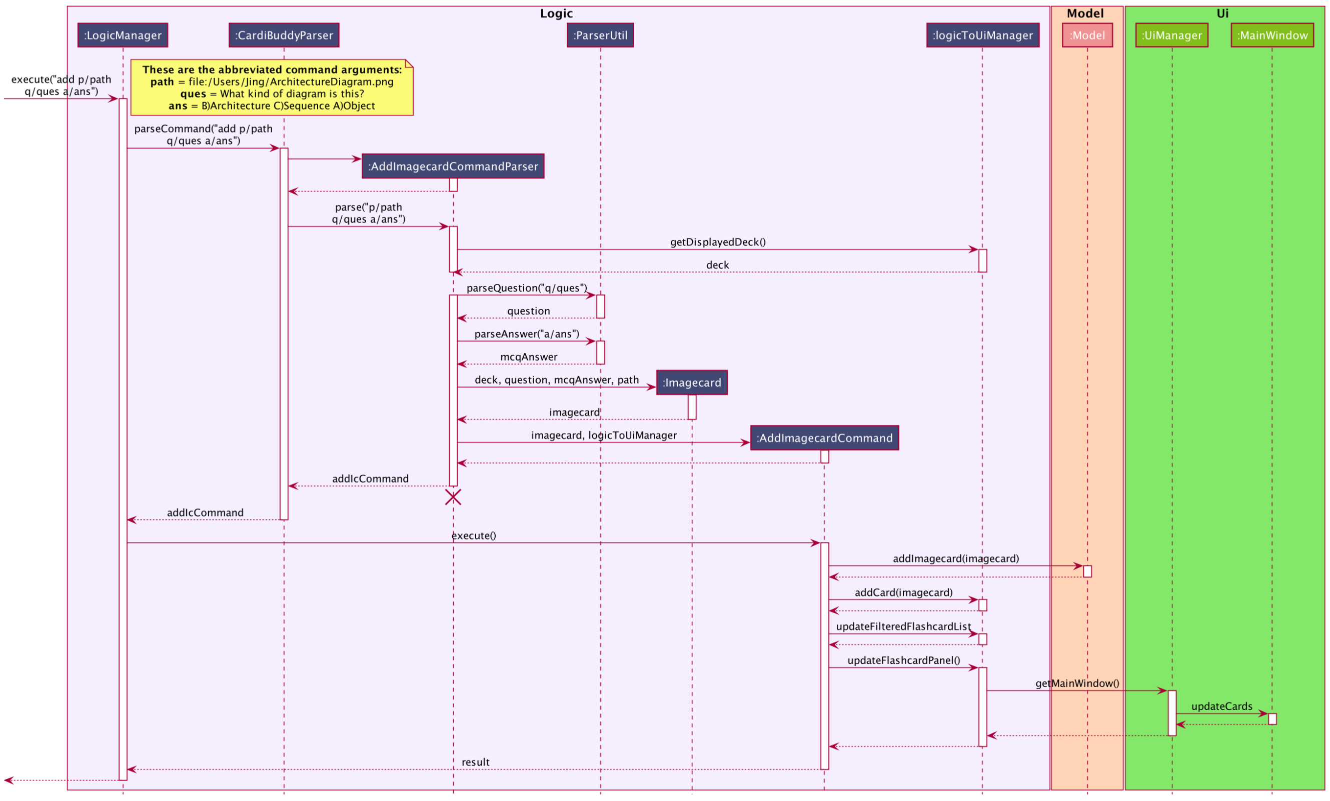
Logic
Model
Ui
:LogicManager
:CardiBuddyParser
:ParserUtil
:logicToUiManager
:Model
:UiManager
:MainWindow
execute("add q/Is defensive code desirable at all times? a/F")
parseCommand("add q/Is defensive code desirable at all times? a/F")
:AddFlashcardCommandParser
parse("q/Is defensive code desirable at all times? a/F")
getDisplayedDeck()
displayedDeck
parseQuestion("Is defensive code desirable at all times?")
question
parseAnswer("F")
tfAnswer
displayedDeck, question, tfAnswer
:Flashcard
flashcard
flashcard, logicToUiManager
:AddFlashcardCommand
addFcCommand
addFcCommand
execute()
addFlashcard(flashcard)
addCard(flashcard)
updateFilteredFlashcardList
updateFlashcardPanel()
getMainWindow()
updateCards
result

Imagecard

To add an `Imagecard` in an opened deck, the user will enter `add p/file:IMAGE_FILEPATH q/QUESTION a/ANSWER`. A sample command would be `add p/file:/Users/Jing/ArchitectureDiagram.png q/What kind of diagram is this? a/B)Architecture C)Sequence A)Object`.

When an `ImagecardCard` is displayed in the `FlashcardPanel`, the image will be retrieved via the stored `IMAGE_FILEPATH` from the user's computer. If the file path is invalid, the middle part of the card will be blank and an image will not be shown. More information regarding the implementation of the `Ui` can be found inside *cardibuddy/ui/ImagecardCard*.

