



Koh Yi Da - Project Portfolio

Project: [Dukemon](#) 

Github: [Dukemon](#) 

RepoSense: [Code](#)

1. Overview of Dukemon

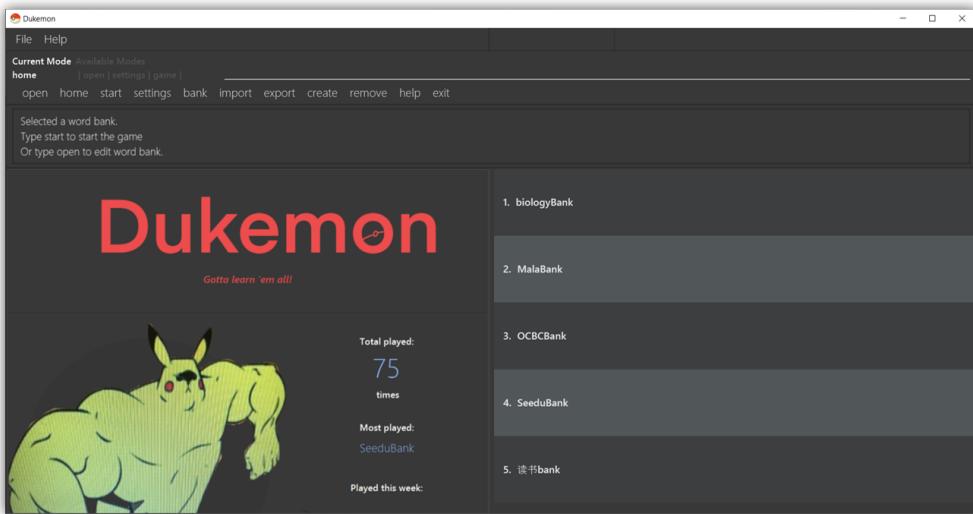


Figure 1. The Home Page of Dukemon upon initial start-up.

Dukemon is a desktop app intended as a fun study tool. It is a CLI-centric (*Command Line Interface*) app that expands upon the idea of Flashcards to aid learning in a fun and exciting way. The main program flow of **Dukemon** is as follows:

1. User creates a *WordBank*.
2. User creates *Cards* that have a *Word* and *Meaning* each.
3. User populates his *WordBank* with such *Cards*.
4. User starts the *Game* and tries to match *Meanings* with *Words* within a certain Time.
5. User completes the *Game* and reviews his performance *Statistics*.

Developed by my team and I, **Dukemon** transforms the basic concept of *Flashcards* into an exciting and engaging game-like app through features such as *automatic Hints*, *Statistics* and so much more.

Below are some **highlights** of the important contributions that I have made to the development of **Dukemon**.

2. Contributions - Summary

2.1. Main Enhancements - Timer and Hints

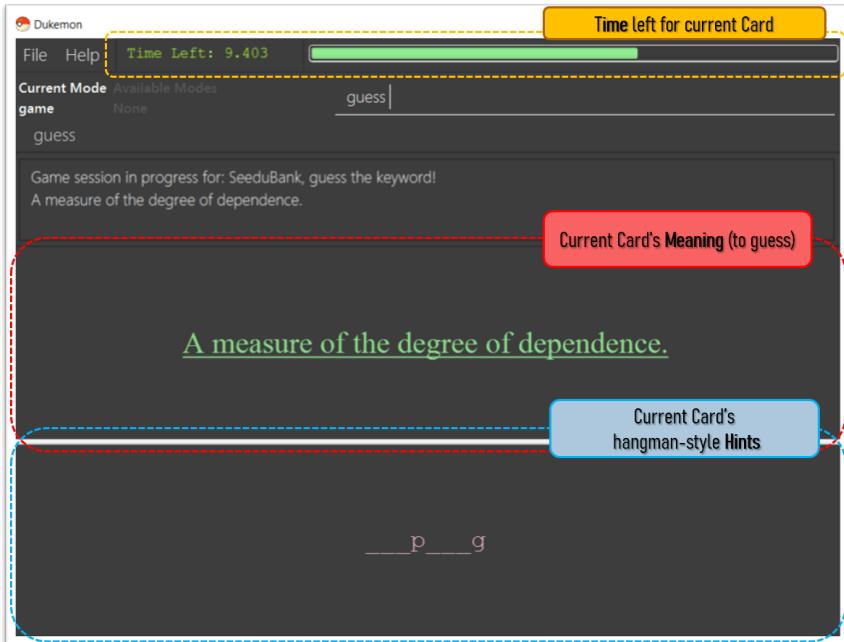


Figure 2. UI of Dukemon in Game Mode, showing the dynamic Timer and Hint features I implemented.

