Dukemon - Developer Guide 🔴

By: Team SErebros   Since: Aug 2019   Licence: MIT

1. Setting up

Refer to the guide here.

2. Design

2.1. Architecture
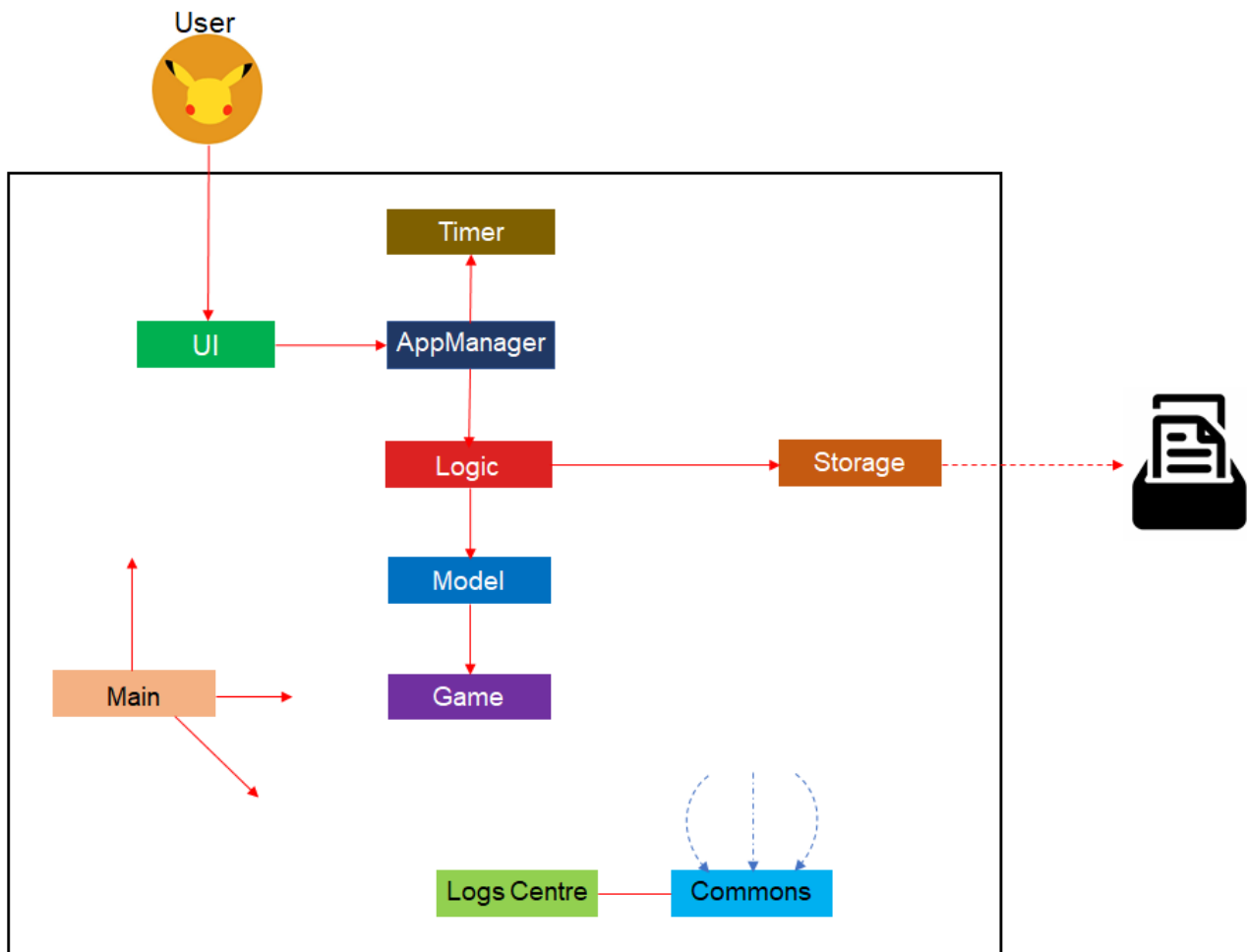
Figure 1. Dukemon Architecture Diagram

The **Architecture Diagram** given above explains the high-level design of Dukemon. 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 **Dukemon** contains *seven* componenets.

- `UI`:
  The Graphical UI of Dukemon that interacts with the user.

- `AppManager`:
  The buffer between the User and Dukemon's internal components.

- `Timer`:
  The internal Timer that triggers events based on time elapsed.

- `Logic`:
  The main command executor and performer of operations.

- `Model`:
  Holds the non-game data in-memory.

- `Game`:
  Holds the data of live game sessions in-memory.

- `Storage`:
  Reads data from, and writes data to, the local hard disk.

For the components UI, Logic, Model, Timer, Storage and Game:

- Defines its *API* in an `interface` with the same name as the Component.
- Exposes its functionality using a `{Component Name}Manager` class.
  - ie. `StorageManager` implements `Storage`, `GameTimerManager` implements `GameTimer`.

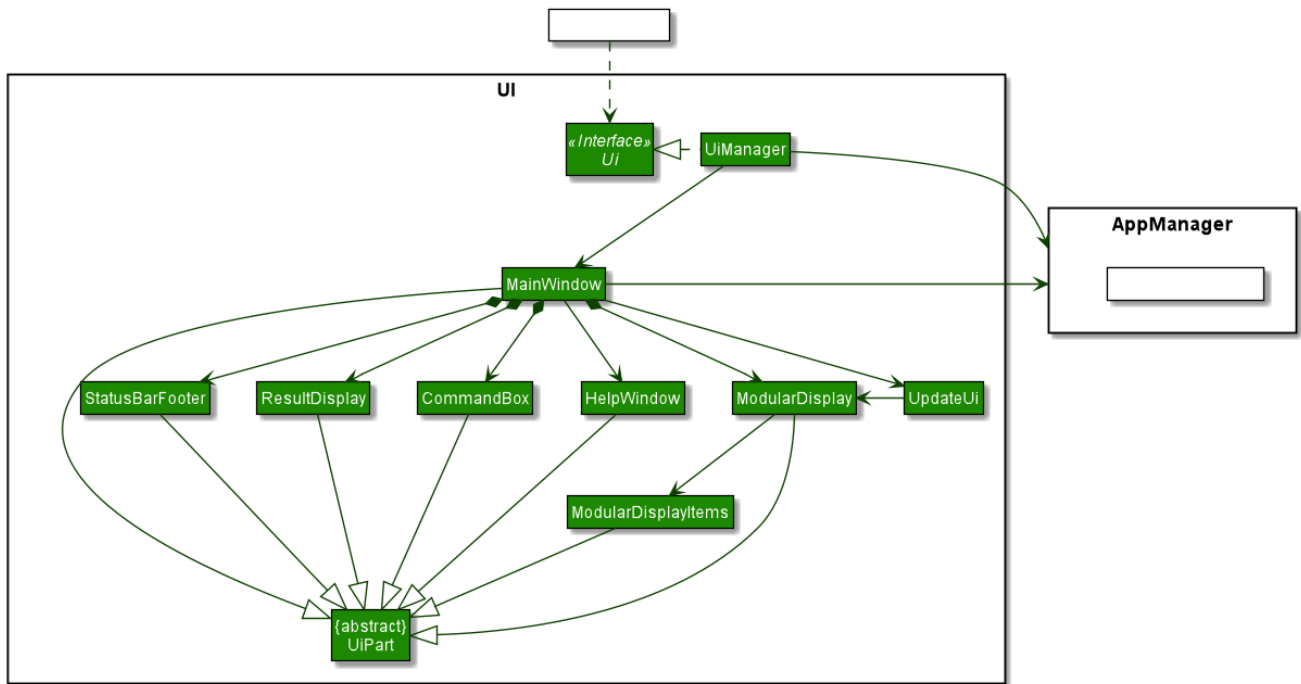The sections below give more details of each component.

2.2. UI component

Figure 2. Structure of the UI Component

API : `Ui.java`

The UI consists of a `MainWindow` that is made up of parts e.g.`CommandBox`, `ResultDisplay`, `ModularDisplay`, `StatusBarFooter` etc. All these, including the `MainWindow`, inherit from the abstract `UiPart` class.
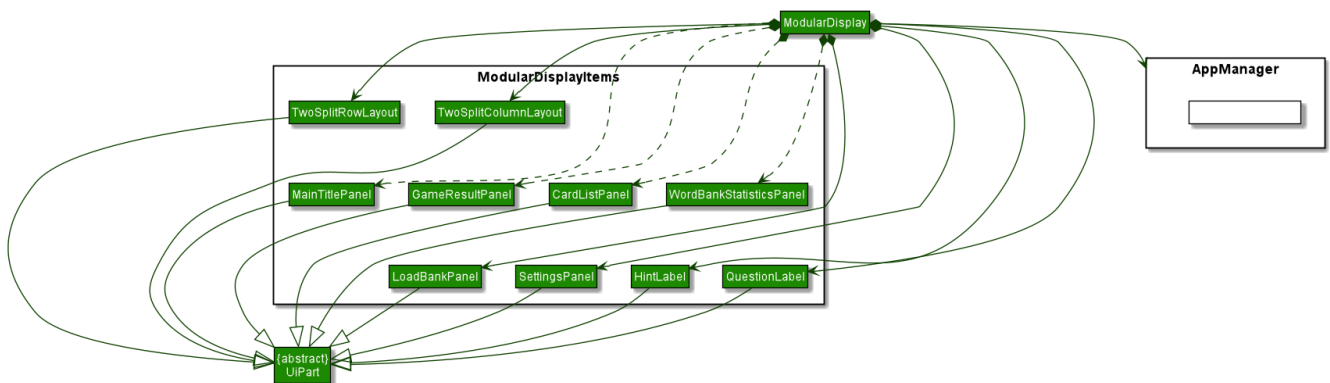
Figure 3. Structure of the ModularDisplay items.

As our application also comprises a dynamically changing UI in replacement of the `PersonListPanel`, `ModularDisplay` contains multiple other `Panel` classes in itself in order to render the proper screens during runtime. This rendering is controlled by `UpdateUi` in `MainWindow`.

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 `AppManager` component.
- Listens for changes to `Model` data and `Timer` through the `AppManager` so that the UI can be updated correspondingly.

2.3. AppManager component
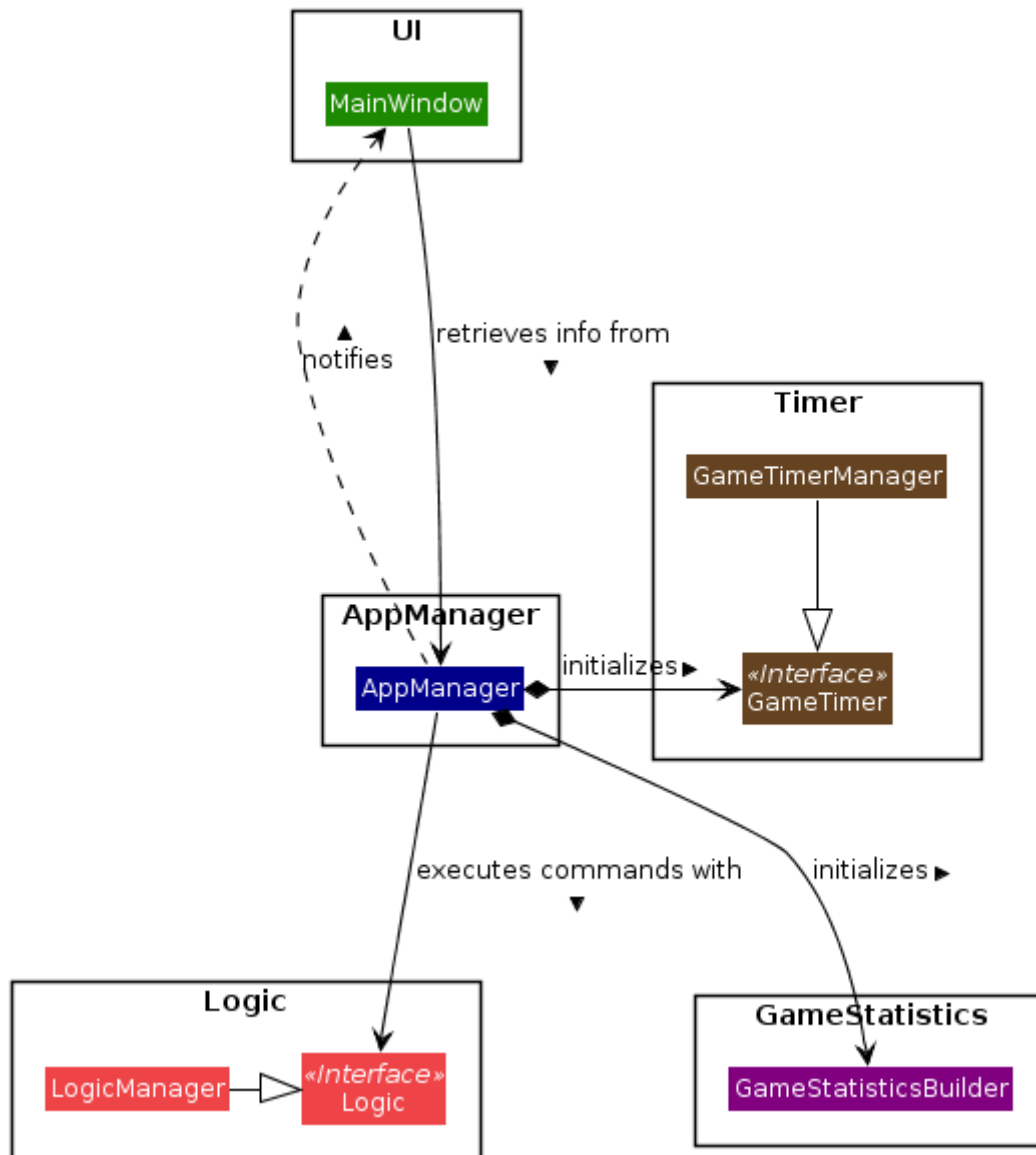
Figure 4. Structure of the AppManager Component

The `AppManager` component serves as a *Facade* layer and communication hub between the internal components of *Dukemon* and the *UI* components. Using this extra layer provides better abstraction between the *UI* and the internal components, especially between the `Timer` and the `UI`.

`AppManager` communicates with both the `Logic` and `Timer` components to send feedback to the `UI` to display back to the user.

- Gets feedback for commands by through `Logic`
- Starts and Stops the `Timer` when required.
- Makes call-backs to the `UI` to update various `UI` components.
- Initiates collection of `Statistics` by pulling data (eg. Time Elapsed) from `Timer` and `Logic`.

2.4. Timer component
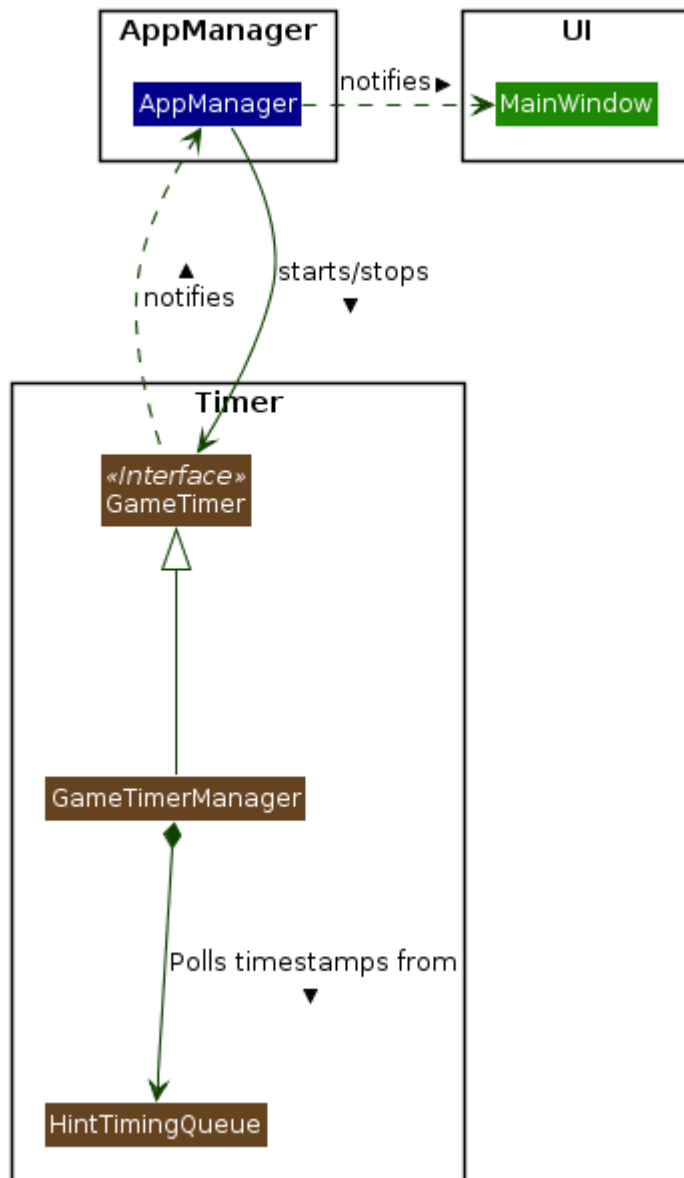
Figure 5. Structure of the Timer Component

API : `GameTimer.java`

The `Timer` consists of a `GameTimer` that will keep track of time elapsed via an internal countdown timer and notify the `AppManager`, who will notify the `UI` components.