The following sequence diagram shows how an `Imagecard` is created from the above command and displayed immediately in the flashcard panel to the user:



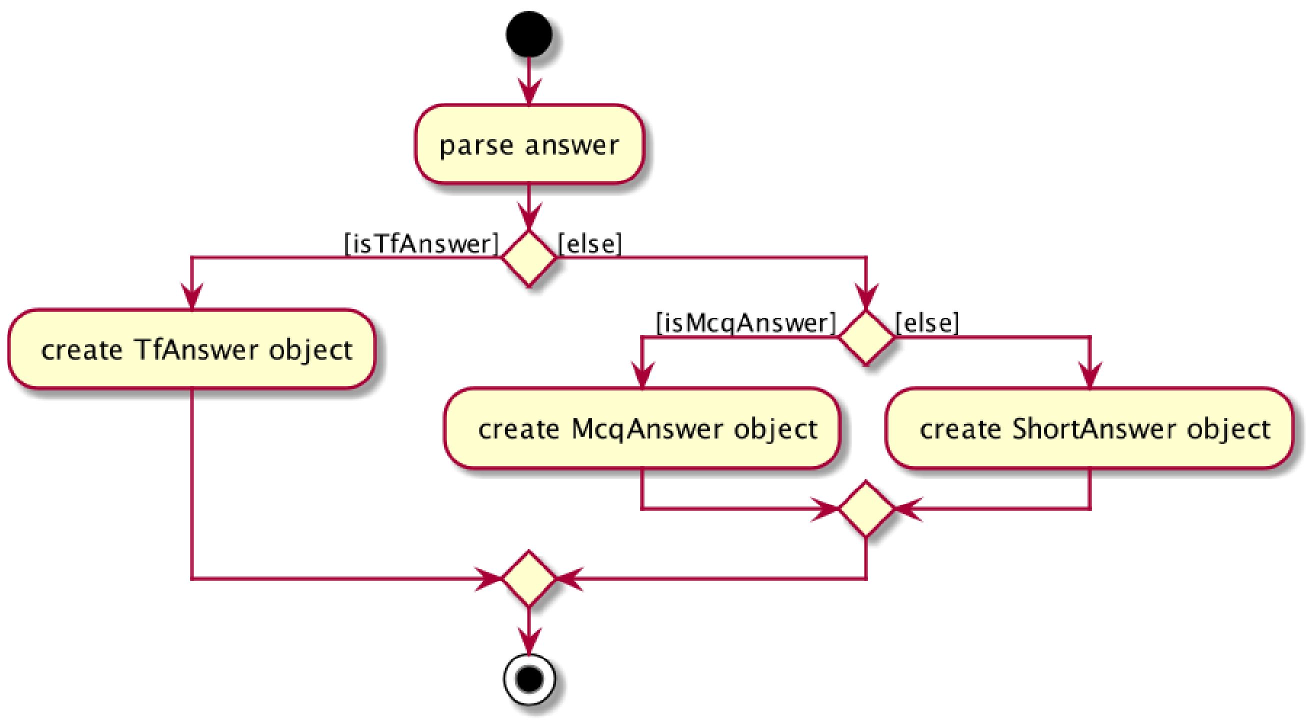
Logic
Model
Ui
:LogicManager
:CardiBuddyParser
:ParserUtil
:logicToUiManager
:Model
:UiManager
:MainWindow
These are the abbreviated command arguments:
path = file:/Users/Jing/ArchitectureDiagram.png
ques = What kind of diagram is this?
ans = B)Architecture C)Sequence A)Object
execute("add p/path q/ques a/ans")
parseCommand("add p/path q/ques a/ans")
:AddImagecardCommandParser
parse("p/path q/ques a/ans")
getDisplayedDeck()
deck
parseQuestion("q/ques")
question
parseAnswer("a/ans")
mcqAnswer
deck, question, mcqAnswer, path
:Imagecard
imagecard
imagecard, logicToUiManager
:AddImagecardCommand
addIcCommand
addIcCommand
execute()
addImagecard(imagecard)
addCard(imagecard)
updateFilteredFlashcardList
updateFlashcardPanel()
getMainWindow()
updateCards
result

It is largely similar to the sequence diagram for the creation of a `Flashcard` but with an extra `IMAGE_FILEPATH` argument.

4.2.3. Types of Answers

When adding cards, the `ParserUtil` will parse the different answer inputs to create one of the three different types of answers.

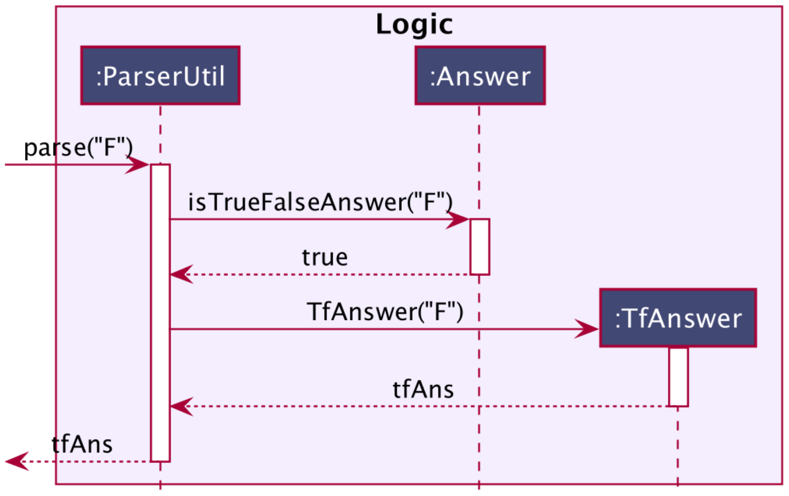
The following activity diagram shows how the `ParserUtil` chooses which type of answer object to create:



TfAnswer

For a `TfAnswer` to be associated with a `Card`, the user will have to enter either `T` or `F` after the answer prefix `a/`. A sample command was mentioned in the example for `Flashcard` above: `add q/Is defensive code desirable at all times? a/F`.