- Added a Timer and automatic Hint feature

- Brief Description:

- A live text and graphical countdown *Timer* (region in yellow box above) that shows the User how much time is left. Based on the time left, *Hints* (presented in a Hangman style) are also automatically generated and shown to the user (region in blue box above).

- Justification:

- This enhancement greatly improves the product as it achieves the intended goal of creating a **game-like learning environment** for the User that is fun and engaging.
 - Hints aids in learning, especially for weaker students or when trying out unfamiliar words.
 - Incremental as opposed to one-shot hints gives the User time to think, and doesn't potentially render the *Game* too easy with hints.

- Highlights:

- Challenging as it required **seamless integration** and **synchronization** between the GUI (Graphical User Interface) and internal logical components in **real time**.
 - Utilized **advanced programming concepts** such as *Observer Patterns*, *Callbacks* and *Functional Programming* to preserve quality and structural integrity of existing code base. API like `java.util.concurrent.CountDownLatch` and `java.lang.reflect` were used to run tests for *Callbacks* and the *Timer* effectively.
 - Integrated external *TestFX* library to allow for testing of *Timer* and other components that run on the *JavaFX Application Thread*.

- Credits (*Framework/Libraries used*):
 - [JavaFX 11](#) (GUI), [TestFX](#) (Testing), [JUnit5](#) (Testing)
- Credits (*People*):
 - Jason (@jascxx) for the bug resolution and implementation of [Cards](#).
 - Paul (@dragontho) for integration of Hints and Questions with UI.
- **Code contributed:**
[\[Functional \(Timer\)\]](#), [\[Functional \(Hints\)\]](#), [\[Tests \(Timer\)\]](#), [\[Tests \(Hints\)\]](#)

2.2. Other Enhancement - Game

- Implemented and designed the *Game* logic, UI and Difficulty.
 - Brief Description:
 - The game is a primary feature *Game* where the User makes guesses for *Words* based on a *Meaning* shown. Different *Difficulty* modes are available that changes the time allowed per question.
- **Code contributed:**
[\[Functional\(Game Logic\)\]](#), [\[Functional \(Game Difficulty\)\]](#), [\[Tests \(Game\)\]](#)

2.3. Other contributions

- Project management:
 - Managed releases [v1.2 - v1.3](#) (2 releases) on GitHub
 - **Designed and prototyped** the general *Game* program flow and commands, which was adopted by the team.
 - Worked closely with teammates in **discovering and resolving bugs** in other areas of code. [#133](#)
 - Actively resolved and fixed project wide issues and code warnings. (**Housekeeping** of Dukemon and its releases) [#141](#) [#96](#)
 - Researched about and implemented *Callbacks* and *Event-Driven Design* which was adopted in other teammate's features. [#185](#)
- Documentation (*Details in next section*):
 - Added icons and diagrams to **User Guide** to aid in reading: [#137](#)
 - Edited and wrote Introduction, Installation, and Quickstart sections in **User Guide**: [#149](#)
 - Drew and explained overall architecture of *Dukemon* in **Developer Guide** [#94](#)
- Community:
 - PRs reviewed (with non-trivial review comments): [#49](#), [#71](#)
 - Contributed to forum discussions (examples: [1](#))
- Tools:
 - Integrated external testing library ([TestFX](#)) to the project ([#79](#)) and Travis CI builds ([#113](#)).

3. Contributions - User Guide

Given below are the highlights of what I contributed to the User Guide, showcasing my ability to write documentation targeting end-users.