- Dealing with the internal countdown timer that runs during a game session.
- Periodically triggering *callbacks* that will notify the `AppManager` component.
- Gets timestamps to trigger `Hints` via a `HintTimingQueue`

Due to the fact that the `Timer` *has to work closely with the* `UI` *and* `AppManager` *(without being coupled directly), it is separated from the* `Logic`, `Model` *and* `Game` *components.*

2.5. Logic component

This section breakdown the logic package into its internal components

Figure 6. Structure of the Logic Component

Logic is primarily built by two segments: Command and Parser.

Command

> Command is an abstract class.

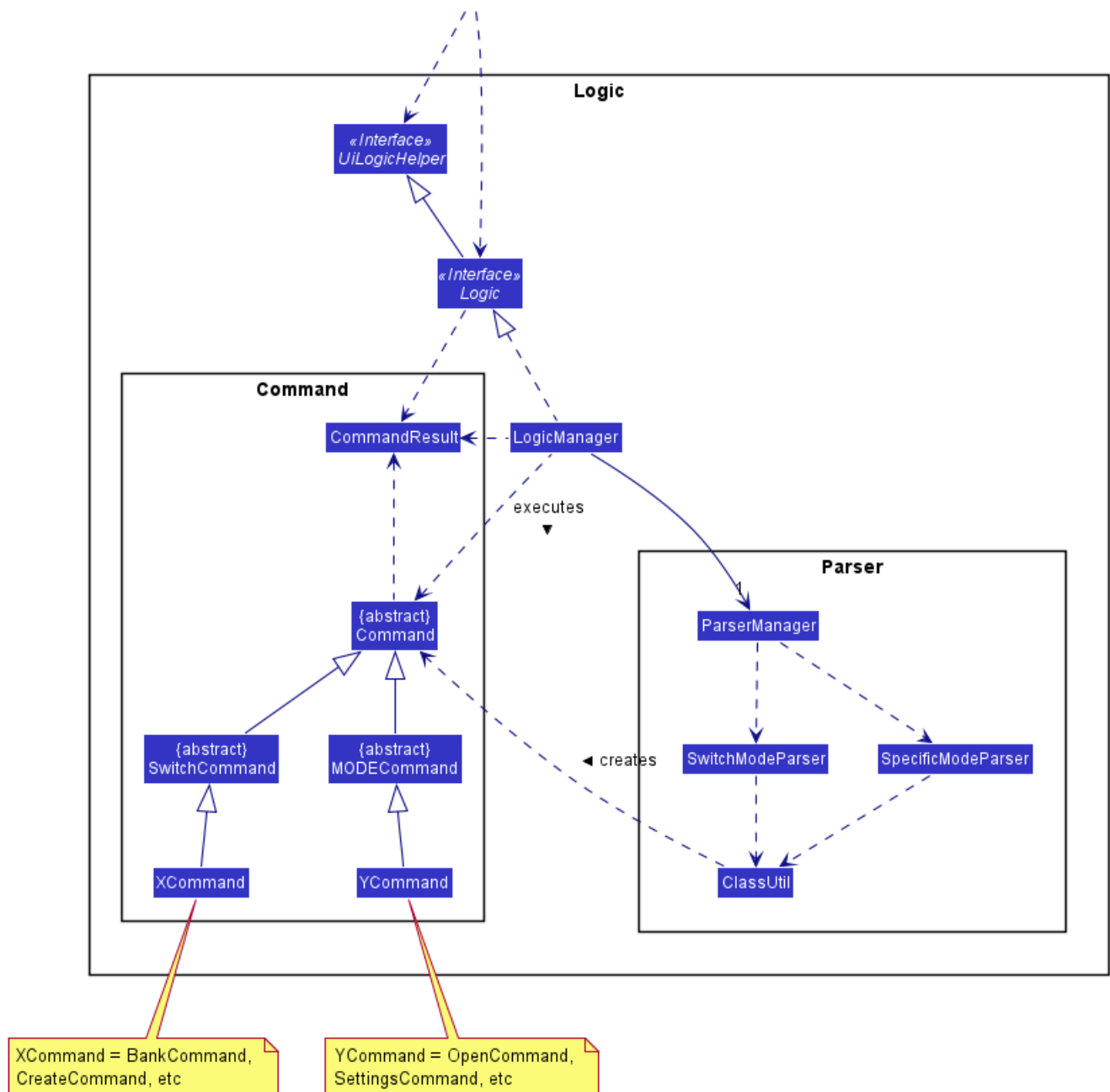> Four other abstract classes (HomeCommand, OpenCommand, GameCommand and SettingsCommand) extend Command.

> Concrete Command classes with an execute method implementation extend one of the above four abstract classes.

Parser

> ParserManager holds reference to a SpecificModeParser and a SwitchModeParser.

> The SpecificModeParser changes based on current application mode.

> Both of them hold references to all concrete Parser and Command Classes with the help of ClassUtil

Logic fulfils its contracts with other packages through two interfaces: Logic and UiLogicHelper

Examples of transactions promised by Logic API include command execution, command result and update statistics.

UiLogicHelper APIs is a subset of Logic APIs and only contains transactions for AutoComplete. It exposes the functionalities through the following getter methods:

- `List<AutoFillAction>#getMenuItems(String text)` — Gets an List of AutoFillActions to fill up AutoComplete display based on current user input given in text

- `ModeEnum#getMode()` — Retrieves the application mode to display visually to the user (represented by enumeration object ModeEnum)

- `List<ModeEnum>#getModes()` — Retrieves the possible modes the user can transition to from current mode

API : `Logic.java UiLogicHelper.java`
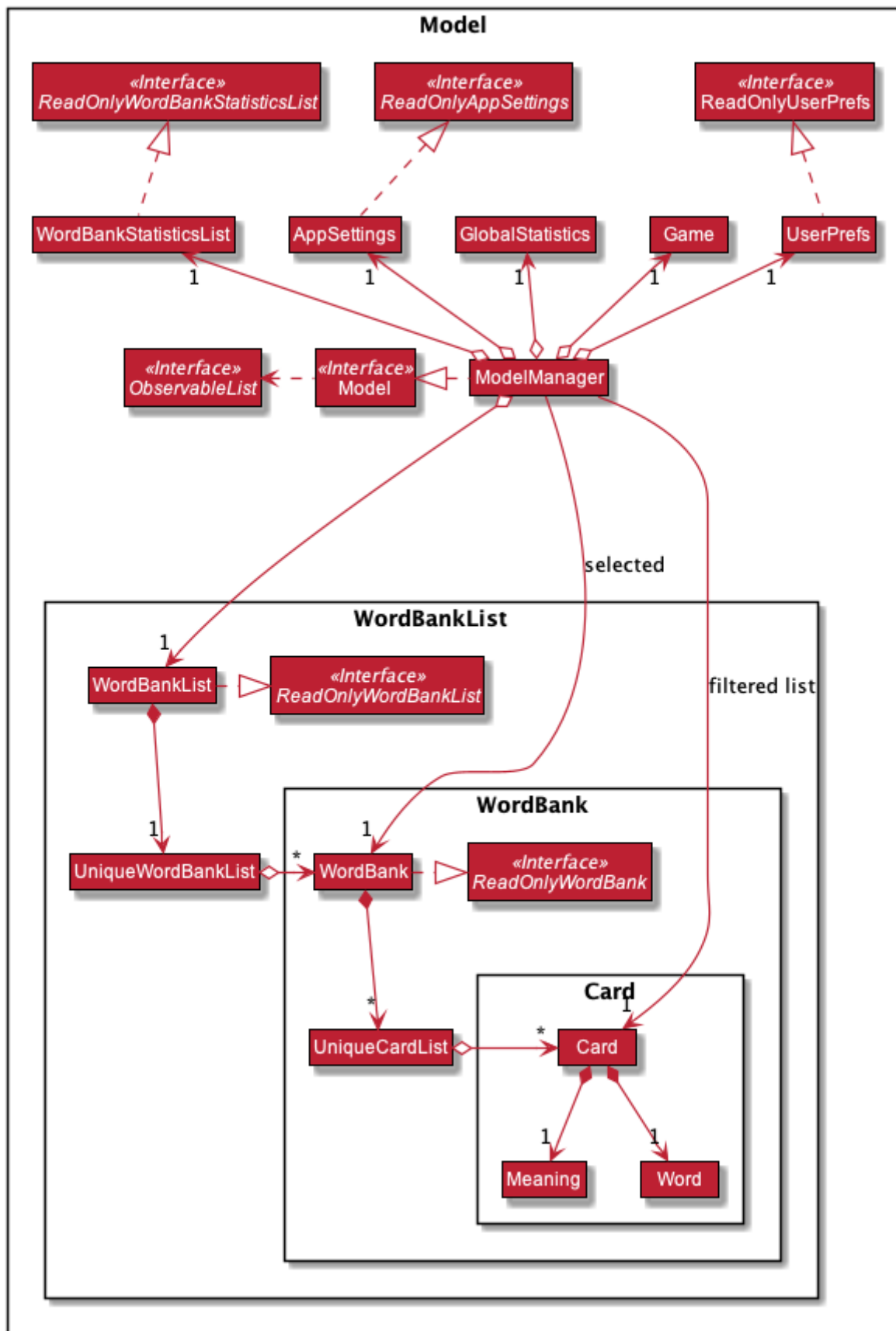
2.6. Model component

Figure 7. Structure of the Model Component

API : `Model.java`

The `Model`,

- contains information that the game requires at run time. They include: `WordBankList`, `WordBankStatisticsList`, `GlobalStatistics`, `Game`, `AppSettings`, `UserPrefs`.

- does not depend on any of the higher level components. i.e. `Ui`, `Timer`, `AppManager`, `Logic`, `Storage`.

- has a direct reference to a user selected `WordBank`.

- exposes an unmodifiable `ObservableList<Card>` 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.
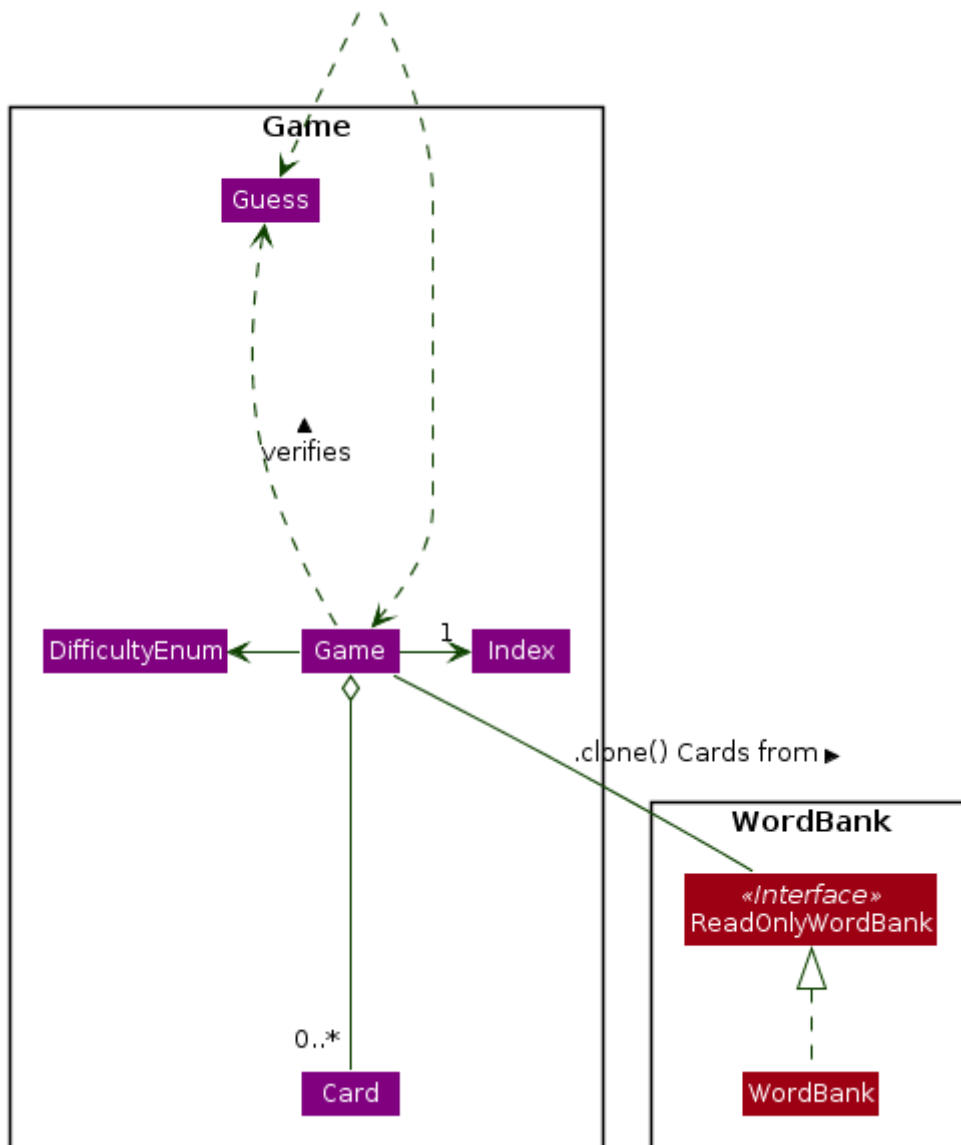
2.7. Game component

Figure 8. Structure of the Game Component

The `Game` component,

- stores a *shuffled* `List<Card>` that is cloned/copied from a `ReadOnlyWordBank`.
- maintains an `Index` to keep track of the state of the game.
- has an associated `DifficultyEnum` that dictates the time allowed for each question.

- verifies `Guess` that are sent by `Logic` (User's guesses)

2.8. Storage component
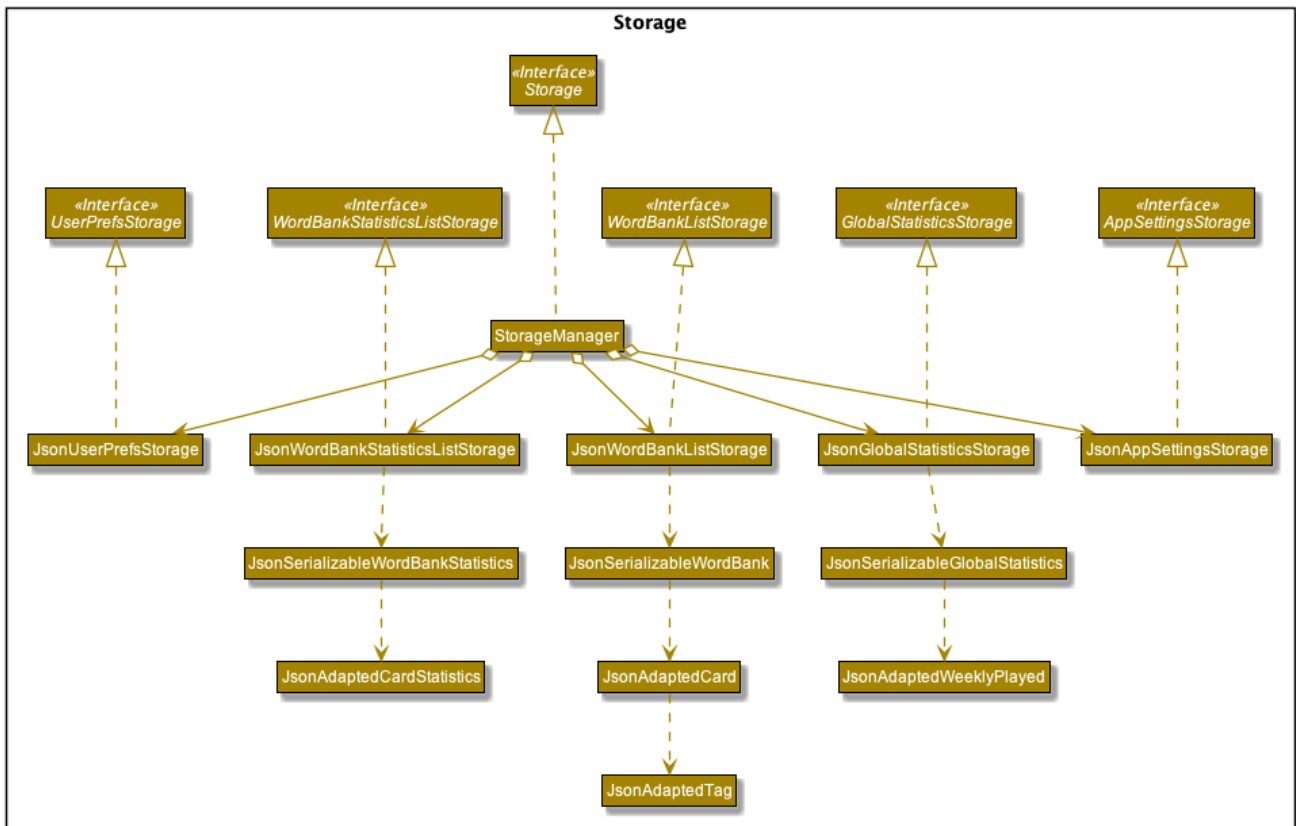
Figure 9. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

- contains multiple types of distinct storage system.
- does not depend on any of the higher level components. i.e. `Ui`, `Timer`, `AppManager`, `Logic`, `Model`.
- handles function calls directly to the computer's system.
- can save data objects in json format and read it back.

2.9. Statistics component

The Statistics component includes 2 main subcomponents:

- A `GlobalStatistics`, containing the user's total number of games played and the number of games played in the current week.
- A `WordBankStatisticsList`, which is a collection of `WordBankStatistics`, one for each `WordBank`.

The class diagram of the Statistics component is shown below:

Figure 10. Statistics class diagram.