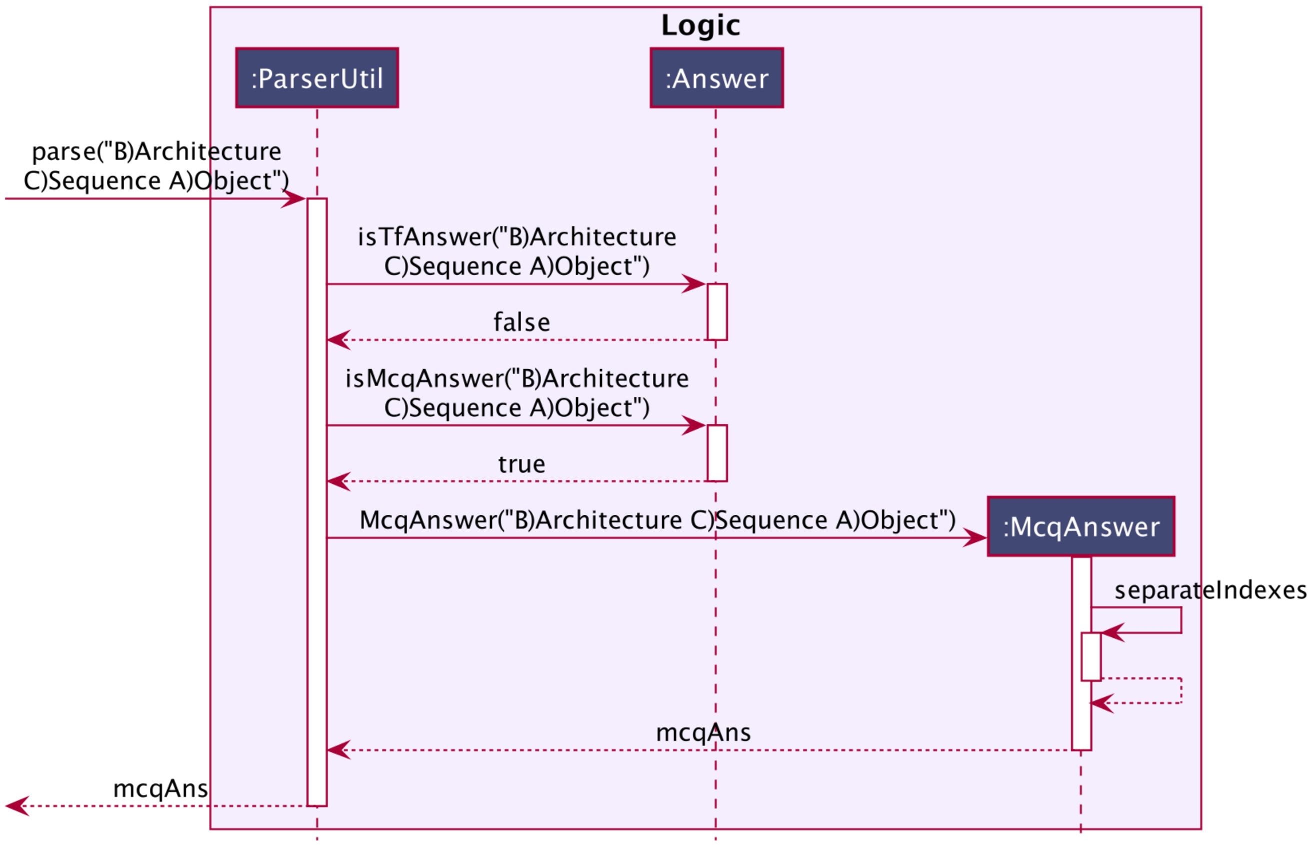
The following sequence diagram shows how the `ParserUtil` class creates a `TfAnswer` answer based on the given sample command:



McqAnswer

For an `McqAnswer` to be associated with a `Card`, the user will have to enter `A)CHOICE_A B)CHOICE_B C)CHOICE_C`, with the correct choice positioned first, for their answer after the answer prefix `a/`. A sample command was mentioned in the example for `Imagecard` above: `add p/file:/Users/Jing/ArchitectureDiagram.png q/What kind of diagram is this? a/B)Architecture C)Sequence A)Object`.

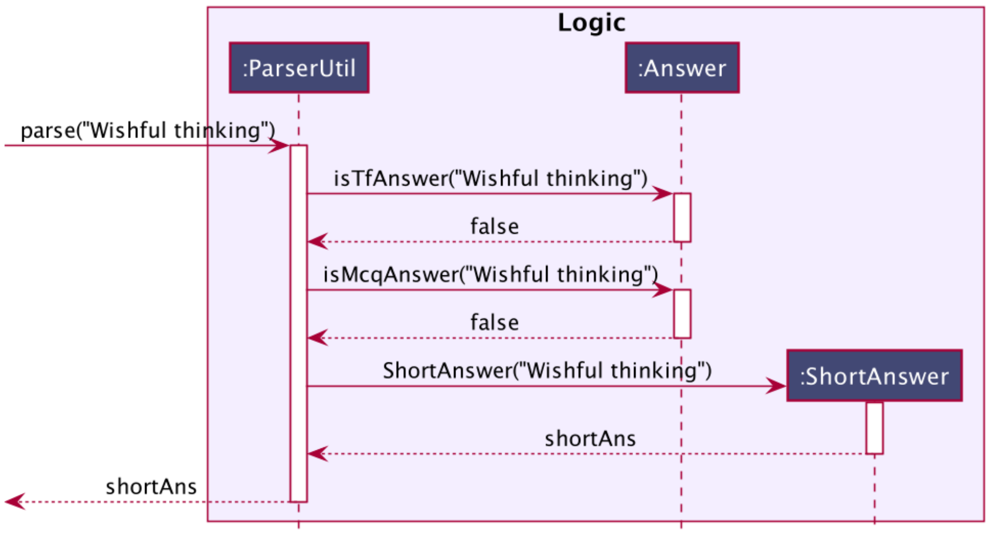
The following sequence diagram shows how the ParserUtil class creates an McqAnswer answer based on the given sample command:


Logic
:ParserUtil
:Answer
parse("B)Architecture C)Sequence A)Object")
isTfAnswer("B)Architecture C)Sequence A)Object")
false
isMcqAnswer("B)Architecture C)Sequence A)Object")
true
McqAnswer("B)Architecture C)Sequence A)Object")
:McqAnswer
separateIndexes
mcqAns
mcqAns

ShortAnswer

For a ShortAnswer to be associated with a Card, the user will have to enter an answer that does not fulfil the requirements of both TfAnswer and McqAnswer after the answer prefix a/. A sample command would be `add q/How does one go about solving recursion problems? a/Wishful thinking`.