By: Serebros Since: September 2019 Licence: MIT

4. Introduction - What is Dukemon?

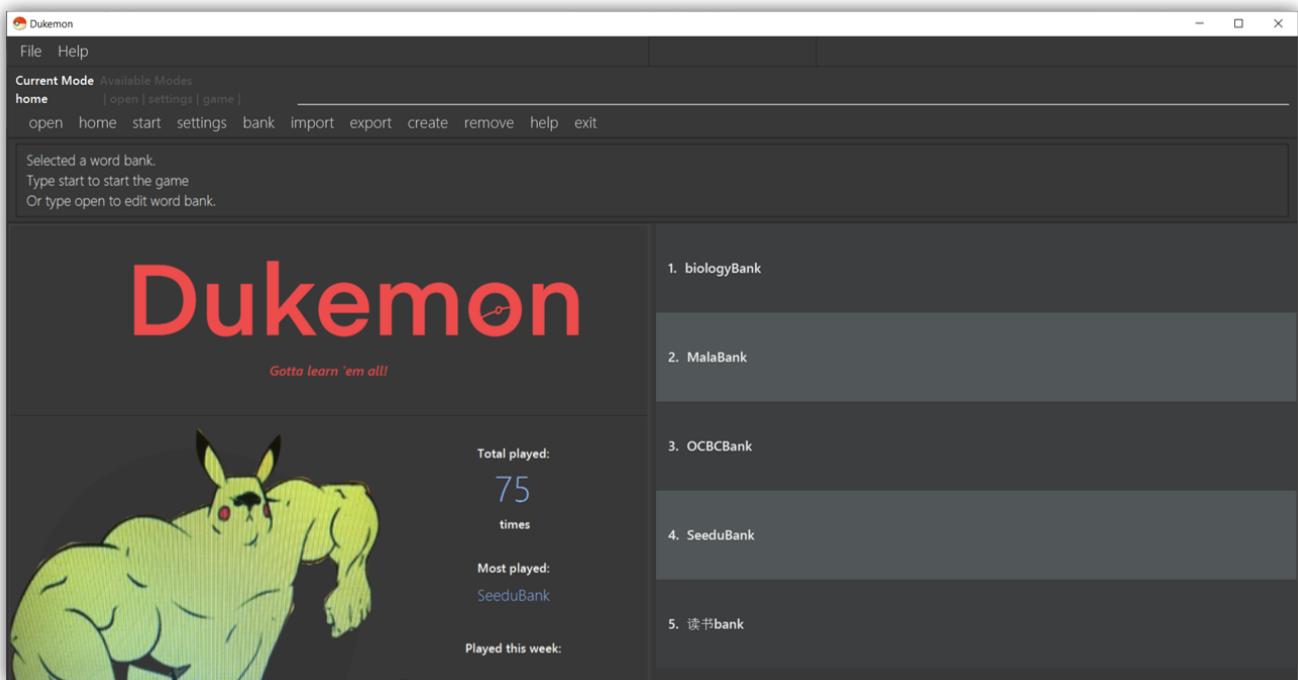


Figure 3. Home screen of Dukemon.

Welcome to Dukemon, the Flashcard app of the future!

Dukemon aims to streamline and gamify the process of learning words or definitions through the use of self-created digital flashcards. **Supercharge your learning with Dukemon!**

5. Getting Started

5.1. Installation

1. Ensure you have Java 11 or above installed on your system.
2. Download the latest [Dukemon.jar](#) here.
3. Copy the file to the folder you want to use as the home directory of Dukemon (this is where your data will be stored).
4. Double-click the [Dukemon.jar](#) to run the app.
5. Before getting to the quick start instructions, get familiar with our interface and application modes.