2.10. Common classes

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

3. Implementation

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

3.1. <u>AutoComplete and Command Execution</u>
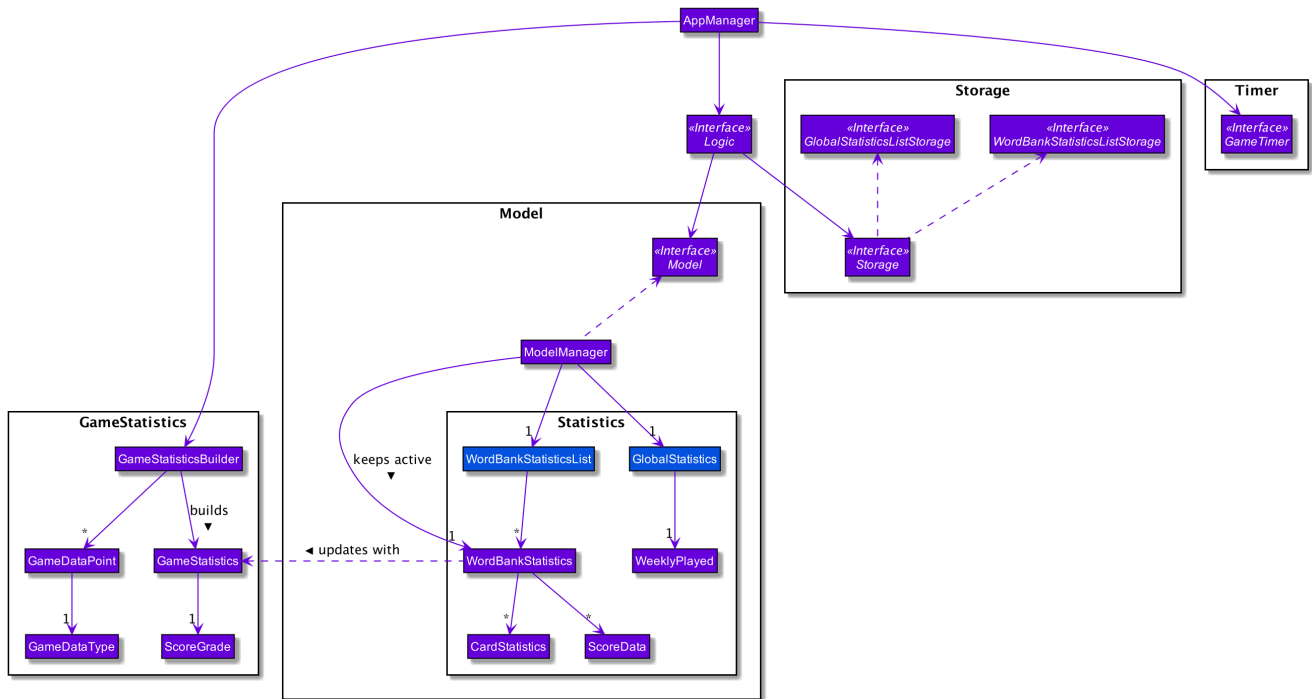
This section explains how the design choice of Dynamic Parsers fulfils AutoComplete and Command Execution.

ParserManager dynamically changes parser depending on current mode the game is at. When updating the User Interface for every keystroke, it ensures only the right commands get parsed and autocompleted at each moment.

3.1.1. Implementation details

1. `ParserManager` instance has reference to a `SwitchModeParser` and `SpecificModeParser`

2. When user enters a keystroke, the `SwitchModeParser` and/or `SpecificModeParser` are accessed based on internal state.

3. It updates the AutoComplete suggestions for every keystroke.

4. Internal State consists of booleans: gameIsOver, bankLoaded and enumeration ModeEnum: HOME, OPEN, GAME, SETTINGS

5. The above process is aided by `ClassUtil` to handle instantiation of Parser and Command objects.

> The state management is complex. The below sequence diagram demonstrates all possible workflows.
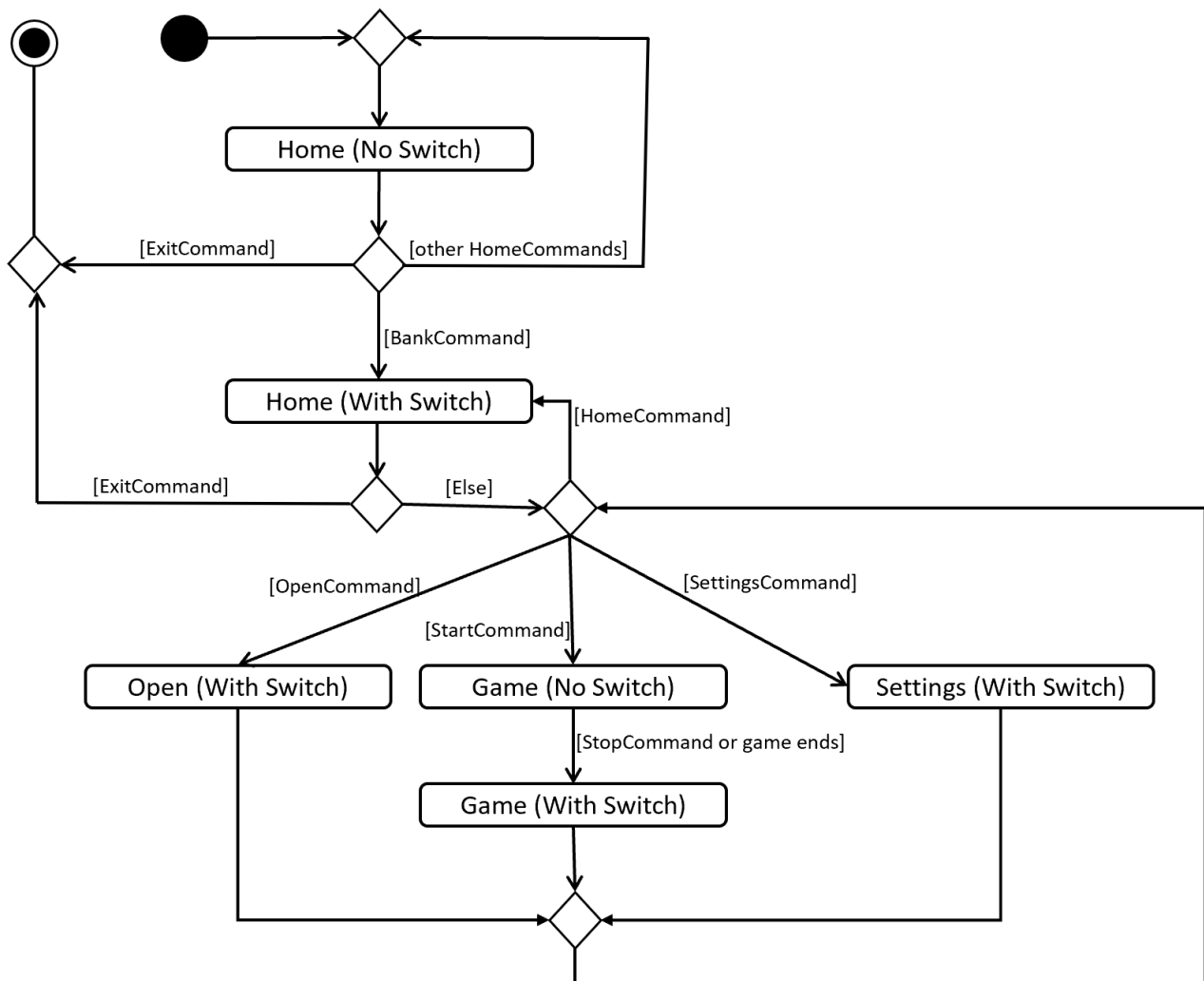
Figure 11. Activity diagram of Application flow

NOTE | Home (No Switch) means HomeModeParser is used and SwitchModeParser is not used

3.1.2. Walkthrough of Command Execution

- Command Execution through `Logic Interface`

  1. A String from Ui package gets to `ParserManager` and gets converted into a `Command` object which is executed by the `LogicManager`.

  2. The command execution can affect the `Model` (e.g. adding a word meaning pair into wordbank).

  3. The result of the command execution is encapsulated as a `CommandResult` object which is passed back to the `Ui` and `AppManager`.

  4. In addition, the `CommandResult` object can also instruct the `Ui` to perform certain actions, such as displaying help to the user.

3.1.3. Walkthrough of AutoComplete

- AutoComplete through `UiLogicHelper Interface`

The following sequence diagram shows how the AutoComplete operation runs when user keys in "st" into command box.
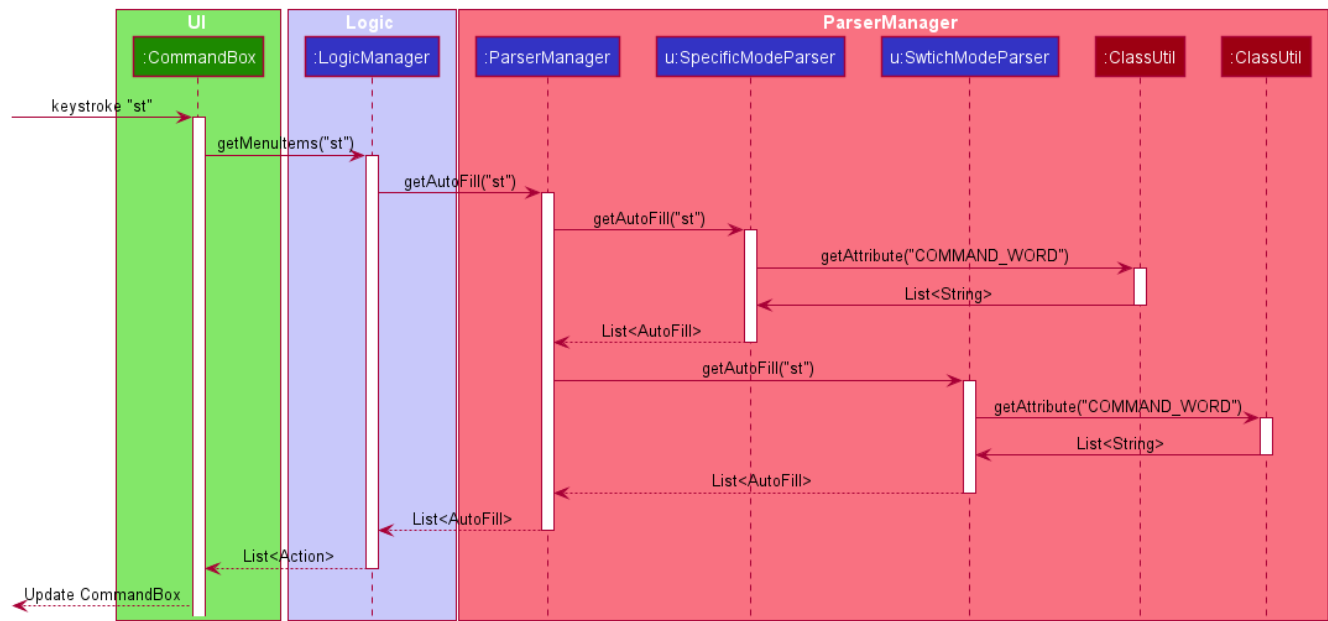
Figure 12. Sequence Diagram of AutoComplete

3.1.4. Design Considerations

	Alternative 1	Alternative 2
Aspect 1: How parser and command objects are instantiated in SpecificModeParser	**Use java reflections to hold a List of Classes and iterate through them to pick the matching Classes** *Pros:* Open Close Principle strictly followed. Adding a command with a parser takes only one line of code. *Cons:* It is developer responsibility to ensure classes subclass Command object as compile time errors would not be thrown.	**Use switches in Parser to match Command Word to create Parser objects if necessary else directly create Command object.** *Pros:* Compile time error would be thrown if new command or parser does not subclass correctly. *Cons:* Adding a new command with parser would require the developer to insert it into multiple locations as the autocomplete feature needs an iterable command list.

Why did we choose Alternative 1: d		
Aspect 2: Single Parser vs Parser Manager	**Using a ParserManager to dynamically switch between Parsers based on current state** *Pros:* Commands not belonging to specific mode would not be parsed *Cons:* More code to write for initial developer. However it is easily extnensible for new modes and parsers by the Open Close Principle	**Use a single parser** *Pros* We do not need to restructure the logic package. *Cons* Bad user experience as it autocompletes and parses commands that do not belong to a particular mode.
Why did we choose Alternative 1: d		

3.1.5. Technical details

<Insert class util code demonstrating java reflections>

<Insert where one line of code can be inserted for adding new commands and parsers>

3.2. <u>Settings Feature</u>

3.2.1. Implementation

`AppSettings` was a class that was created to be integrated into the `Model` of the app. It currently contains these functionalities:

- `difficulty [EASY/MEDIUM/HARD]` to change the difficulty of the game.

- `hints [ON/OFF]` to turn hints on or off.

- `theme [DARK/LIGHT]` to change the theme of the app. Currently only supporting dark and light themes.

This feature provides the user an interface to make their own changes to the state of the machine. The settings set by the user will also be saved to a `.json` file under `data/appsettings.json`.

The activity diagram below summarizes what happens in the execution of a settings command:
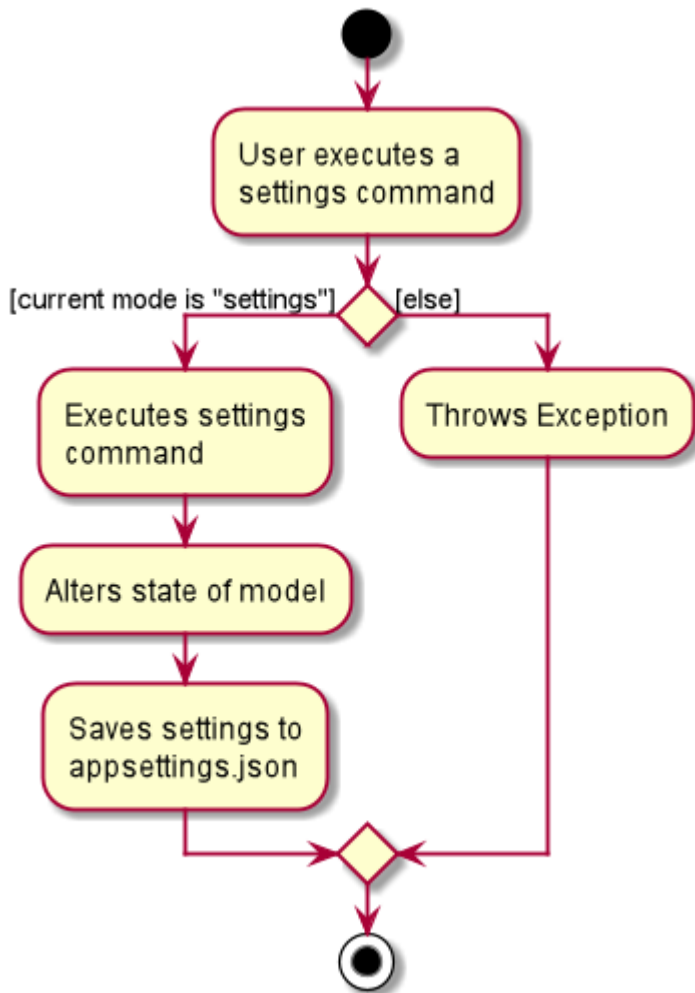
Figure 13. Activity diagram of the execution of a settings command.

NOTE	Take note that "mode" as defined in our project is the state in which the application is able to take commands specific to that mode.

Given below is a step by step walk-through of what happens when a user executes a difficulty command while in settings mode:

Figure 14. Before state of application.

Step 1:
Let us assume that the current difficulty of the application is "EASY". The object diagram above shows the current state of `AppSettings`.
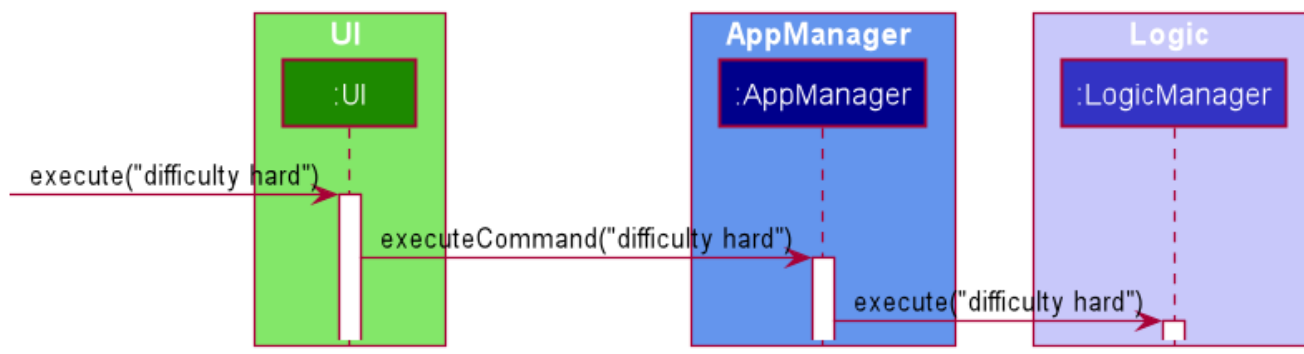
Figure 15. Sequence diagram of Step 2.