The following sequence diagram shows how the ParserUtil class creates a ShortAnswer answer based on the given sample command:


Logic
:ParserUtil
:Answer
parse("Wishful thinking")
isTfAnswer("Wishful thinking")
false
isMcqAnswer("Wishful thinking")
false
ShortAnswer("Wishful thinking")
:ShortAnswer
shortAns
shortAns

4.2.4. Design Considerations

Aspect: How to implement the different card and answer types

- **Alternative 1 (current choice):** Using an abstract class to contain general functionalities.
  - Pros: Can define method bodies general methods. Easier to make changes such as creating new methods which can be defined directly in the abstract class.
  - Cons: Child `Card` and `Answer` classes cannot extend multiple abstract classes.
- **Alternative 2:** Using an interface to contain general functionalities.
  - Pros: Supports multiple inheritances.
  - Cons: Cannot define method bodies in the interface.

Due to the similarities in method bodies of the different answers and cards classes, a `Card` and `Answer` abstract class was used instead of an interface.

4.3. Test feature

4.3.1. Design

A flashcard application is not complete without the ability to test oneself.

Model Component

The following classes can be found inside *cardibuddy/model/testsession*.

The test feature revolves around 2 classes: `TestSession` and `TestResult`.

`TestSession` stores the tested deck, retrieves the questions to be tested, acts according to the user's commands and manages the test queue.

`TestResult` stores the `Result` (explained below) of each *individual* test on a flashcard. That is, whether the flashcard was answered wrongly, correctly, or was skipped. `Result` is an enums class to store these 3 outcomes.

Another enums class used is `AnswerType` which contains 3 answering options to display to the user - True/False, MCQ or Short Answer.

To ensure good design practice, `Test Session` is not exposed to the other classes, but is instead called using the `ModelManager`.

The following figure is a more focused class diagram for the *cardibuddy/model/testsession* package, and displays important methods and fields used.

Click here to view the full class diagram for the `Model` component, to see all the classes within `Model` interact.

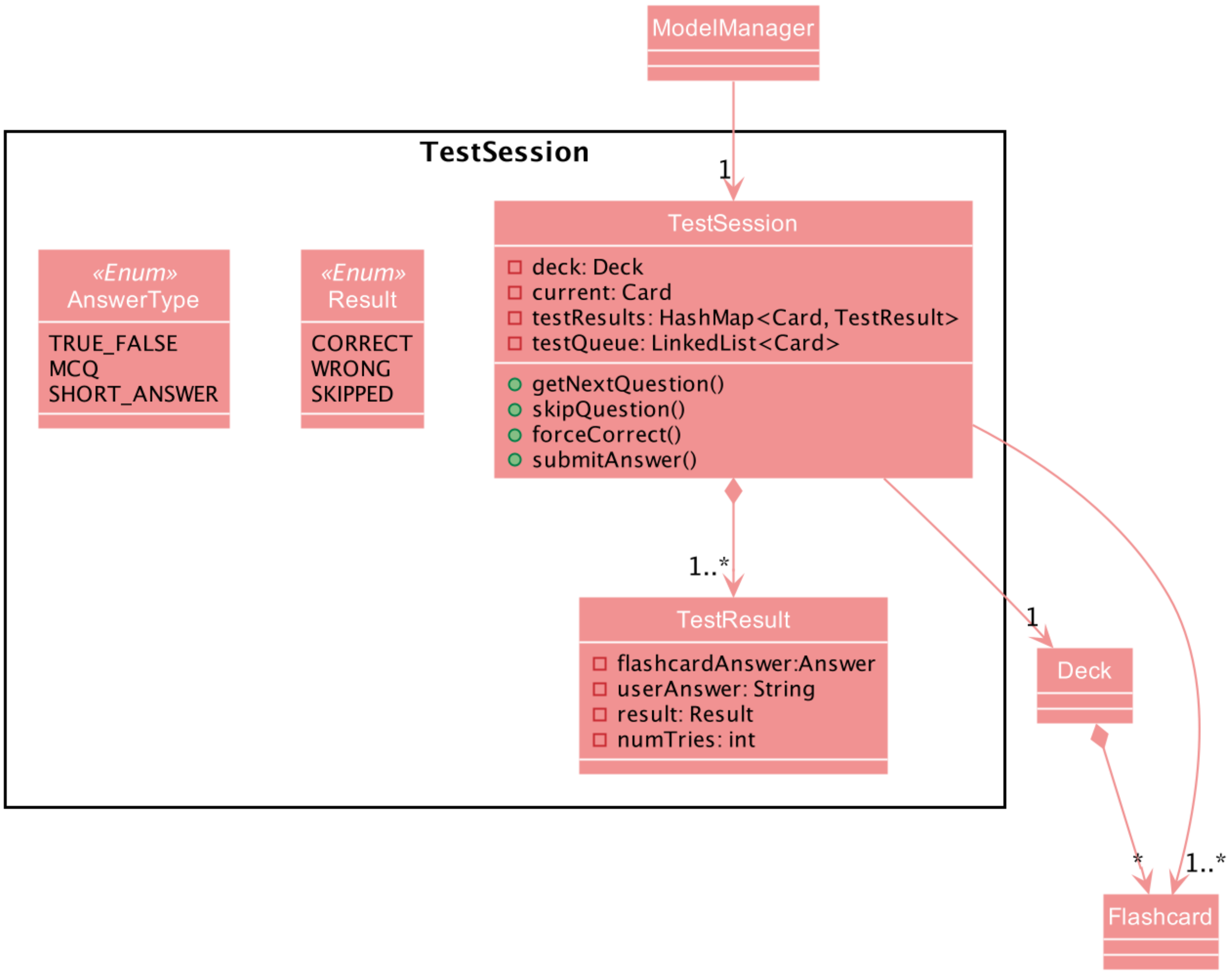


Figure 10. Class diagram for the test session package.