5.2. User Interface

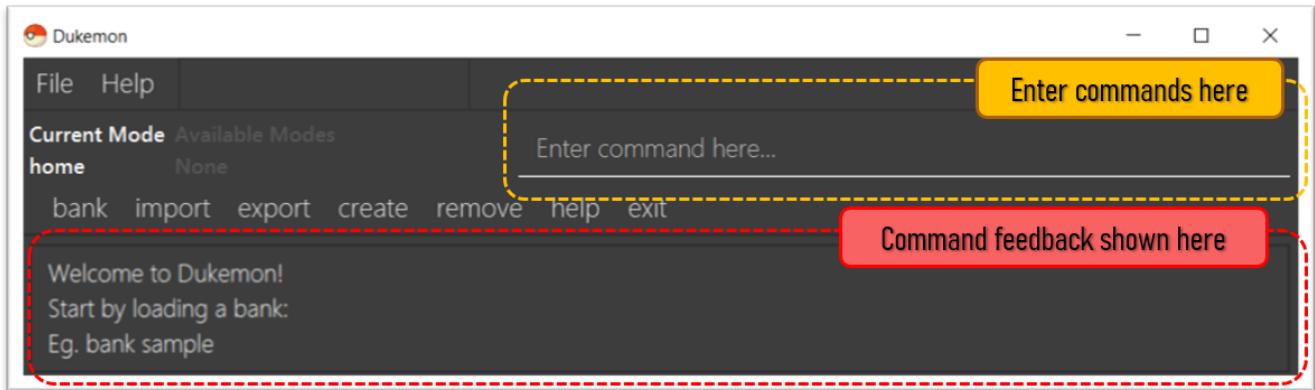


Figure 4. Regions of the UI where commands are entered (via CLI) and feedback from Dukemon is shown.

1. Click on the *CommandBox* as shown above (region in yellow box) and type commands in.
2. Use **Enter** to execute commands.
 - a. e.g. Typing **help** into the *CommandBox* and pressing **Enter** will open the *Help* window.
3. Text-based feedback for each command entered is shown in the *ResultDisplay* (region in red box).

5.3. Drag and drop

Streamline the process of sharing of word banks with friends.

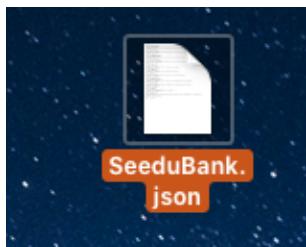


Figure 5. Word bank file



Figure 6. Dragging into Dukemon

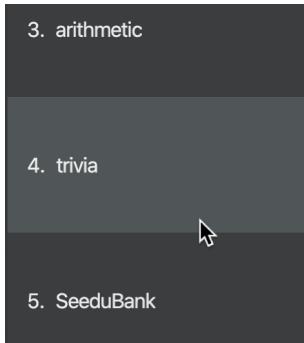


Figure 7. Dukemon registers the bank

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.

Try it!

NOTE Dukemon only reads json files.

5.4. Revision bank

A revision bank is automatically updated for you, every time a game session ends.

It collates all your wrong cards, whichever *WordBank* it may have came from.

It is also smart enough to remove the cards if you have cleared it afterwards.