Step 2:
When the user enters `difficulty hard`, the command gets passed into Ui first, which executes `AppManager#execute()`, which passes straight to `LogicManager#execute()` without any logic conditions to determine its execution path.
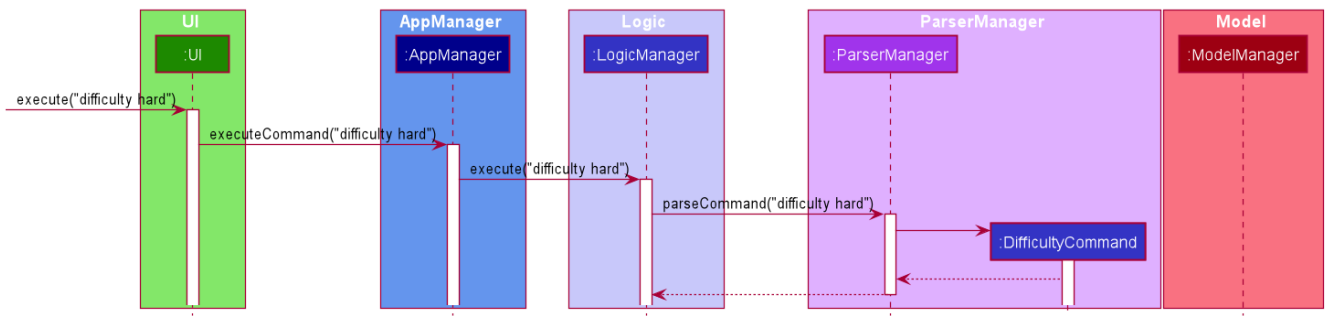
Figure 16. Sequence diagram of Step 3.

Step 3:

At `LogicManager#execute()` however, the command gets passed into a parser manager which filters out the `DifficultyCommand` as a non-switch command and it creates a `DifficultyCommand` to be executed.
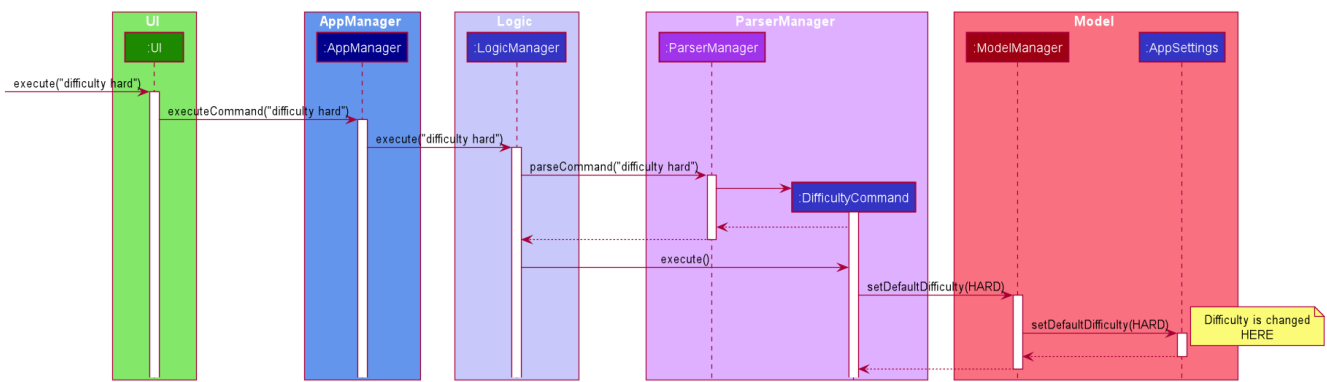
Figure 17. Sequence diagram of Step 4.

Step 4:

Upon execution of the `DifficultyCommand`, the state of the model is changed such that the `DifficultyEnum` in `AppSettings` is now set to `HARD`.
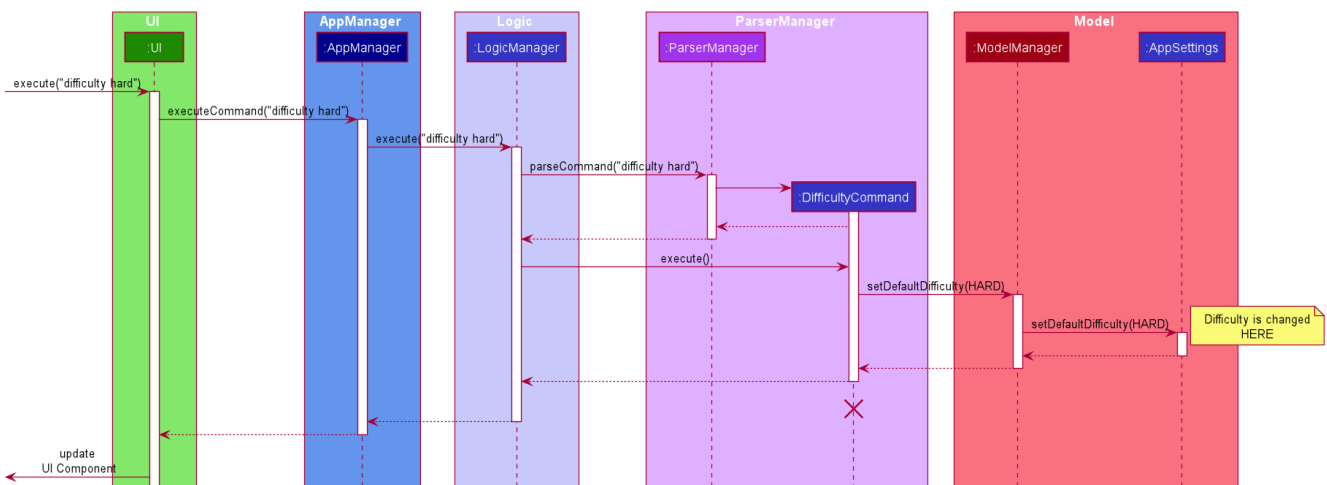
Figure 18. Sequence diagram of Step 5.

Step 5:

Since the main function of the `difficulty` command is accomplished and all that is left is to update the ui, the `CommandResult` that is produced by the execution of the command goes back to `Ui` without much problem.
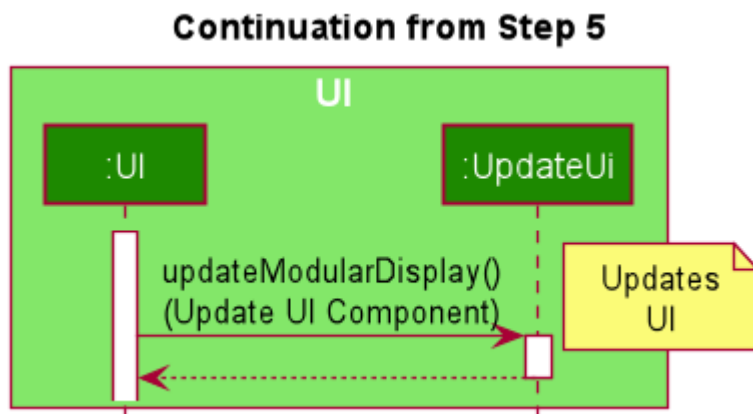
Figure 19. Sequence diagram of Step 6.

Step 6:

Assuming that there were no errors thrown during the execution of the `difficulty` command, the execution calls `updateModularDisplay` in `UpdateUi`. In here, the `ModeEnum.SETTINGS` is registered and it updates the settings display to properly reflect the change in difficulty.

The state of appSettings is then as follows:

Figure 20. After state of application

3.2.2. Design Considerations

There were a few considerations for implementing an interface that essentially allows users to touch a lot of parts of the application through settings and some of these methods break software design principles. These are the considerations we came across:

	Alternative 1	Alternative 2

Aspect 1: Where to effect change when a setting is changed by the user	Effecting the change inside the `execute()` command of the settings commands: *Pros:* Since the Command is taking care of all the execution, there is no need to worry about extra implementation of the settings' effects in their classes. *Cons:* However, there are certain situations that will break software design principles, such as the **Single Responsibility Principle** by doing the job of already existing classes.	Effecting the change in the part of the architecture that the setting is affecting. E.g, Changing the theme inside Ui or changing the difficulty inside model *Pros:* This method practises good software engineering principles and it abides by the architecture diagram shown above as to where the changes of the settings are being effected. *Cons:* This method however requires that the reader gets familiar with the whole architecture diagram as they need to know where to implement the actual change in settings as opposed to creating a new class that performs the same functionality of an existing class.

Why did we choose Alternative 2:
We believe that software design principles exist for a reason. Furthermore, while alternative 1 may seem a lot simpler, Alternative 2 allows for extension just by adding new methods and refrains the user from having to extensively rework the structure of the application in order to add a new setting.

Aspect 2: How to store information regarding the different settings	Storing it inside the enumerations that make up the choices for the settings	Storing it inside the classes that implement the settings
	Pros: Having the information stored inside the enum allows for immutablilty, such that no other class can change the properties of the enums. Only the developer can change the values of the enums and it will subsequently affect all the methods and functionality that relies on said enum. **Cons:** In the case that the user wants to customise certain continuous settings such as time limit, they are unable to as those settings are already defined by the developer to be discrete options.	**Pros** The information is easily accessible from within the class itself and there is no need for extra import classes to handle the enums in alternative 1. **Cons** Unlike Alternative 1, the developer can create an extension to the class implementing the setting to allow the user to customise their settings even further, allowing for continuous values to be used rather than discrete values.

Why did we choose Alternative 1:
The considerations for this aspect was mainly down to how much customisability we wanted to grant our users. While having more customisability is better in some cases, in this one, we do not think the added functionality of allowing the user to extensively customise their experience with our application to be particularly impactful not necessary. Moreover, alternative 2 makes for a less organised code base and we wanted to avoid that as much as possible.

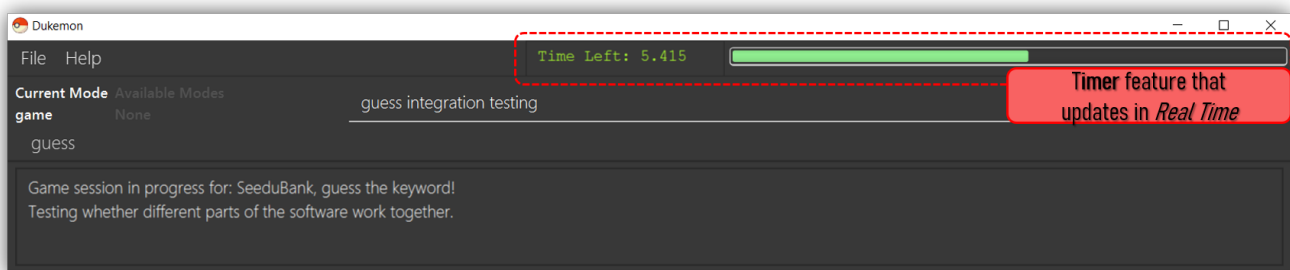## 3.3. Timer-based Features

Figure 21. Screenshot of the `Timer` component in action.

3.3.1. Implementation Overview

The `Timer` component utilizes the `java.util.Timer` API to simulate a stopwatch that runs for each `Card` in a `Game`. It also relies on using *Functional Interfaces* as *callbacks* to periodically notify other

components in the system. Using *callbacks* allows the `Timer` to enact changes in other components of the system without directly holding a reference to those components.

Internally, the `Timer` works by using the method `java.util.Timer.schedule()` that will schedule `java.util.TimerTasks` at a fixed rate (*every 1ms*).

An *Observer Pattern* is loosly followed between the `Timer` and the other components. As opposed to defining an *Observable* interface, the `AppManager` simply passes in *method pointers* into the `Timer` to *callback* when an event is triggered. The `AppManager` thus works closely with the `Timer` as the main hub to enact changes based on signals given by the `Timer`.

NOTE	To avoid synchronization issues with the `UI` component, all `TimerTasks` (such as requesting to refresh a component of the `UI`) are forced to run on the **JavaFX Application Thread** using `Platform.runLater()`.
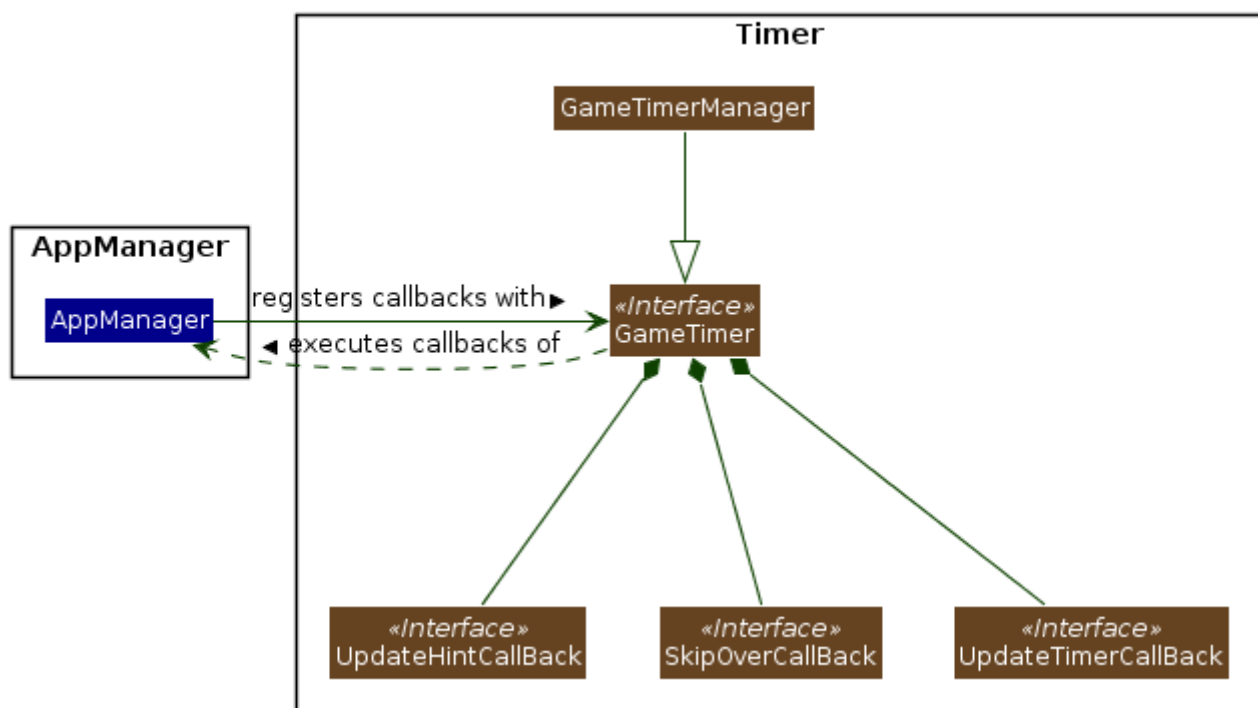
Figure 22. Class diagram reflecting how the callback-functions are organized in the Timer component.

The three main events that are currently triggered by the `Timer` component which require a *callback* are:

1. Time has elapsed, *callback* to `AppManager` to **update and display the new timestamp** on the `UI`.

2. Time has run out (*reached zero*), *callback* to `AppManager` to **skip over** to next `Card`.

3. Time has reached a point where `Hints` are to be given to the User, *callback* to `AppManager` to **retrieve a `Hint` and display** accordingly on the `UI`.

The *callbacks* for each of these events are implemented as nested *Functional Interfaces* within the `GameTimer` interface, which is implemented by the `GameTimerManager`.

3.3.2. Flow of Events - `Hints` Disabled

This section describes the general sequence of events in the life cycle of a single `GameTimer` object with **no hints**.
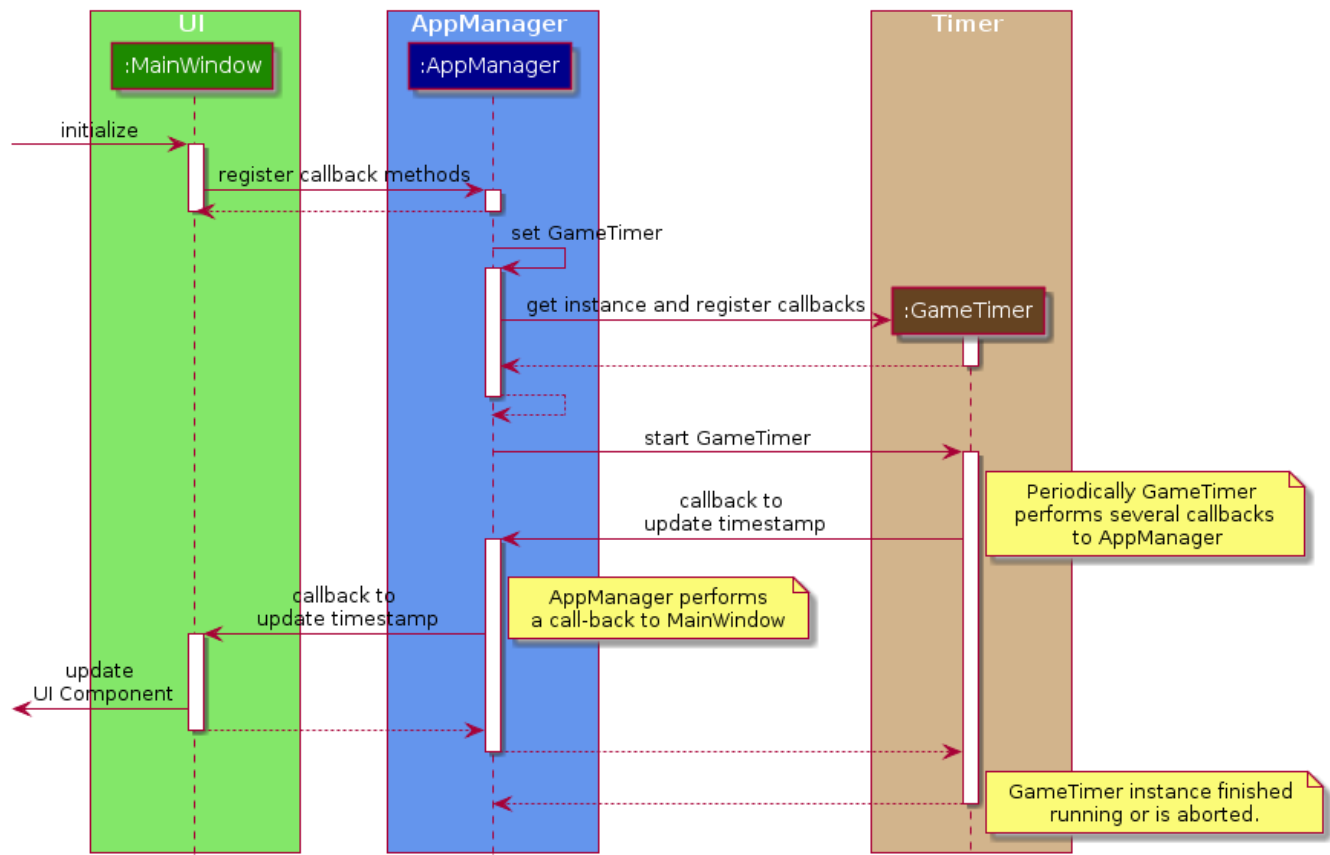
Figure 23. Sequence diagram (with some details omitted) describing the flow of registering and executing callbacks between the different components

NOTE	`GameTimer` interface uses a factory method to create `GameTimerManager` instances. This behavior is omitted in the above diagram for simplicity.