Logic Component

The test feature makes use of a suite of `Command` classes exclusive to a `TestSession`. These `Command` classes are part of the **Logic** component of CardiBuddy, and can be found within the *cardibuddy/logic/testsession* package.

These commands allow the user to perform the following during a test session:

- The user will enter `test INDEX` to start the test session for the deck at the `INDEX`.
- The user will enter `ans` followed by their answer to the question.
- The user can use the `next` command to view the next question.
- The user can choose to `skip` questions
- The user can choose to `quit` the session halfway
- The user can choose to `force` correct their answer if they wish to manually mark their answer as correct.

The above commands can only be triggered when certain conditions are met. The following table provides a summary of all the `Command` classes related to the test feature, as well as the conditions for their execution and exceptions thrown when these conditions are not met.

4.3.2. Using the test feature

Starting a Test Session

Users can start a TestSession with a chosen `Deck`:

- The `index` of the deck will be provided by the user and parsed by Cardi Buddy. A `TestSession` object containing the indicated `Deck` is created.
- The `TestSession` object creates a HashMap named `testResults` that contains `<Flashcard, TestResult>` for easy access to the testing history.

The following is a UML Sequence Diagram of how a `TestSession` object is created when the `test INDEX` command is called:

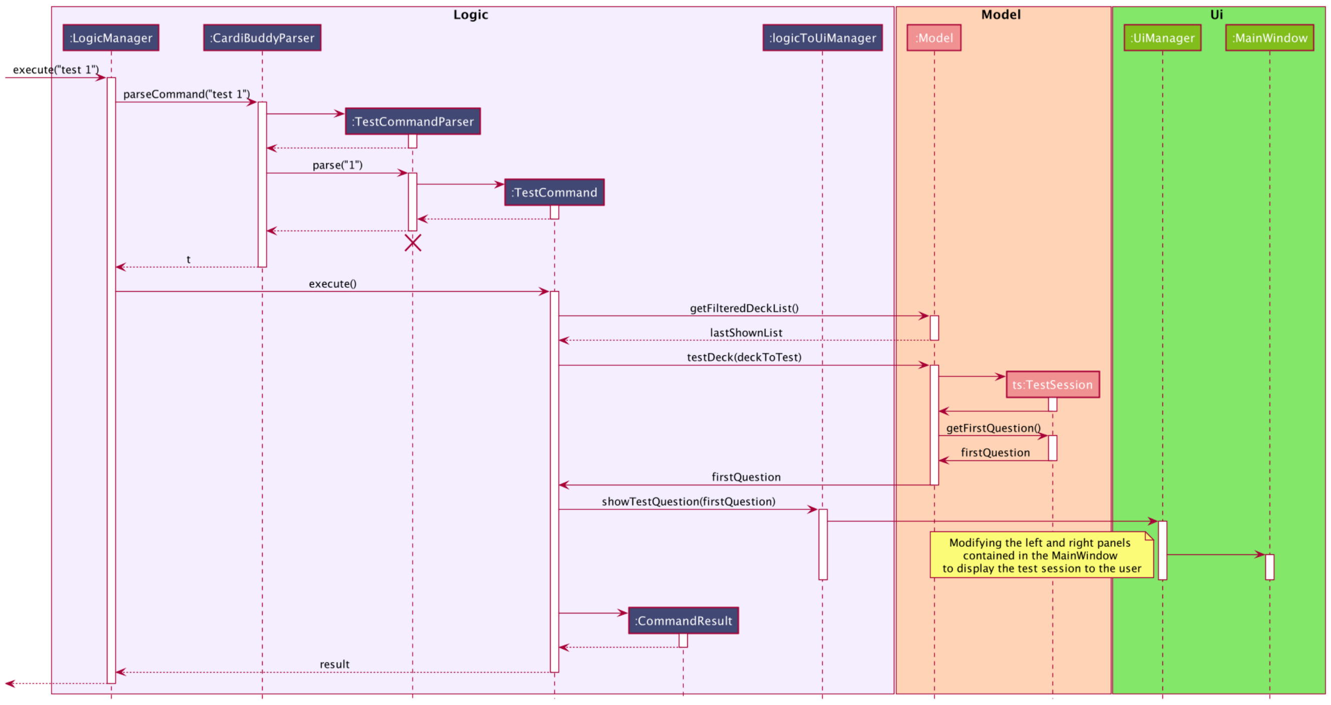


Figure 11. A sequence diagram illustrating the logic flow when a TestSession is created. The first question in the provided deck will immediately be displayed.

Other commands included in the Test Feature

Aside from `TestCommand` which is called to create the `TestSession`, there are specific commands that can be used only when a TestSession is running. These commands have certain **conditions** that must be met before they can be executed. Otherwise, they will throw a `CommandException`.

For example, other application-wide commands, such as `add deck` and `delete card` will not be allowed to be executed during the TestSession.

The following activity diagrams describe the logic flow.

Note the following terminology used in the activity diagrams:

- `tr` stands for a `TestResult` object
- `testResults` stands for a `HashMap<Flashcard, TestResult>` object stored in the `TestSession` object

- `testQueue` is a `LinkedList<Flashcard>` that stores the queue of flashcards to be tested.