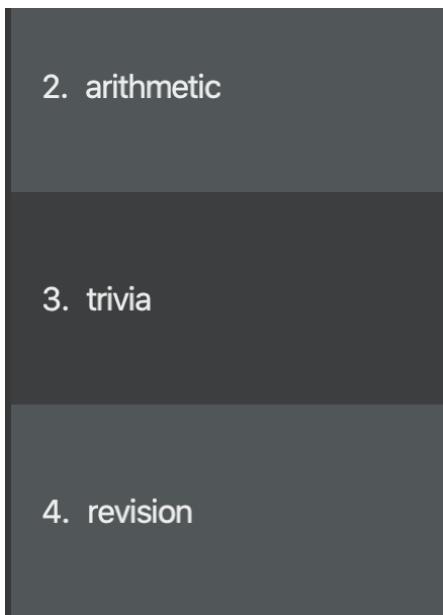


Figure 8. Revision bank alongside with other word banks

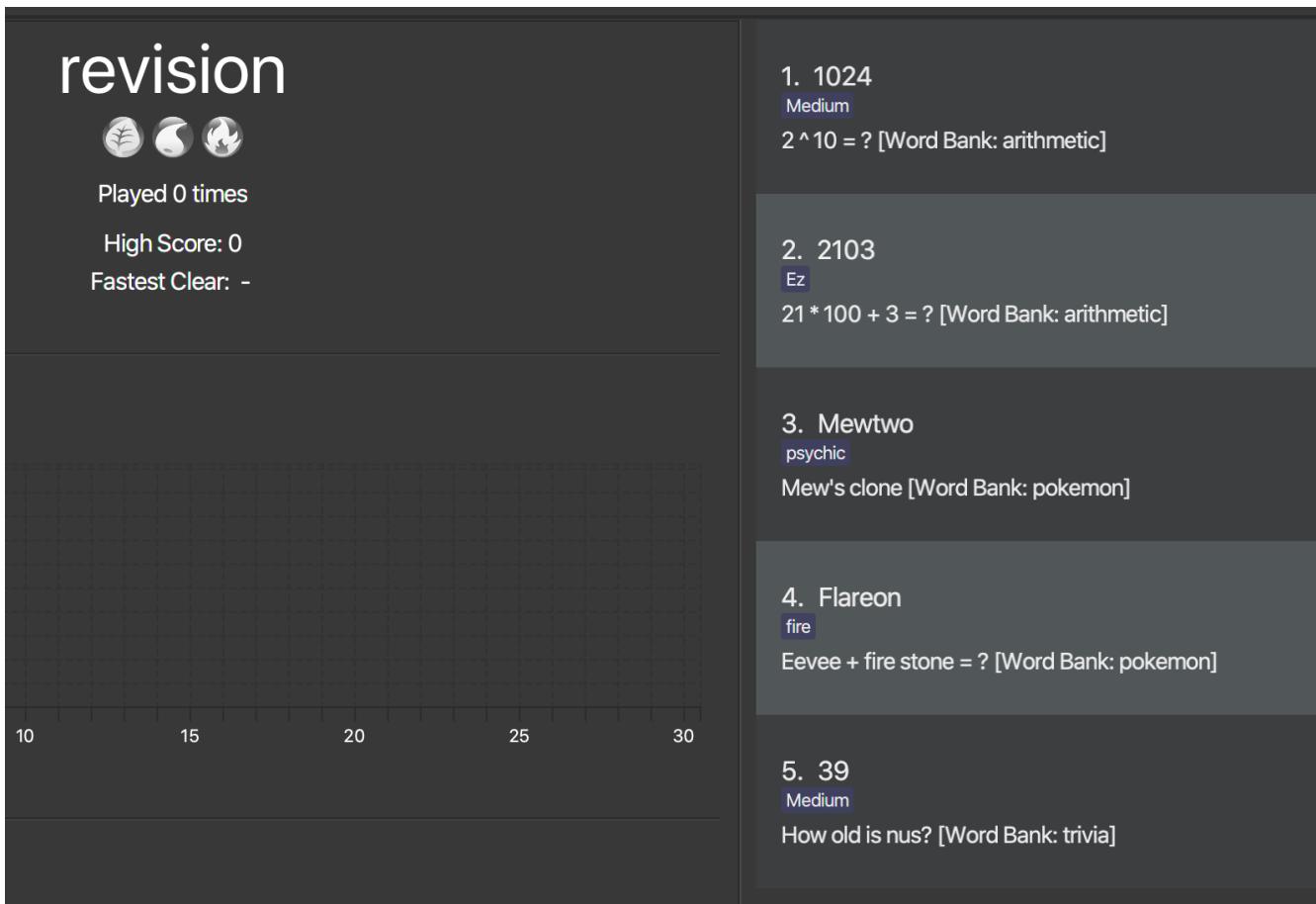


Figure 9. Screenshot of how a revision page looks like

To try it, simply play some *WordBank* and then visit the revision bank to attempt all your wrong cards.