1. `UI` component first registers *callbacks* with the `AppManager`.
2. When a *Game* is started, `AppManager` initializes a `GameTimer` instance for the first *Card*.
3. `AppManager` registers *callbacks* with the `GameTimer` component.
4. `AppManager` starts the `GameTimer`.
5. Periodically, the `GameTimer` notifies the `AppManager` to update the `UI` accordingly.
6. `AppManager` is notified by `GameTimer`, and then notifies `UI` to actually trigger the `UI` change.
7. `GameTimer` finishes counting down (or is **aborted**).
8. `AppManager` repeats Steps 2 to 7 for each *Card* while the *Game* has **not** ended.

Using this approach of *callbacks* provides **better abstraction** between the `UI` and `Timer`.

NOTE	A new `GameTimer` instance is created by the `AppManager` for every `Card` of a `Game`. The `AppManager` provides information regarding the duration in which the `GameTimer` should run for, and whether `Hints` are enabled.

3.3.3. Flow of Events - `Hints` Enabled
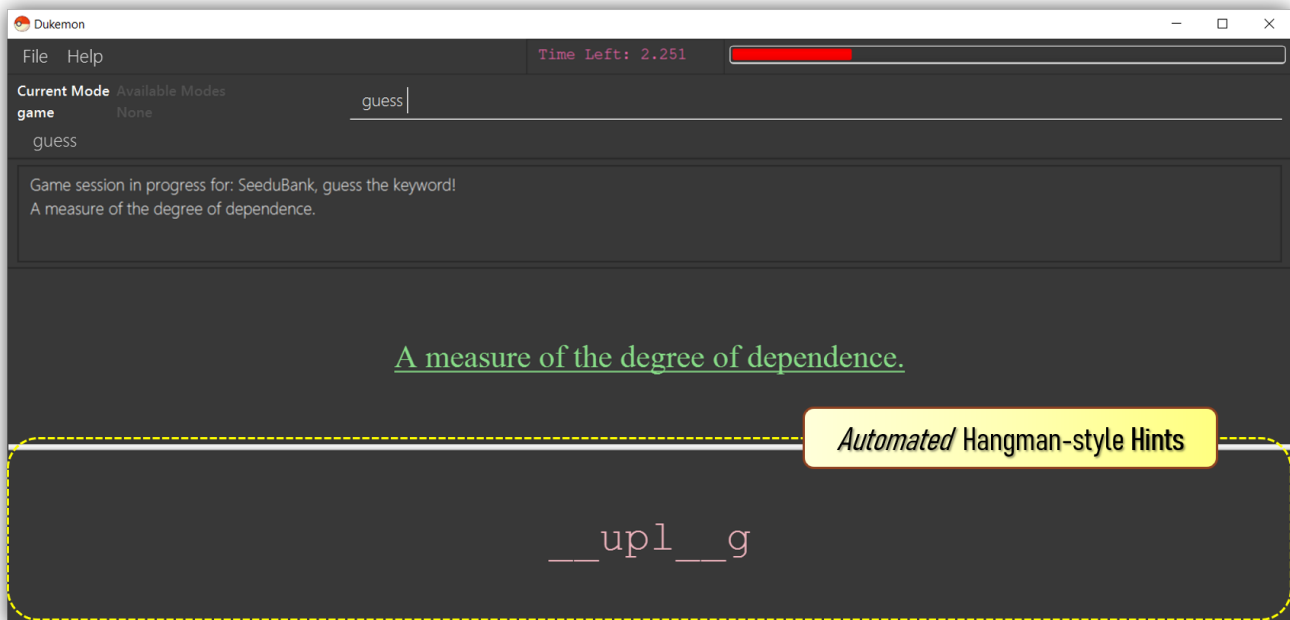
Figure 24. Screenshot of the automatic `Hints` feature in action.
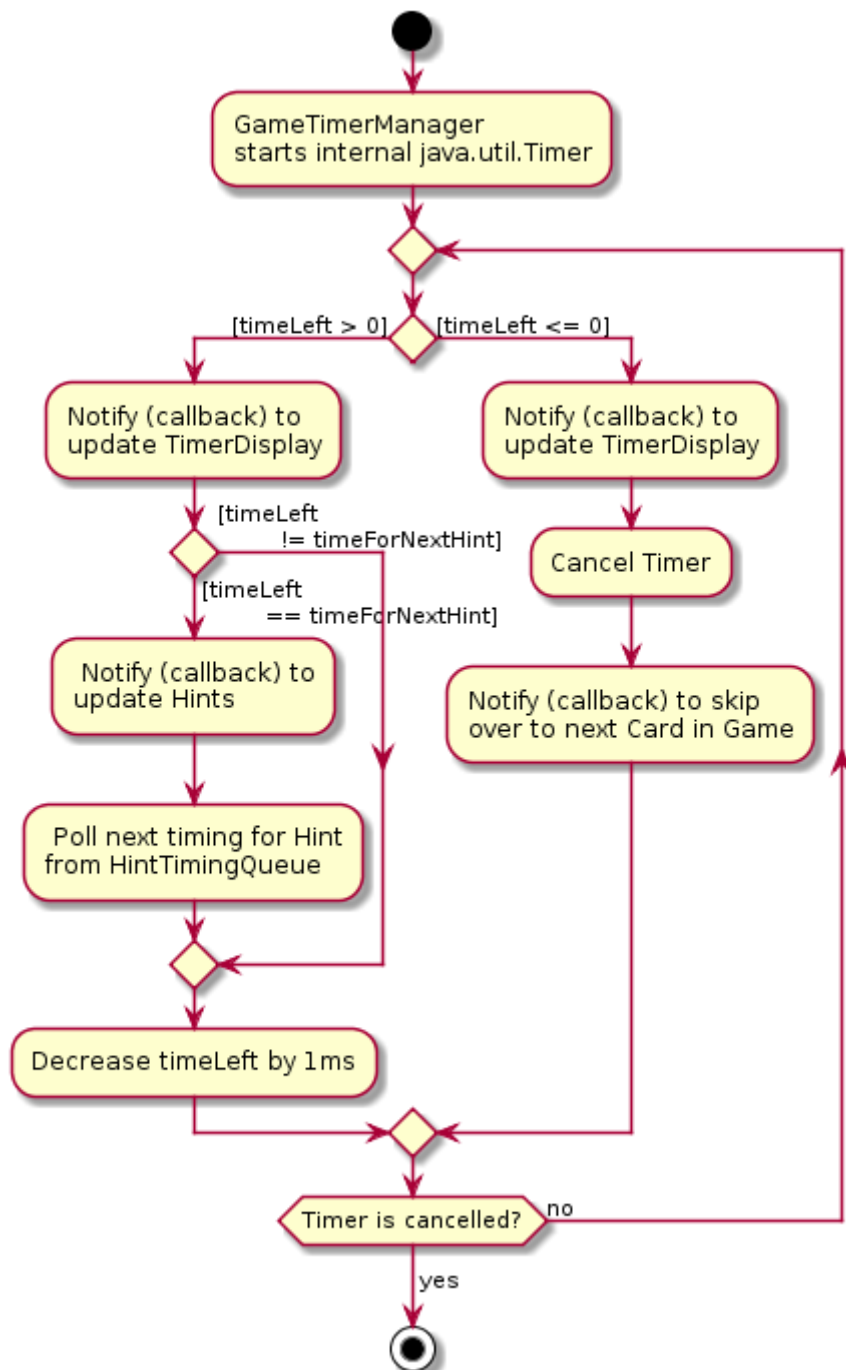
Figure 25. Activity diagram of the `run()` *method of an instance of* `GameTimerManager` *when* `Hints` *are enabled.*

The behavior of `Timer` when `Hints` are enabled is **largely still the same.**

When `Hints` are enabled, `AppManager` initializes a `HintTimingQueue` in the `GameTimer` for each *Card*. `HintTimingQueue` is a class that contains a `java.util.Queue` of *timestamps* (in milliseconds). `GameTimer` polls from the `HintTimingQueue` and checks against these polled *timestamps* to update the `Hints` provided periodically.

The described activity is visualized via the activity diagram as **shown above**. The internal `Timer` is started when `GameTimerManager` calls the `.schedule()` method of its internal `java.util.Timer`, which schedules `TimerTasks` immediately, every millisecond until the `java.util.Timer` is cancelled. The field `timeLeft` is initialized to be the amount of time allowed per *Card* (in milliseconds), and is updated every 1ms.

3.3.4. Design Considerations

There were a few considerations for designing the `Timer` this way.

	Alternative 1	Alternative 2
Aspect 1: Where and How to effect changes to the `Ui` and other components when the `Timer` triggers an event.	**Holding a reference to `Ui` and other components directly inside `GameTimer` itself**: *Pros:* Straightforward and direct, can perform many different tasks on the dependent components. *Cons:* Poor abstraction and high potential for cyclic dependencies, resulting in high coupling.	**Using *Functional Interfaces* as Call-backs to notify components indirectly.** *Pros:* Maintains abstraction and minimal coupling between `Timer` and other components *Cons:* Relies on developer to register correct call-back methods with the `Timer`. Different actions need to be implemented as different call-backs separately. Possible overhead in performing few levels of call-backs.
Why did we choose Alternative 2: To ensure better extendability of our code for future expansion, we felt it was important to maintain as much abstraction between components. This is also to make life easier when there comes a need to debug and resolve problems in the code.		

3.4. <u>Word Bank Management</u>

This section discusses the implementation of Word Bank Management in varied levels of detail. This can be split into four complimentary distinct sections.

They are:

- Word bank's data structure and it's storage system
- User Commands
- Drag and drop
- Revision word bank

3.4.1. Word bank's data structure and it's storage system

Allows developers to use and extend this architecture to streamline their feature implementation. Allows user to save and load their word banks.

Observe closely the attributes and methods of the following class diagrams.
They describe and explain word bank's data structure and storage in detail.
Each of them are implemented with close examination so that it is easily usable and extended upon by other teammates/complimentary features.

We start from the lowest level - `Card`.
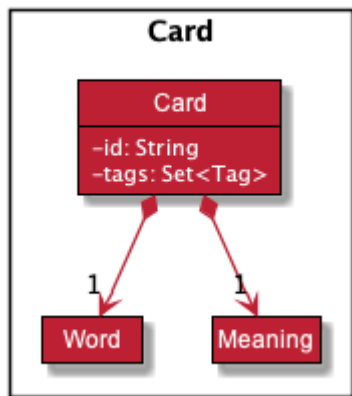
Figure 26. Class diagram of `Card`.

A `Card` contains a unique `id`, a `word`, a unique `meaning`, a set of `tags`.

`id` : for statistical tracking
`word`: answer to the question (meaning)
`meaning`: the question that will appear in the game
`tags`: optional tags to classify cards

> **NOTE** Cards with the same meaning are duplicates, and is disallowed.

Now the second level - `WordBank`
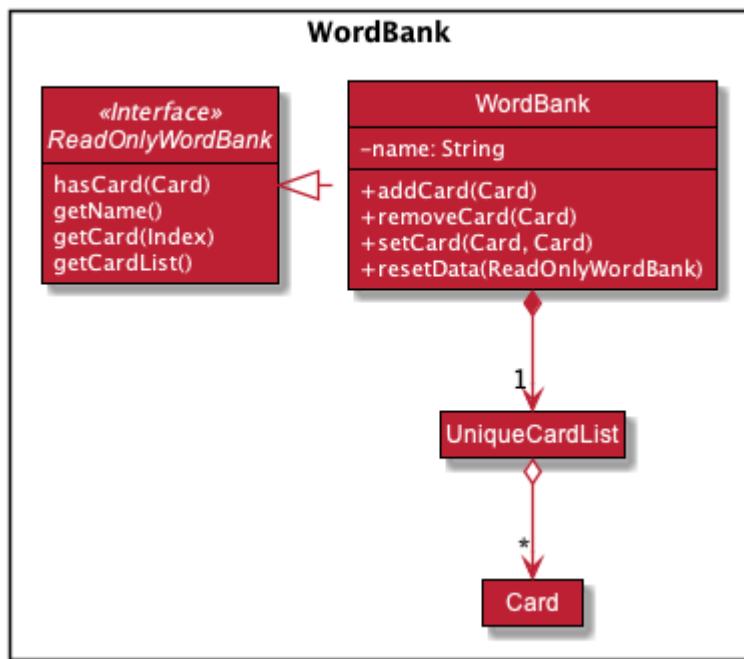
Figure 27. Class diagram of `WordBank`.

A `WordBank` contains a `UniqueCardList` and a unique `name`.

`UniqueCardList` : prevent duplicate cards
`name`: unique name of the word bank

NOTE	`WordBank` exposes an unmodifiable `ObservableList<Card>` that can be 'observed'. The UI can be bound to this list so that the UI automatically updates when the cards in the list change. Word banks with the same name are duplicates, and is disallowed.

Now the third level - `WordBankList`
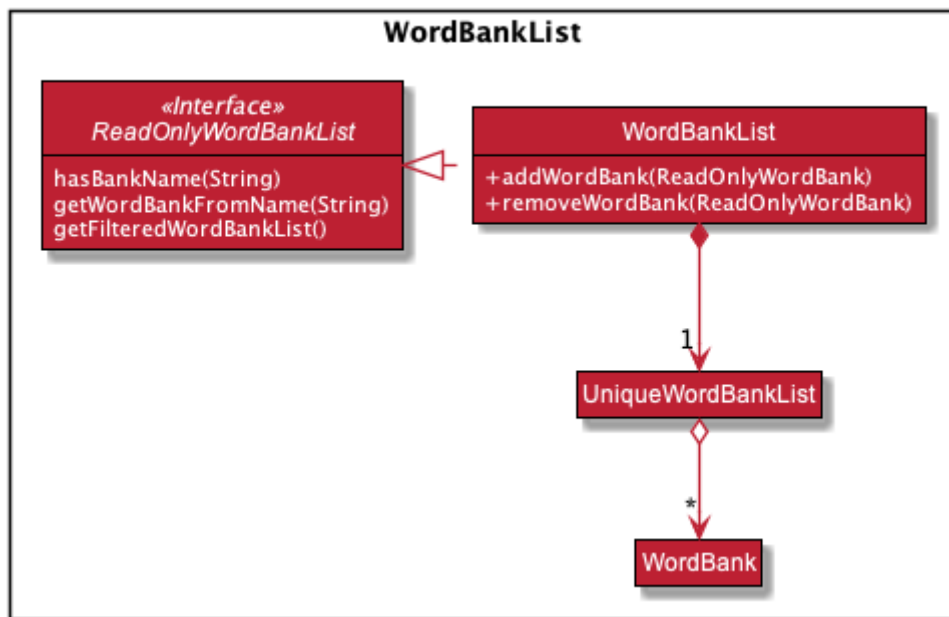
Figure 28. Class diagram of `WordBankList`.

A `WordBankList` contains a `UniqueWordBankList`.

`UniqueWordBankList` : prevent duplicate word banks

NOTE	`WordBank` exposes an unmodifiable `ObservableList<WordBank>` that can be 'observed'. The UI can be bound to this list so that the UI automatically updates when the cards in the list change.

In Dukemon, there is should only be one `WordBankList`, which is created upon `Storage` initialisation. `Model` holds a reference to that specific `WordBankList`.