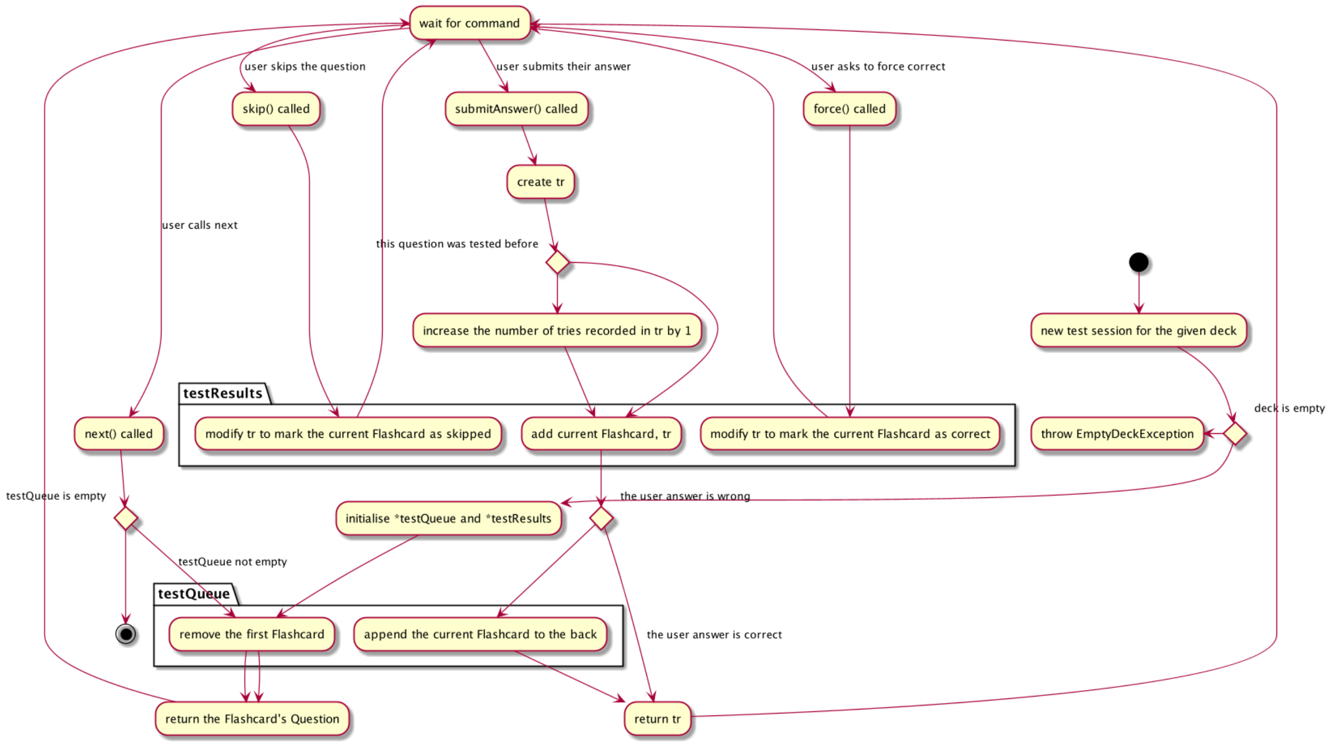

wait for command
user skips the question
user submits their answer
user asks to force correct
skip() called
submitAnswer() called
force() called
create tr
user calls next
this question was tested before
increase the number of tries recorded in tr by 1
new test session for the given deck
testResults
next() called
modify tr to mark the current Flashcard as skipped
add current Flashcard, tr
modify tr to mark the current Flashcard as correct
throw EmptyDeckException
deck is empty
testQueue is empty
initialise *testQueue and *testResults
the user answer is wrong
testQueue not empty
testQueue
remove the first Flashcard
append the current Flashcard to the back
the user answer is correct
return the Flashcard's Question
return tr

Figure 12. Overall activity diagram for the different use cases, extension cases omitted.

For more descriptive use case scenarios from a user's perspective, please take a look at *Appendix C, Use Cases*.

4.3.3. Design considerations

Aspect: Data structure for the test queue

- **Alternative 1**
- Use an ArrayList to store the flashcards in the test queue.
  - Pros:
    - Makes use of the current data structure used to store flashcards in a list.
    - Easy retrieval of flashcards at indices.
    - Easy iteration of the flashcard list.
  - Cons:
  - Possible difficulties in designing how flashcards should be retested.
  - It is possible that we create a large ArrayList, and split it such that the first half is used for flashcards that have not been tested, and the second half be used for flashcards that are going to be retested.
  - However, this approach is unnecessarily complicated, and uses excessive memory space.
  - In addition, it is still difficult to check if flashcards have been tested before, which is part of our feature which counts the number of times a flashcard has been attempted.
- **Alternative 2 (current choice)**

- Use a LinkedList to store the flashcards in the test queue.
    - Pros:
        - Easy insertion and removal of flashcards from the front and back of the queue, which is the main driver behind a test session.
    - Cons:
        - Difficult to retrieve flashcards at indices, and check if a flashcard has been visited before
        - In addition, other features of CardiBuddy require the easy retrieval of flashcards such as the get(index) method offered by ArrayList.
    - Why we went with this:
        - Logically, a test session will not be used as often as the other commands that entail retrieval of elements at a specific index, such as the `edit` command.
        - There may be some overhead in converting the ArrayList used to a LinkedList, but our tests have showed minimal lag in starting a test session.
    - Workarounds:
        - To still retain the ability to check if a flashcard has been visited before, we use a separate HashMap data structure that makes use of flashcards that have been visited before as keys.
    - A HashMap will also allow us to link a flashcard to a specific TestResult, and modify it as needed (such as when a user calls force correct on the flashcard).