5.5. QuickStart

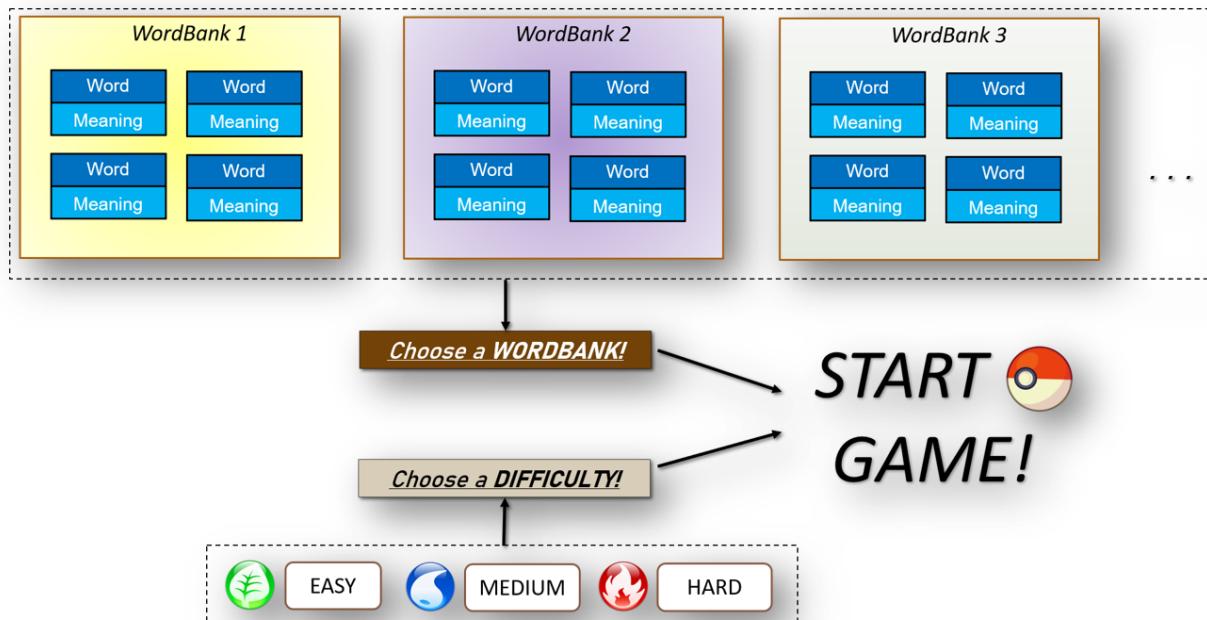


Figure 10. General program flow of Dukemon, showing how the different parts work together.

5.6. Game Commands

(Available in Game mode)

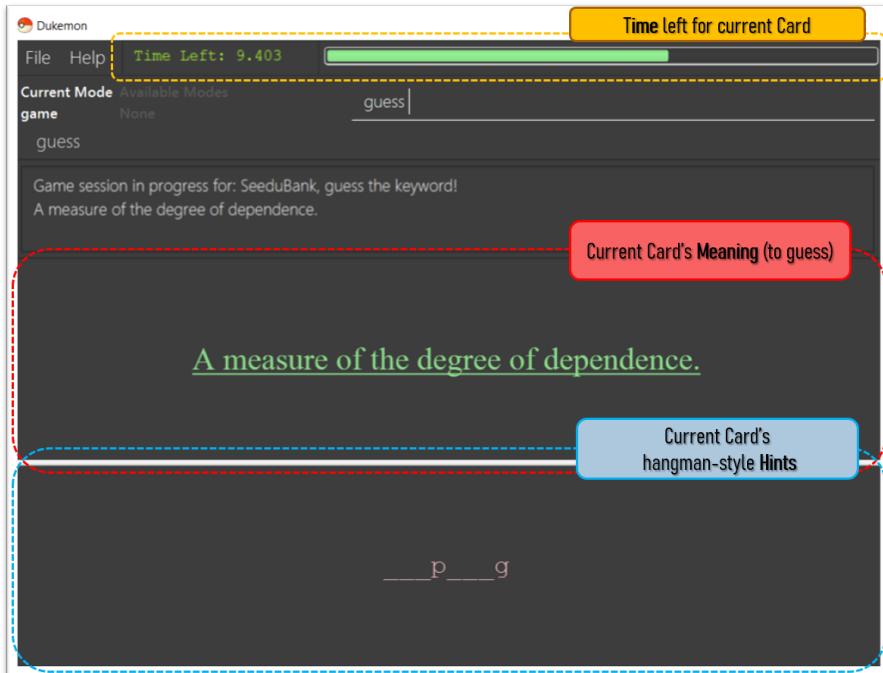


Figure 11. UI regions that are relevant when a Game session is in progress.

This section covers the actions and feedback that are relevant to the *Game* mode. The general layout of the UI when a *Game* is in progress is as seen above.

1. The timer will be activated to reflect the time left before the *Game* skips over to the next card. (region in yellow box)
2. The *Meaning* of the current *Card* is shown in the region contained by the red box. Based on this *Meaning* you will make a *Guess* for the *Word* it is describing.
3. *Hints* (if enabled) will be periodically shown as time passes (region in the blue box) in a Hangman-style. The number of hints given differs across each *Difficulty*.

5.6.1. Game Mode - Starting

The relevant command(s) are:

1. Starting new game session:

Format: **start [EASY/MEDIUM/HARD]**

- Starts a game session with the currently selected *WordBank* and specified *Difficulty*. (*WorkBank* selection is done in *Home* mode.)
- If no *Difficulty* is specified, the default *Difficulty* in *Settings* will be used.