Architecture overview - `WordBankList`
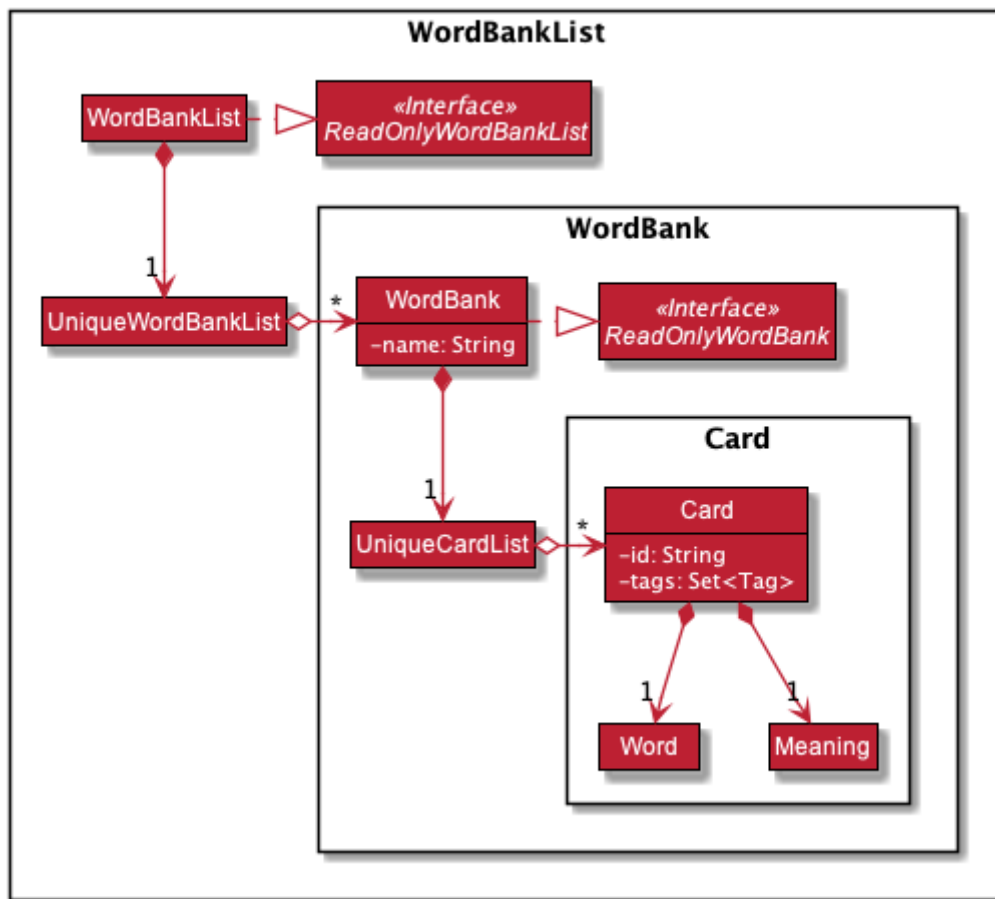
Figure 29. Overview of `WordBankList`.

Now the integration with storage - How `WordBankList` is stored in `Model` and `Storage`.
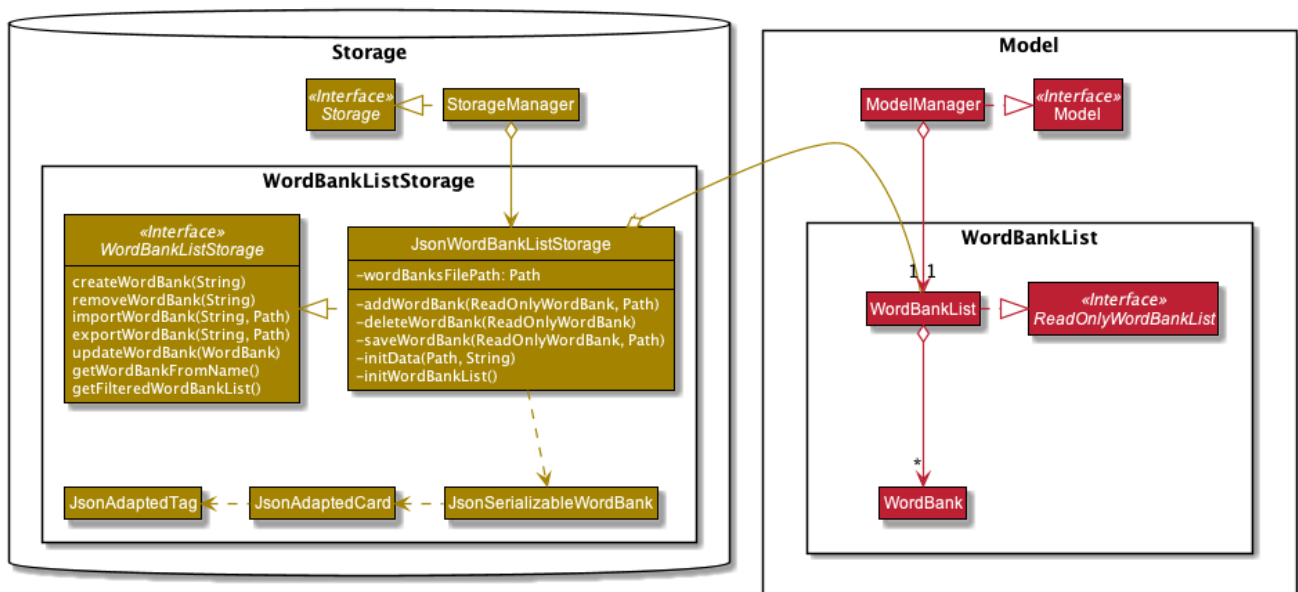
Figure 30. Integration of `WordBankList` *within* `Storage` *and* `Model`.

WordBankListStorage consists of robust and self-explanatory methods in which users can use and extend upon easily.

Alongside with word bank's data structure, they lay the foundation for my other complementary

sections of Word Bank Management.

On top of that, they serve as a essential foundation for Dukemon. As such, these data structures and methods were required by my teammates, to build their own feature implementations. (Statistics, Game, Settings)

3.4.2. User Commands

Allows user to customise his cards and group them according to topics (word banks).
User commands edits and manipulates `Cards` and `WordBanks` heavily.

NOTE	As mentioned previously, user commands will extend and utilise word bank's data structure and storage heavily. You can refer to it to enhance your understanding of this implementation.

Let me first introduce you how these commands are implemented and structured in `Logic`.
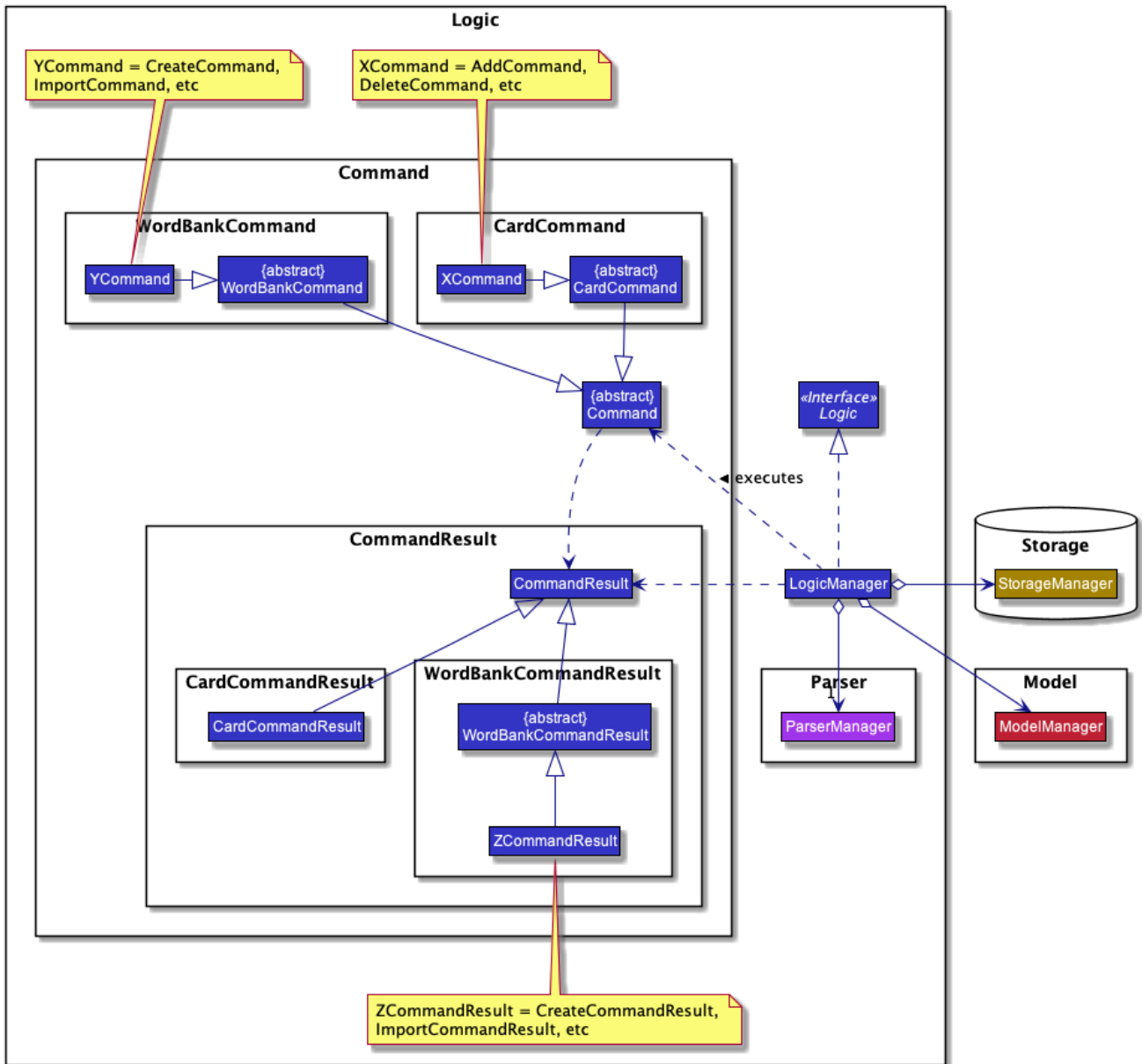
Figure 31. Overview class diagram of `Logic` *with emphasis on CardCommands and WordBankCommands.*

Commands reside in `Logic`. They work on `Model` and `Storage` through `Logic`.
To segregate cards according to their function, we distinguished the following:

`CardCommands` work on `Cards`.

`WordBankCommands` work on `WordBanks`.

Walkthrough - `ImportCommand`.
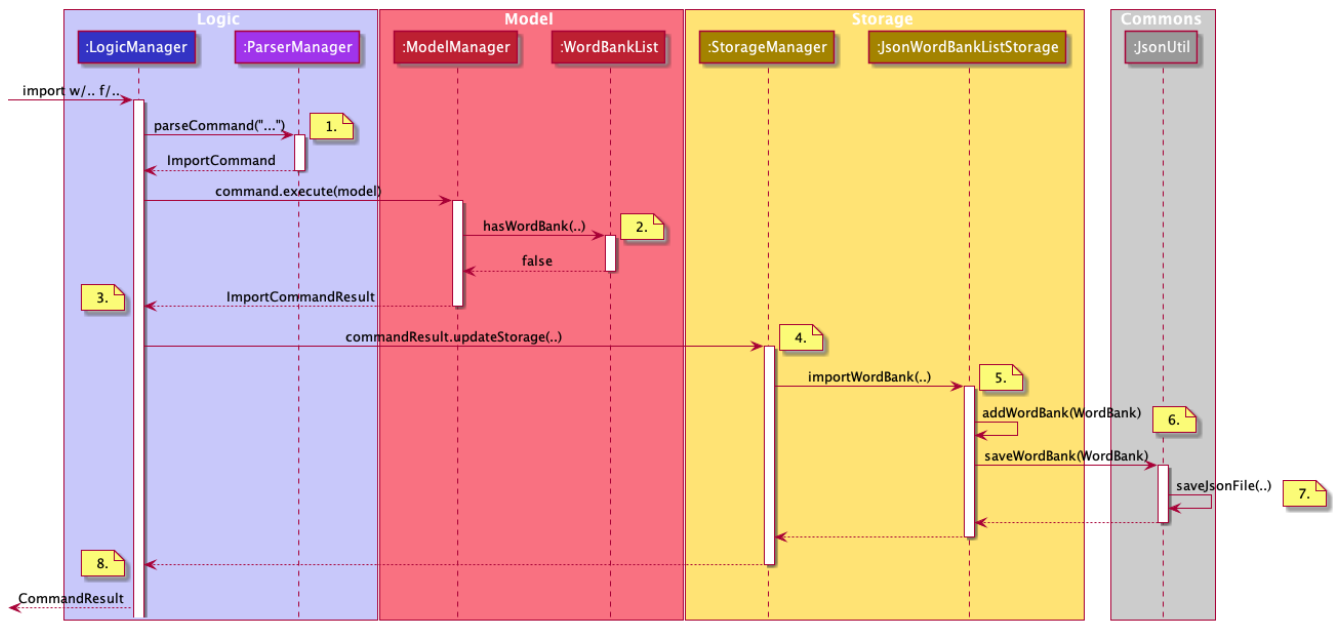
Figure 32. Sequence diagram detailing a successful `WordBankCommand` through different components.

NOTE
> Most of these methods utilised can be found in my word bank data structure and storage class diagrams.
> The emphasis here is to show how these commands utilise word bank's data structure and storage.

We will see the case where an Import command is valid.
For example, for the input: "import w/bank1, f/~/downloads"

1. Depending on the input, a specific `Command` type is returned by `ParserManager`. i.e. `ImportCommand`.

2. Each type of `Command` executes with slight variance. `ImportCommand` executes and checks in `Model` to check if `WordBankList` already contains `WordBank`.

3. Relevant information is stored in a specific `CommandResult` and is returned back to `LogicManager`. i.e. `ImportCommandResult`.

4. Each type of `CommandResult` updates the storage with slight variance. `ImportCommand` calls the importWordBank method.

5. `JsonWordBankListStorage` contains the abstracted details of how a commandResult should be handled. For importWordBank method, addWordBank and saveWordBank private methods are called.

6. Within addWordBank method, `WordBank` is added into the underlying UniqueWordBankList. Two synchronisation happens here.
   Firstly, as `Model` contains the same `WordBankList`, the two list contains synchronised data.
   Secondly, `WordBankList` exposes an unmodifiable `ObservableList<WordBank>` that can be 'observed'. `UI` was bounded to this list upon initialisation. Hence, it allows the user sees the updated word bank automatically.

7. Within saveWordBank method, an even lower level saveJsonFile function is called to write to the disk. This is performed through the common class: `JsonUtil`.

8. It returns back to `LogicManager`, and a success message is passed back to `AppManager`, then to the `UI` to notify the user.

- Other `CardCommands` and `WordBankCommand` work similarly to `ImportCommand`, with slight variance.

3.4.3. Drag and drop

Enables saving/loading a word bank file into/out of user's computer through drag and drop. Improves user experience by making it easy to share word banks with friends.

| NOTE | As mentioned previously, drag and drop will extend and utilise word bank's storage heavily.
You can refer to it to enhance your understanding of this implementation. |
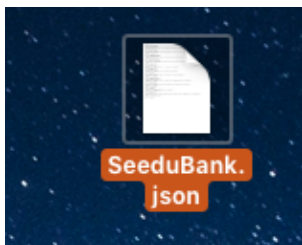
Figure 33. Word bank file.
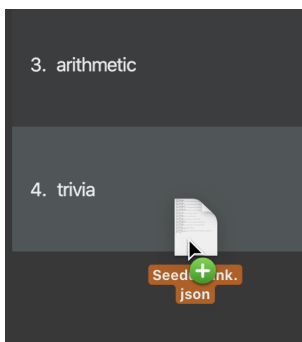
Figure 34. Dragging into Dukemon.
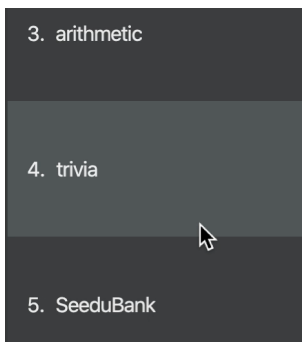
Figure 35. Dukemon registers the bank.

From `HOME` mode, you can view your `WordBanks`. Simply drag and drop a `WordBank` json file from your computer into Dukemon. Likewise, drag and drop a `WordBank` out of the application, into say, your desktop, or chat applications.

Walkthrough - Drag in.
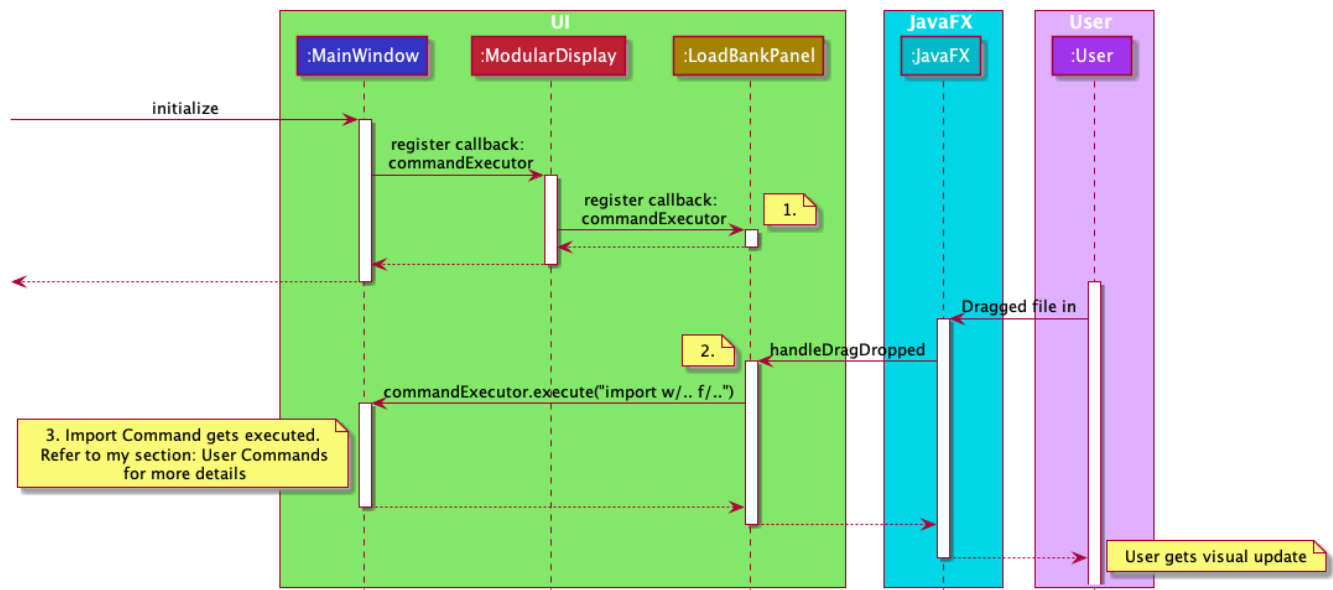
Figure 36. Sequence diagram showing how drag and drop utilises the ImportCommand and thus the word bank's storage.