Aspect: How to store the results of each individual test on a flashcard

- **Alternative 1**
- Use a `Result` enums to indicate if the question was correct, wrong or skipped
    - Pros:
        - An enums class improves code readability, and ensures that there are only 3 possible results linked to each flashcard.
    - Cons:
        - Too simplistic and does not give room for further enhancements as described below.
- **Alternative 2 (current choice)**
- Use a custom `TestResult` class that stores the user answer, as well as the model flashcard answer
    - Pros:
        - Provides room for further customisation, such as recording the number of attempts on the flashcard, which alternative 1 was unable to achieve.
        - More Object-Oriented Design, and keeps `TestSession` from getting too cluttered.
    - Cons:
        - Higher chance for error as more code and methods needed to be written.

4.3.4. Aspect: When to replace the `current` variable holding the latest flashcard that was tested

- **Alternative 1**
- Move on to the next question, hence replacing `current`, as soon as the user submits their answer.
  - Pros:
    - Simple and easy to implement
  - Cons:
    - This would have been fine in the initial stages, but would not work after further enhancements (`skip`, `force`) were added.
    - The 2 mentioned enhancements require modifying of the `testResults` HashMap, and hence require the flashcard they are being used on as a key to retrieve and modify the TestResult.
- **Alternative 2 (current choice)**
- Only replace `current` with the next flashcard when `next` or `skip` is used.
  - Pros:
    - These 2 commands are the ones that trigger the displaying of the next question in the queue to the user.
    - This ensures that should there be a need to retrieve the current flashcard, such as in modifying the `testResults` HashMap as well as checking the flashcard's `CardType` (image or normal) and getting its `AnswerType`.
  - Cons:
    - This complicates the code, and may possibly be difficult to understand for another developer.
  - Workarounds:
    - Do our best to document the test feature in this Developer's Guide, and supplement it with activity diagrams and other explanations to help the reader better understand the workings of this feature.

4.4. Statistics

4.4.1. Design

The user's activities will be logged and stored in `Statistics`. It is entirely a data storage class, recording data wherever necessary.

Currently, the actions that update the `Statistics` are:

- Adding and removing a deck
- Adding and removing a card

- Finishing a `TestSession` - this updates a number of items, which are:
  - `TestSessions` finished
  - Total cards answered
  - Average correct percentage
  - Average tries to get correct

There’s 2 different locations where a `Statistics` instance is stored:

- `CardiBuddy`: The Model of the program. Tracks user statistics across the whole program.
- `Deck` : Each `Deck` also has a `Statistics` instance. This is to help identify which `Deck`s are used more often, are more difficult, etc.

Each time the user does something that can be tracked, it updates both the universal `Statistics` and the `Statistics` of the `Deck` that he or she is using. The call to update both statistics is done simultaneously in the `CardiBuddy` model for adding/removing decks/cards, or in `ModelManager` for finishing a `TestSession`.

4.5. Logging

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

- The logging level can be controlled using the `logLevel` setting in the configuration file (See Section 4.6, “Configuration”)
- 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.

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

4.6. Configuration

Certain properties of the application can be controlled (e.g user prefs file location, logging level) through the configuration file (default: `config.json`).

5. Documentation

Refer to the guide here.

6. Testing

Refer to the guide here.

7. Dev Ops

Refer to the guide here.

Appendix A: Product Scope

Target users: NUS School of Computing students

Target user profile:

- enrolled in content-heavy modules in university
- has a need for an effective way to revise and memorise content
- values efficiency
- prefer desktop apps over other types
- able to type quickly
- generally prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition:

1. Time efficient
    - It is easier for fast typers to add and delete flashcards
    - Unlike regular GUI apps in the market, minimal navigation and clicking is required
    - Faster loading time for CLI applications
2. No steep learning curve
    - Command words are intuitive and uncomplicated
    - Easy for computing students to learn and remember
3. Retests cards
    - More tests for flashcards that the student got wrong
4. Flexible answering
    - Students can paraphrase their answers, and manually evaluate if their answers are correct
    - No need for word-for-word answers

- Accommodates different module types and scenarios eg. having to describe a situation, or a diagram
- Paraphrasing is also a much more effective way to learn, compared to rote memorisation

Appendix B: User Stories

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

Priority	As a …	I want to …	So that I can…
* * *	new user	see usage instructions	refer to instructions when I forget how to use CardiBuddy
* * *	student currently taking cs2105	edit my flashcards	change or add extra information whenever I learn something new
* * *	student practicing for my finals	create test sessions	repeatedly test myself on the same content
* * *	student studying for finals	see the flashcards that I got correct during test sessions	know what content I am more familiar with
* * *	student studying for finals	test flashcards that I got wrong more often during test sessions	better remember unfamiliar content
* * *	student taking many modules	create new decks to contain my flashcards	organise my notes and modules
* * *	student who is very busy	easily search for a deck that I want to access	more efficient with my time
* * *	student who likes to keep things organised	delete decks of the modules that I am no longer taking	be more organised

Priority	As a …	I want to …	So that I can…
`* * *`	student with many content-heavy modules	easily search for any flashcards that are relevant to my modules	more efficient with my time
`* *`	student	revisit previous test sessions	continue my revision
`* *`	student	set priority levels for the flashcards I am less familiar with	it will appear more often during future test sessions
`* *`	student	tag flashcards with different topics	filter and revise the topics that I am less familiar with
`* *`	student taking timed examinations	time myself during a test session	better prepared to think under timed conditions
`* *`	student who likes designing and aesthetics	customise the colours and fonts of the flashcards	tweak the theme to my preferences
`* *`	student who owns multiple devices	access flashcards on all my devices	revise them while travelling
`* *`	student who receives flashcard images from her friends	drag and drop the images into the application	conveniently create new flashcards
`* *`	student with short attention span	play memory games in the application	remember my key concepts better