5.6.2. Game Mode - Playing 🎮

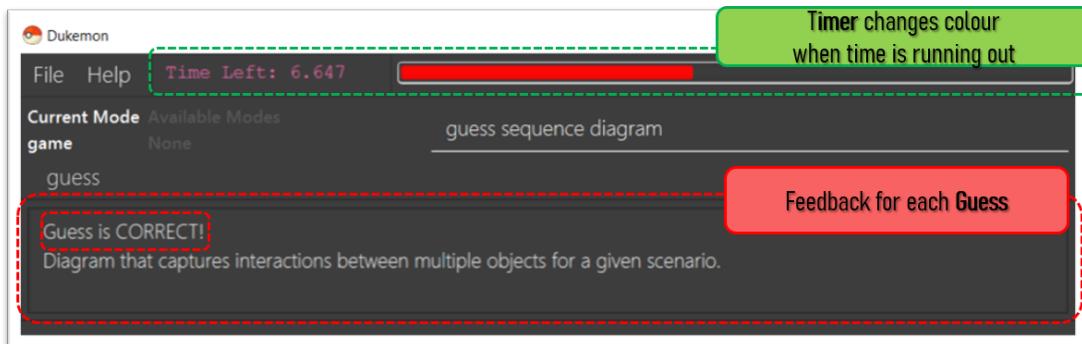


Figure 12. UI regions that show feedback during a Game session.

During a *Game*, the *Timer* will change colour according to the time left (region in green box). Feedback for each *Guess* is shown via the *ResultDisplay* (region in the red box).

The relevant command(s) are:

1. Making a *Guess* for a *Word*:

Format: `guess WORD`

- Makes a guess for the *Word* described by the currently shown *Meaning*. (**non case-sensitive**)

2. Skipping over a *Word*:

Format: `skip`

- Skips over the current *Word*. (**is counted as a wrong answer**)

5.6.3. Game Mode - Terminating & Statistics 🎮

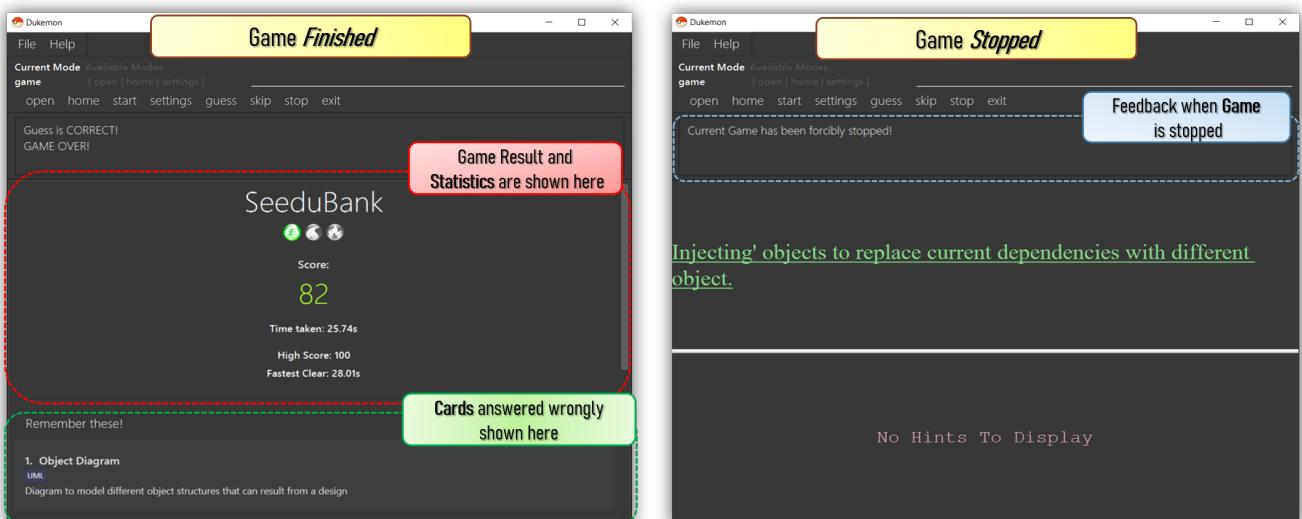


Figure 13. Comparison of UI Regions between Game finished vs. Game forcibly stopped.

A *Game* finishes when **all Cards have been attempted**. *Statistics* are **automatically shown** upon completion of a *Game* (see Fig. above).

The user can choose to **stop** a *Game* before it has finished. This will result in all current *Game* progress being lost, and no *Statistics* being collected or generated (see Fig. above).