`LoadBankPanel` is the corresponding class and the FXML file that displays the word banks for the user.
It is deeply nested within `UI` and only has access to an `ObservableList<WordBank>`.
This means it has no way to perform commands, update model or update storage.

1. To work around this, a functional callback is registered within LoadBankPanel.

2. LoadBankPanel registers JavaFX's UI drag detection and drag dropped methods, with the callback.

3. After which, the callback essentially performs an `ImportCommand`, to load the word bank.

It is also noteworthy to mention that, dragging into Dukemon functionality is well guarded against:

- Not json file format.

- Json file but data in wrong format.

- Json file with correct format but contains duplicate cards within.

User receives apt feedback through the command box for different cases. This is possible with careful exceptions handling within the `ImportCommand` itself.

3.4.4. Revision word bank

Serves as a centralised bank that collects cards across different word banks through the game play.
The update to revision word bank is done automatically after every game session.

As mentioned previously, revision word bank will extend and utilise word bank's data structure and storage heavily.

You can refer to it to enhance your understanding of this implementation.
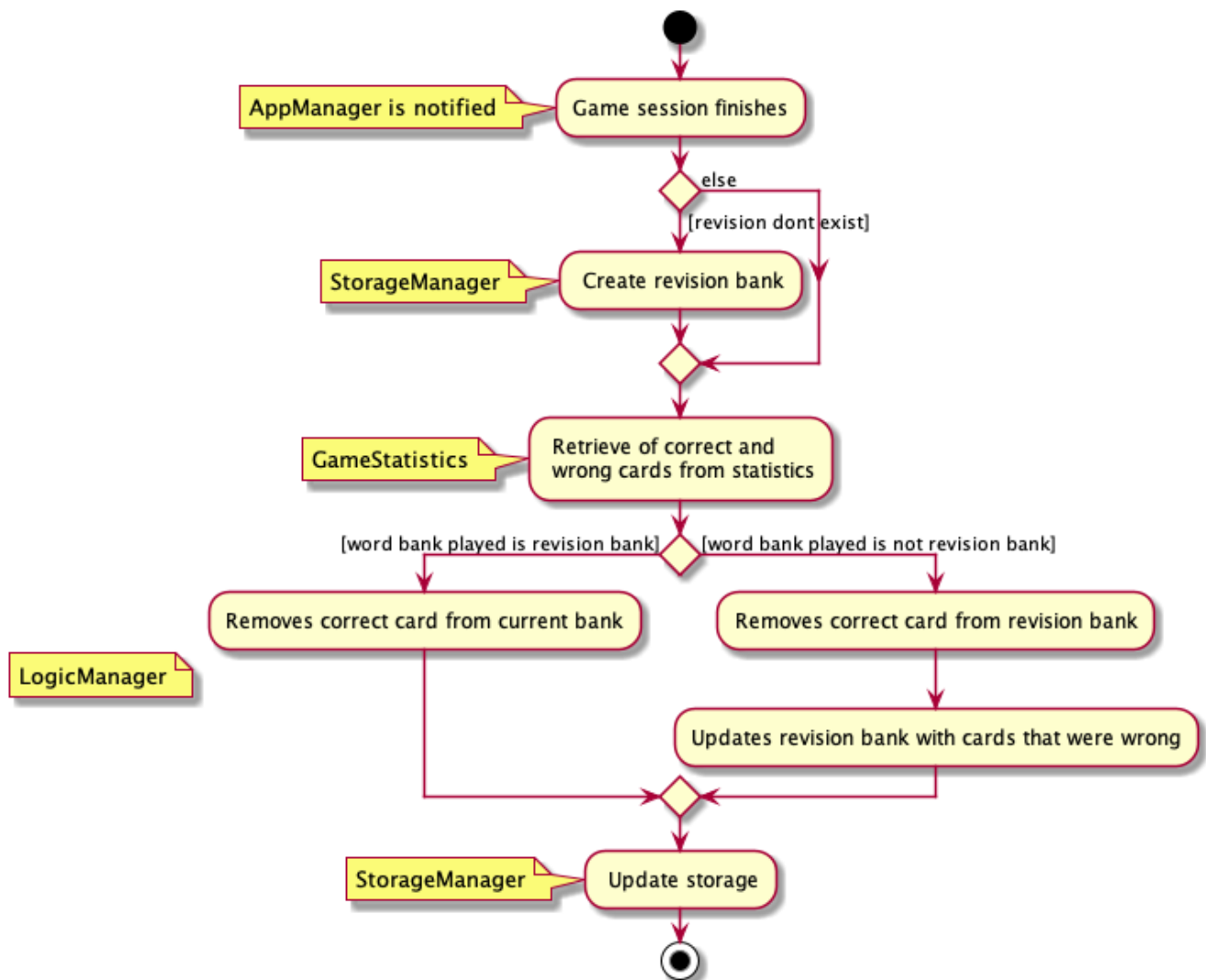

Figure 37. Activity diagram showing different scenarios possible while trying to update revision bank.

Revision bank was one of the essential and dominant features we wanted to implement since early development, however it had to be implemented last because it required multiple components working together.

These components include:

- AppManager
- StorageManager
- LogicManager
- GameStatistics

Given that well-developed methods reside in each of these components, we then require an overview of revision bank implementation. The activity diagram above is able to detail my thought process and implementation.

1. `AppManager` gets first notification that a `Game` session has ended.

2. `StorageManager` creates revision word bank if necessary.

3. `GameStatistics` gives the required information: correct and wrong cards.

4. `LogicManager` manages the processing of these cards, with some slight variance depending on situation, into revision bank.

5. `StorageManager` saves it back into hard disk.

3.4.5. Design Considerations

	Alternative 1	Alternative 2
Aspect 1: Data structure for WordBankList.	**Create classes for both WordBankList and WordBank, even though they are very similar in structure.**	**Create a generic data structure class, and let both WordBankList and WordBank extend it.**
	Pros: User's modification to their word banks and cards requires very different methods. These two data structure requires different access to the storage as well. With two different classes, implementation of the Commands that work on these data becomes more distinct. This ensures methods within `WordBankList` are written for `WordBankCommands` and methods within `WordBank` are written for `CardCommands`, thereby increasing cohesion of individual components and decreasing coupling between the two classes. *Cons:* Implementation requires much more effort.	*Pros:* Code that are reusable in WordBank can now be reused for WordBankList. *Cons:* This couples WordBank with WordBankList. Does not follow the Open-Closed principle.

Why did we choose Alternative 1:
In the spirit of software engineering principles, it is better to have the basic data structure implemented well. Commands that depend on it becomes much easier to implement. (This can be seen in the drag and drop feature.)

	Alternative 1	Alternative 2
Aspect 2: Storage system for word banks.	**Store one single large json file with word bank names as keys and it's word bank data as values**: *Pros:* Always save a snapshot of the data to the same file, regardless of what commands are executed. *Cons:* Unable to share word banks with friends, because one file contains all the word banks.	**Store each word bank as a json file.** *Pros:* Enables sharing of word bank files to friends. *Cons:* Require more consideration to deal with different type of commands which affects the storage dynamically. Harder to read from multiple files.

Why did we choose Alternative 2:
This choice was based largely from the user's perspective.
As our app is designed to streamline learning, we figured that easy sharing of word banks file with friends is an important aspect in our app, and cannot be compromised.

	Alternative 1	Alternative 2

Aspect 3: Command implementation. (Same goes for Command Result implementation)	**All types of commands extends a single abstract class `Command`:** *Pros:* A rather simple implementation which does not break any software engineering principles. *Cons:* Can be further improved, as in Alternative 2.	**Distinguishing `WordBankCommand` and `CardCommand` specifically - Commands that work on Cards extends the abstract `CardCommand` class and commands that work on `WordBank` extends the abstract `WordBankCommand` class.** *Pros:* As we have created distinct data structure for `WordBankList` and `WordBank`, distinguished commands now work solely on their respective data structure. It follows the Single Responsibility Principle and the Separation of Concerns Principle more closely, and decreases the coupling between the two component. *Cons:* Requires tedious implementation to follow the principles.

Why did we choose Alternative 2:
Allows for easy extension of Dukemon's functionality. Implementation of the drag and drop feature is now a few function calls away, as all data structure and functions are well written.

	Alternative 1	**Alternative 2**

Aspect 4: How to implement Drag and Drop. `LoadBankPanel` is a deeply nested class, and is the corresponding class for the `UI` to interact with user's drag and drop action.	**Updates the storage directly from `LoadBankPanel`**: *Pros:* It only requires a reference and then saving directly to `Storage`. This can be implemented with ease. *Cons:* Practically, there are a few exceptions being thrown when calling the storage's method directly. `LoadBankPanel` cannot handle them effectively. This also leads to poor abstraction and high potential for cyclic dependencies, resulting in high coupling.	**Using *Functional Interfaces* as Call-backs to call an `ImportCommand` from `LoadBankPanel`.** *Pros:* Calling an already well-implemented `ImportCommand` allows all exceptions caught to be handled properly. It also maintains abstraction and minimal coupling between `LoadBankPanel` and other components. *Cons:* It makes the code less OOP and more functional.

Why did we choose Alternative 2:
Provides a more complete implementation, as it would make sense for exceptions to be caught and allow user to see feedback messages.

3.5. Statistics features

3.5.1. Implementation

The work of the Statistics component can be neatly captured and explained using a common series of user actions when operating the app.

User action	**Statistics work**	**UI Statistics updates**
User opens the app.	User's `GlobalStatistics` and `WordBankStatisticsList` are loaded into `Model` by the `MainApp`.	User is shown their `GlobalStatistics` and their most played word bank from the `WordBankStatisticsList` in the main title page.
User selects a word bank.	The selected `WordBankStatistics` from the `WordBankStatisticsList` is loaded into `Model`.	
User opens the selected word bank.		In open mode, User is shown the `WordBankStatistics` of the opened word bank.

User action	Statistics work	UI Statistics updates
User plays the game.	A `GameStatisticsBuilder` is used to record user actions during the game.	
User finishes the game.	- A `GameStatistics` is created from the `GameStatisticsBuilder`. - The `WordBankStatistics` and `GlobalStatistics` are updated accordingly and saved to disk.	`GameStatistics` and the corresponding `WordBankStatistics` are displayed to user in the game result page.

We will discuss each step with its implementation details primarily on the statistics work.

1. **User opens the app**

When the user opens the app, their `GlobalStatistics` and `WordBankStatisticsList` are loaded into `Model` by `MainApp`.

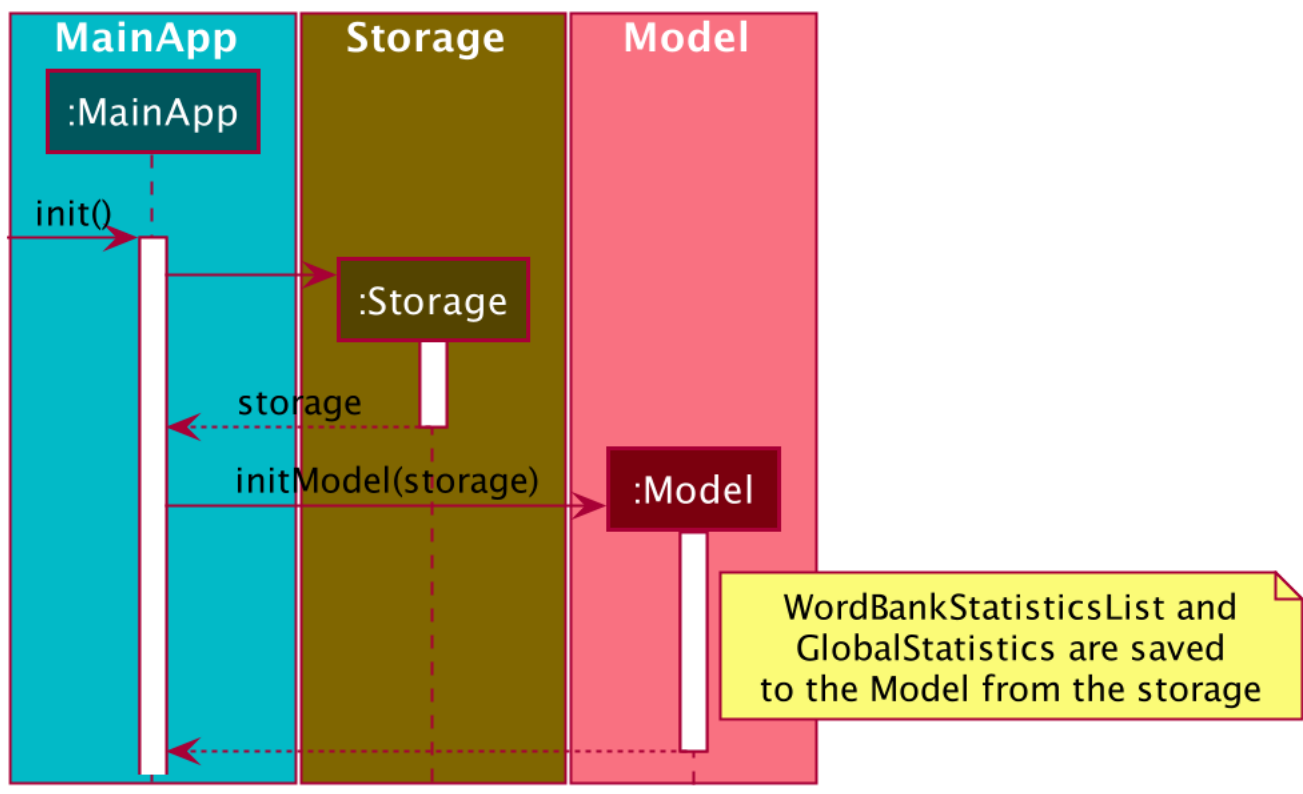

Figure 38. Sequence diagram for loading statistics

2. **User selects a word bank**

When the user selects a word bank, the selected `WordBankStatistics` from the `WordBankStatisticsList` is loaded into Model.
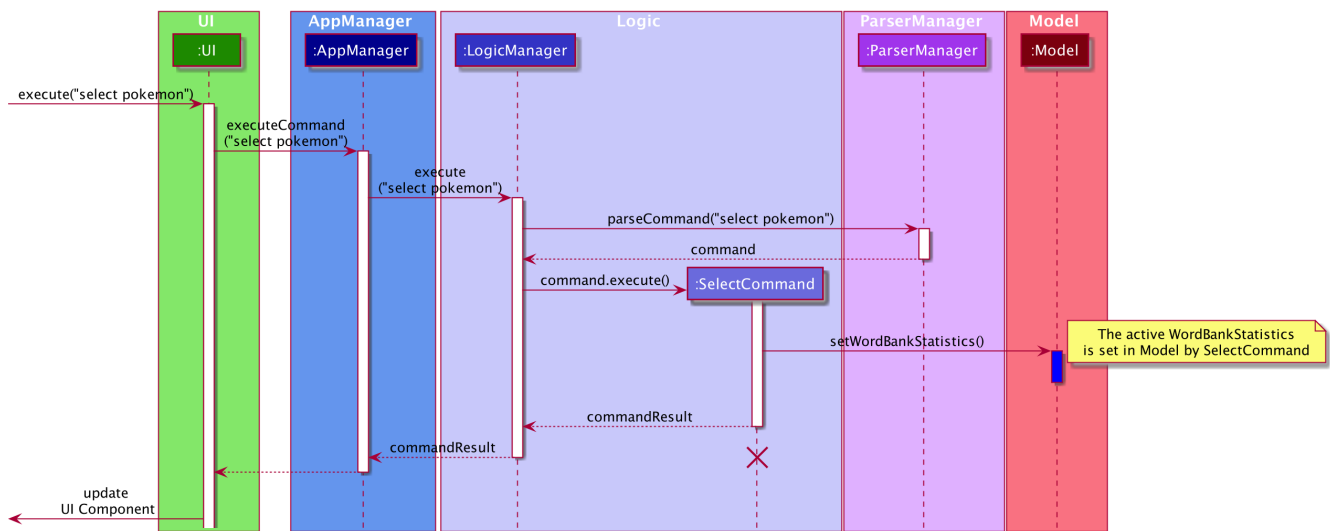
Figure 39. Sequence diagram for selecting a word bank statistics.

It is necessary to set the active `WordBankStatistics` in the `Model` such that when the user opens the `WordBank`, the `WordBankStatistics` can be found in `Model` and shown in the UI.

3. <u>User opens the selected word bank</u>

In open mode, the user is shown the `WordBankStatistics` of the opened word bank, which is set in `Model` at step 2.

4. <u>User plays the game</u>

A `GameStatisticsBuilder` is used to record user actions during the game.