Priority	As a …	I want to …	So that I can…
*	competitive student	graded on a bell-curve with other Computer Science students who are using the same application	see how well I have revised compared to the rest
*	student	edit other people’s decks	fill in any gaps in my knowledge
*	student taking modules with other friends	collaborate on decks with other users	help each other revise the content
*	student who likes to store content to study on her phone	convert the flashcards to images	so that I can refer to them easily
*	student who likes to study with her friends	send my friends flashcards that I created	share my flashcards with them
*	student who needs incentive	earn rewards	will be motivated to use the flashcards more
*	student with a short attention span	add animations to my flashcards	remain entertained

Appendix C: Use Cases

(For all use cases below, the **System** is the `CardiBuddy` and the **Actor** is the `user`, unless specified otherwise)

Test Sessions

C.1. Use case: Starting a test session

MSS

1. User requests to start a test session with a chosen deck.
2. CardiBuddy displays the test session page to the user, with the first question displayed.
3. CardiBuddy awaits the user's answer.

   Use case ends.

Extensions

1a. The given index is invalid

- i. CardiBuddy shows an error message
- ii. Use case resumes at step 1.

2a. The deck is empty

- i. Use case ends.

C.2. Use case: Submitting an answer

MSS

1. User submits their answer to a question not tested before.
2. CardiBuddy gets the result of the user's answer. The user answered the question correctly.
3. CardiBuddy creates a new record for this flashcard to save this correct result.
4. CardiBuddy displays the result to the user.

   Use case ends.

Inclusions

1a. The current flashcard has been answered before.

- i. CardiBuddy modifies its records by increasing the number of tries logged for this flashcard.
- ii. Use case resumes at step 3.

2a. The user got the question wrong.

- i. CardiBuddy creates a new record for this flashcard to save this wrong result.
- ii. CardiBuddy appends this flashcard to the back of the queue for retesting later.
- iii. Use case resumes at step 4.

C.3. Use case: Skipping a question

MSS

1. User requests to skip the current question.
2. CardiBuddy modifies its records to show that this flashcard was skipped

3. CardiBuddy removes the next flashcard in the queue.
4. CardiBuddy displays the question on this flashcard to the user.

   Use case ends.

Extensions

1a. The user has already answered this question correctly

- i. CardiBuddy shows an error message and prompts user to type 'next' instead
- ii. Use case ends.

Inclusions

- 1a. The user has already answered this question wrongly
  - i. CardiBuddy removes this flashcard, that was set to be retested, from the back of the queue
  - ii. Use case resumes at step 2.

C.4. Use case: Forcing a wrong answer to be marked correct

MSS

1. User requests to force their answer to be marked as correct
2. CardiBuddy acknowledges the user’s request and changes the recorded result for this flashcard + Use case ends.

Extensions

1a. The user has not answered the question yet

- i. CardiBuddy shows an error message and tells the user to answer the question first, or skip it
- ii. Use case ends.

1a. The user is trying to force correct their already correct answer

- i. CardiBuddy shows an error message and prompts the user to type 'next'
- ii. Use case ends.

C.5. Use case: Going to the next question

MSS

1. User requests to go to the next question.
2. CardiBuddy removes the next flashcard in the queue and displays its question to the user.

Use case ends.

Inclusions

- 2a. There are no more flashcards in the queue
    - i. CardiBuddy ends the test session, and returns the user to the home page
    - ii. Use case ends.

Extensions

- 1a. The user has not answered the question yet
    - i. CardiBuddy shows an error message and prompts the user to answer the question or skip it.
    - ii. Use case ends.

C.6. Use case: Add a new deck

MSS

1. User requests to add a new deck
2. CardiBuddy creates a new deck with the specified name

Extensions

1a. A deck with the same name already exists

- i. CardiBuddy shows an error message, tells the user that there already exists a deck with the same name.
- ii. Use case ends.

1b. While creating the deck, user also specifies a few tags to attach to the deck.

- i. At step 2, CardiBuddy creates a deck with the specified name and tags.
- ii. Use case ends.

1c. The user tries to create tags with more than one word

- i. CardiBuddy shows the user an error message, as tags cannot have more than one word.
- ii. Use case ends.

{More to be added}

Appendix D: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java 11 or above installed.
2. Should be able to hold up to 100 decks without a noticeable sluggishness in performance for typical usage.

3. Each deck should be able to hold up to 100 flashcards without a noticeable sluggishness in performance for typical usage.
4. A user with above average typing speed for regular English text (i.e. not code, not system admin commands) should be able to accomplish most of the tasks faster using commands than using the mouse.

Appendix E: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

Flashcard

A card created by the user to test themselves. Contains a question and an answer.

Example

```
Question: `Give the code to create a new ArrayList containing Integers`
Answer: `ArrayList<Integer> lst = new ArrayList<>();`
```

Deck

A group containing flashcards belonging to the same category, both of which are created and defined by the user.

Example

```
A deck named "CS2103T" contains the flashcards testing CS2103T content.
```

Tag

A single word that can be attached to a deck. Typically describes the category the deck belongs to, and is used to filter and organise the user’s flashcards.

Example

```
A tag called "computing" can be assigned to decks named "CS2103T", "CS2101" and
"CS3223". When the user filters their deck by the tag "computing", these 3 decks will
be shown. (These 3 modules are read in the School of Computing)
```

Test Session

A session started by the user when the user wishes to test themselves on the contents of a deck. Each flashcard in the deck specified by the user is shown sequentially, and can only proceed when the user enters the answer to the question.

Test Queue

A temporary queue created when a Test Session is started. It stores the flashcards that either have not been tested yet, or have been tested but the user got wrong.