When the user starts the game by calling a `StartCommand`, the `GameStatisticsBuilder` is initialized. Additionally, the `GameStatisticsBuilder` is updated with every `GuessCommand` or `SkipCommand` made during the game. It receives the timestamp from the `GameTimer` which also resides in `AppManager`.
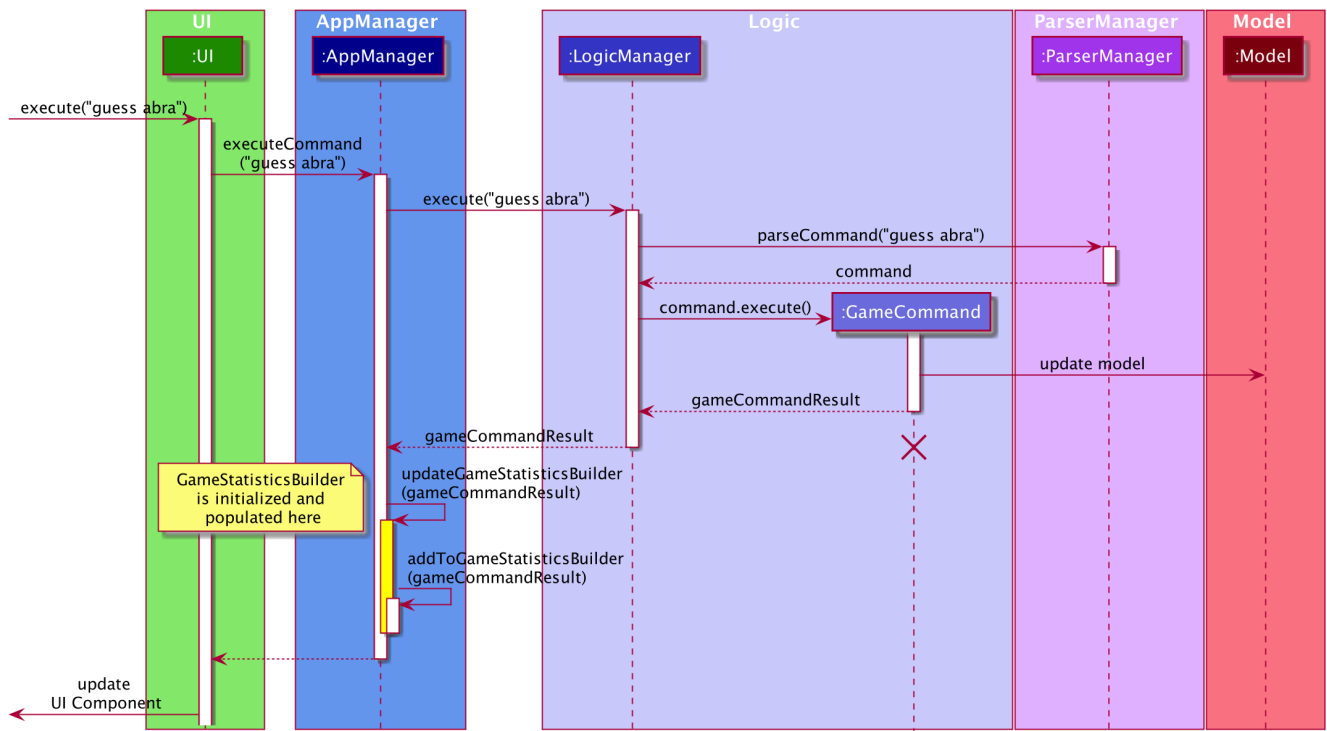
Figure 40. Sequence diagram when user makes a guess.

5. User finishes the game

When the user finishes the game, a `GameStatistics` is created from the `GameStatisticsBuilder`. The `GameStatistics` is shown to the user in the game result page.

The `GameStatistics` is used to update its corresponding `WordBankStatistics`, which is then saved to disk. Additionally, the `GlobalStatistics` is also updated and saved to disk.
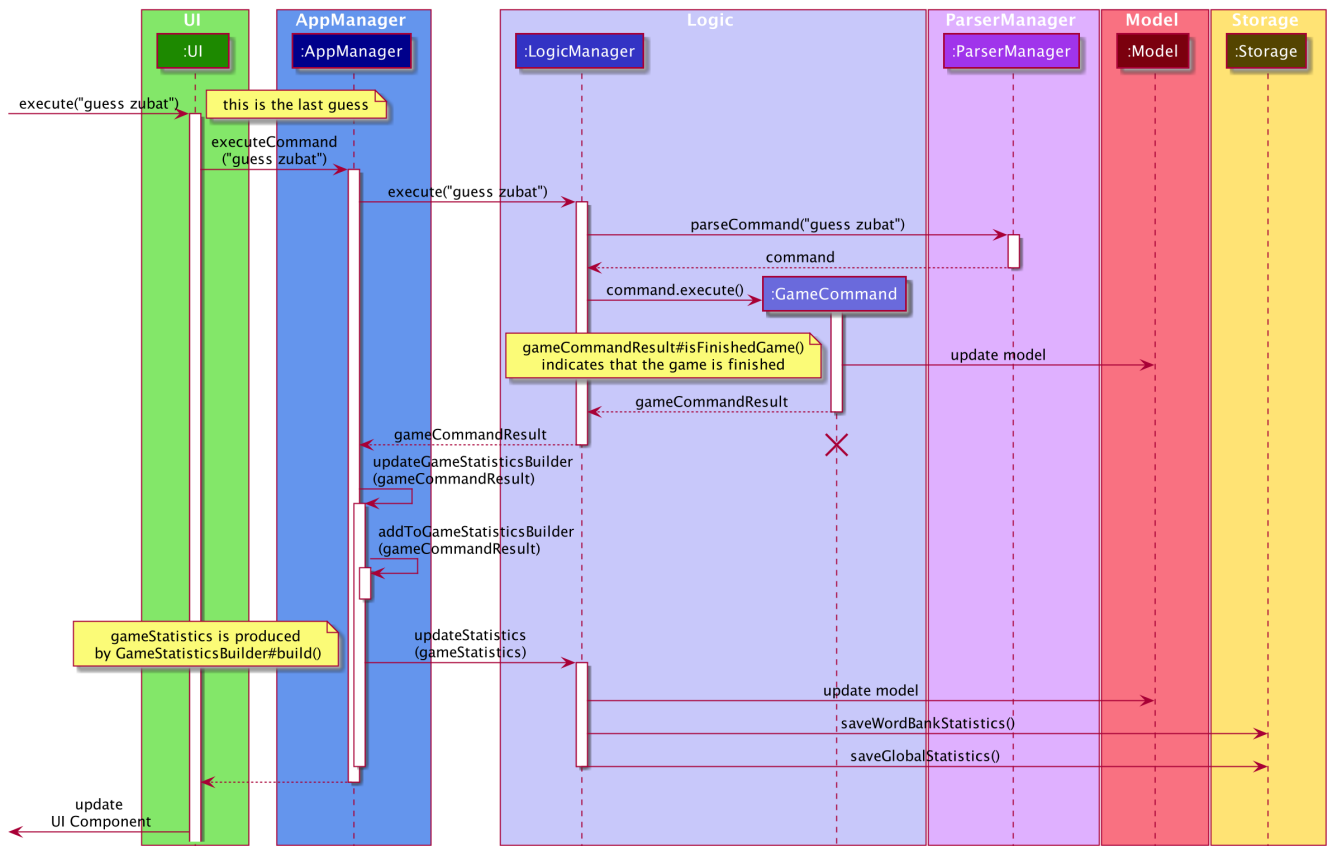
Figure 41. Sequence diagram when the user makes the final guess.

The work done in step 4 and 5 is executed in `AppManager` and the checks to decide what to do are done in the same method `updateGameStatisticsBuilder(CommandResult)`.

Figure 42. Activity diagram when `AppManager` receives a `CommandResult` (Details unrelated to statistics are omitted).

3.5.2. Design Considerations

There were some design considerations on implementing the statistics.

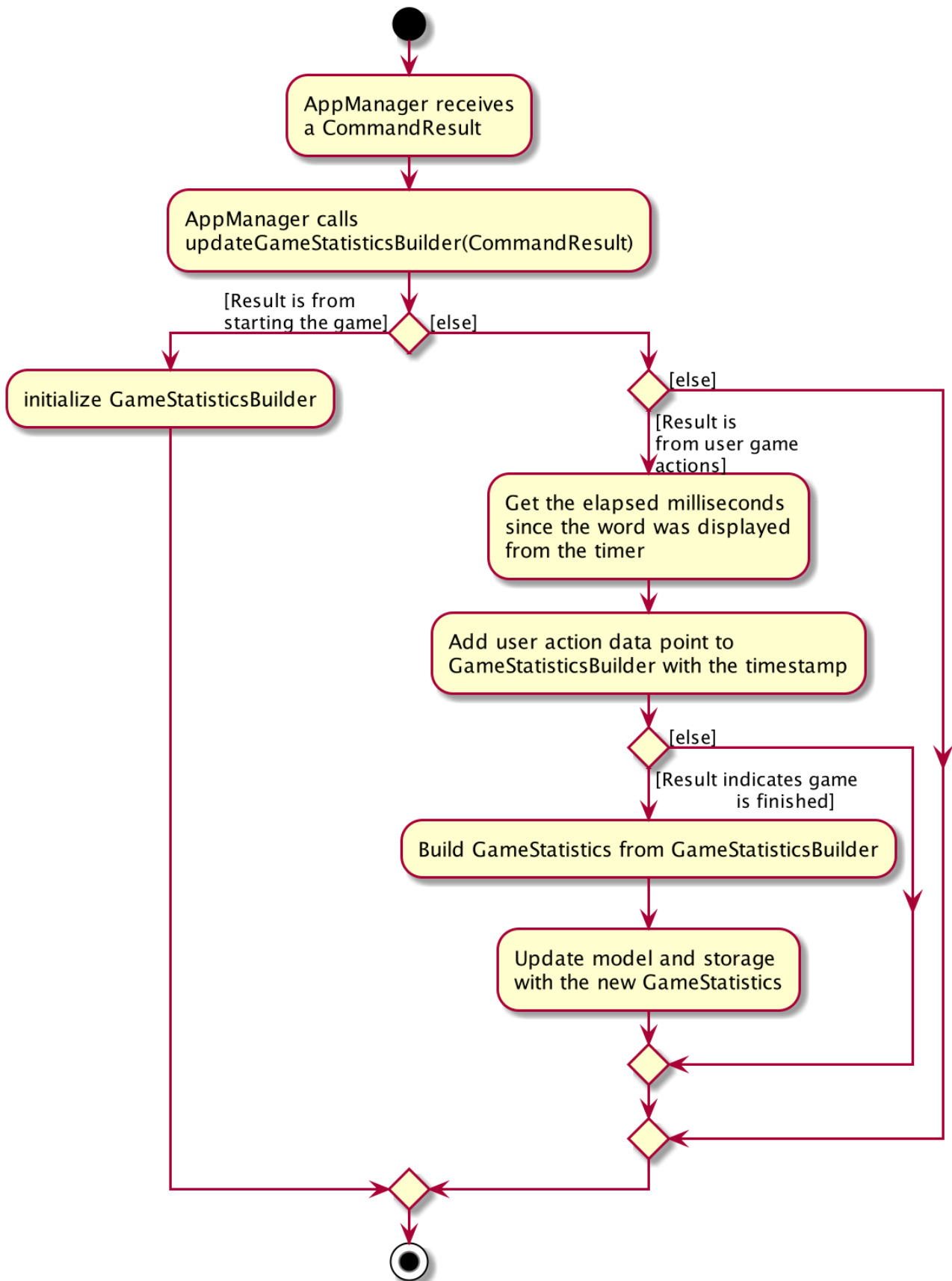|	Alternative 1	Alternative 2
Aspect 1: How to store `WordBankStatistics` in the storage?	**Store in a separate file from the `WordBank` json file, but with the same name in a different directory.** Example: `WordBank` data is stored at *data/wordbanks/pokemon.json* while the `WordBankStatistics` data is stored at *data/wbstats/pokemon.json* *Pros:* More abstraction to separate the data. *Cons:* The data is linked by name, so if the user changes the file name, the link is broken.	**Store `WordBankStatistics` data in the same file as `WordBank`** *Pros:* Less number of files. *Cons:* Data is combined into one which lowers abstraction.
Why we decided to choose Alternative 1: We decided that abstraction between the data is important as each team member should work in parallel, such that it is easier for one person to modify the storage system for the word bank and another person to modify the storage system for the word bank statistics freely.		

3.6. [Proposed] Data Encryption

{Explain here how the data encryption feature will be implemented}

3.7. [Proposed] User Profiles

The user profiles could allow multiple users to use the same app and have different statistics tracked. This feature is a work in progress and will be delayed to v2.0.

3.8. 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 3.9, "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

3.9. Configuration

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

4. Documentation

Refer to the guide here.

5. Testing

Refer to the guide here.

6. Dev Ops

Refer to the guide here.

Appendix A: Product Scope

Target user profile:

- students
- wants to learn new English words or definitions
- can type fast
- enjoys games
- is reasonably comfortable using CLI apps

Value proposition: gamify learning experiences

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...
* * *	teacher	add, edit, and delete questions in the word banks	make corrections on what my students are supposed to learn
* * *	teacher	give customised word banks and definitions	can let my students practice specific problems.
* * *	user	list all my word banks	
* * *	user	give titles to word banks	recognise them better
* * *	user	delete word banks	free up some memory when I don't need it anymore
* * *	user	see the content of the word bank	study beforehand/make changes
* * *	young student	trivia questions to be gamified	enjoy the process
* * *	student	create my own question banks	tailor fit to my learning
* * *	computer science student	have a manual of the commands available	refer to them when I am lost
* *	frequent user	easily access my most recently attempted question sets	can quickly resume my revision
* *	studious student	set and complete goals	have something to work towards

Priority	As a …	I want to …	So that I can...
* *	student	see my test statistics	track my progress/improvement
* *	student	choose different kinds of time constraints	can simulate exam conditions
* *	student	categorise my question sets	easily look for relevant materials
* *	student	mark question sets as important/urgent	know how to prioritise my revision
* *	module coordinator	export lessons	send to their students
* *	student	share and compare my results with my classmates	know where I stand
* *	student	partition the trivia	attempt questions that I'm comfortable with
* *	weak student	have the option to see hints	won't get stuck all the time
* *	computer science student	practise typing bash commands into the CLI	strengthen my bash skills
* *	teacher	export statistics	can compare performance across different students

Priority	As a ...	I want to ...	So that I can...
*	computer science student	customize my "terminal"	changing themes/ background/ font size/ font colour, so that I feel comfortable working on it
*	teacher	protect tests with passwords	let my students do them in lessons together when password is released
*	teacher	protect the files	doesn't get tampered when distributing to students
*	student	have smaller sized files	have more space on my computer

{More to be added}

Appendix C: Use Cases

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

Use case: Delete person

MSS

1. User requests to list persons

2. Dukemon shows a list of persons

3. User requests to delete a specific person in the list

4. Dukemon deletes the person

   Use case ends.

Extensions

   2a. The list is empty.

   Use case ends.

3a. The given index is invalid.

    3a1. Dukemon shows an error message.

    Use case resumes at step 2.

{More to be added}

Appendix D: Non Functional Requirements

1. Compatible any mainstream OS as long as it has Java 11 or above installed.

2. User has above average typing speed for regular English text (i.e. not code, not system admin commands); able to accomplish most of the tasks faster using commands than using the mouse.

3. Users can export and import their wordbanks or *Statistics*.

4. Feedback shown to user must be fast (< 0.2s) especially during a *Game*.

{More to be added}

Appendix E: Glossary

Mainstream OS

    Windows, Linux, Unix, OS-X, Ubuntu and *etc.*

WordBank

    Data structure that contains a list of several *Cards*.

Card

    Analogue to a physical Flashcard- contains a *Word* and a *Meaning*.

Word

    The component of a *Card* that is to be guessed by the user during a *Game*.

Meaning

    The meaning represented by the *Word* of a *Card*; is shown to the user during a *Game*.

Game

    A game session that runs on a specific *WordBank* of *Cards*.

Callback

    A piece of executable code that is passed as an argument to other another code that is expected to *callback* (execute) the argument at a given time. (Adapted from Wikipedia)

Appendix F: Product Survey

Product Name

Author: ...

Pros:

- ...

- ...

Cons:

- ...

- ...

Appendix G: Instructions for Manual Testing

Given below are instructions to test the app manually.

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

G.1. Launch and Shutdown

1. Initial launch

    a. Download the jar file and copy into an empty folder

    b. Double-click the jar file
    Expected: Shows the GUI with a set of sample contacts. The window size may not be optimum.

2. Saving window preferences

    a. Resize the window to an optimum size. Move the window to a different location. Close the window.

    b. Re-launch the app by double-clicking the jar file.
    Expected: The most recent window size and location is retained.

{ more test cases ... }

G.2. Deleting a person

1. Deleting a person while all persons are listed

    a. Prerequisites: List all persons using the `list` command. Multiple persons in the list.

    b. Test case: `delete 1`
    Expected: First contact is deleted from the list. Details of the deleted contact shown in the status message. Timestamp in the status bar is updated.

    c. Test case: `delete 0`
    Expected: No person is deleted. Error details shown in the status message. Status bar

remains the same.

d. Other incorrect delete commands to try: `delete`, `delete x` (where x is larger than the list size) *{give more}*
Expected: Similar to previous.

{ more test cases … }

G.3. Saving data

1. Dealing with missing/corrupted data files

   a. *{explain how to simulate a missing/corrupted file and the expected behavior}*

{ more test cases … }