The relevant command(s) are:

1. Stopping a *Game* (before it has finished):

Format: **stop**

- Forcibly terminates the current active *Game* session (**all progress will be lost, and no *Statistics* will be reported.**)

6. Contributions - Developer Guide

Given below are the highlights I contributed to the Developer Guide, showcasing my ability to write technical documentation and the technical depth of my contributions to the project.

6.1. Architecture

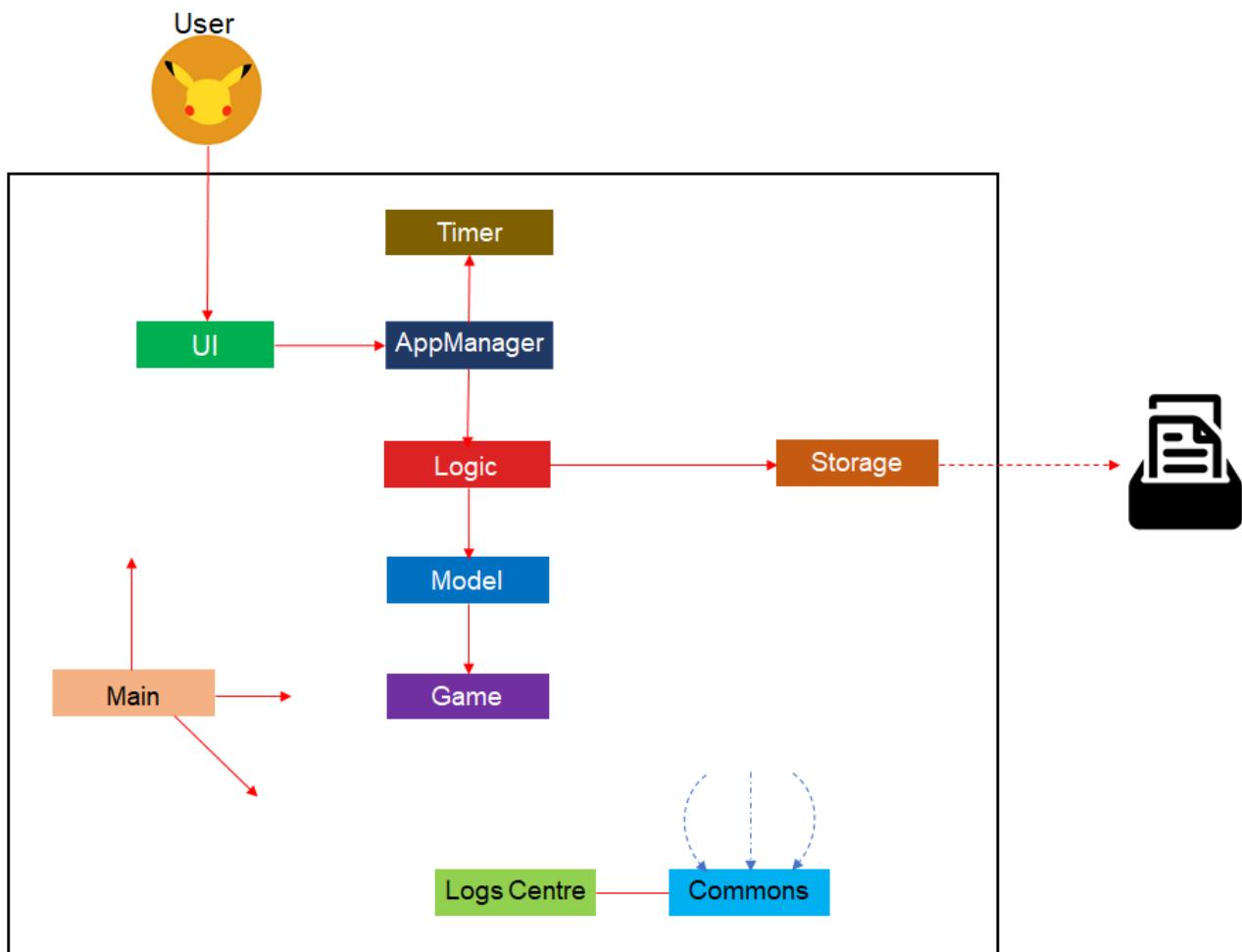


Figure 14. Dukemon Architecture Diagram

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* components.

- **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**.

6.2. AppManager component

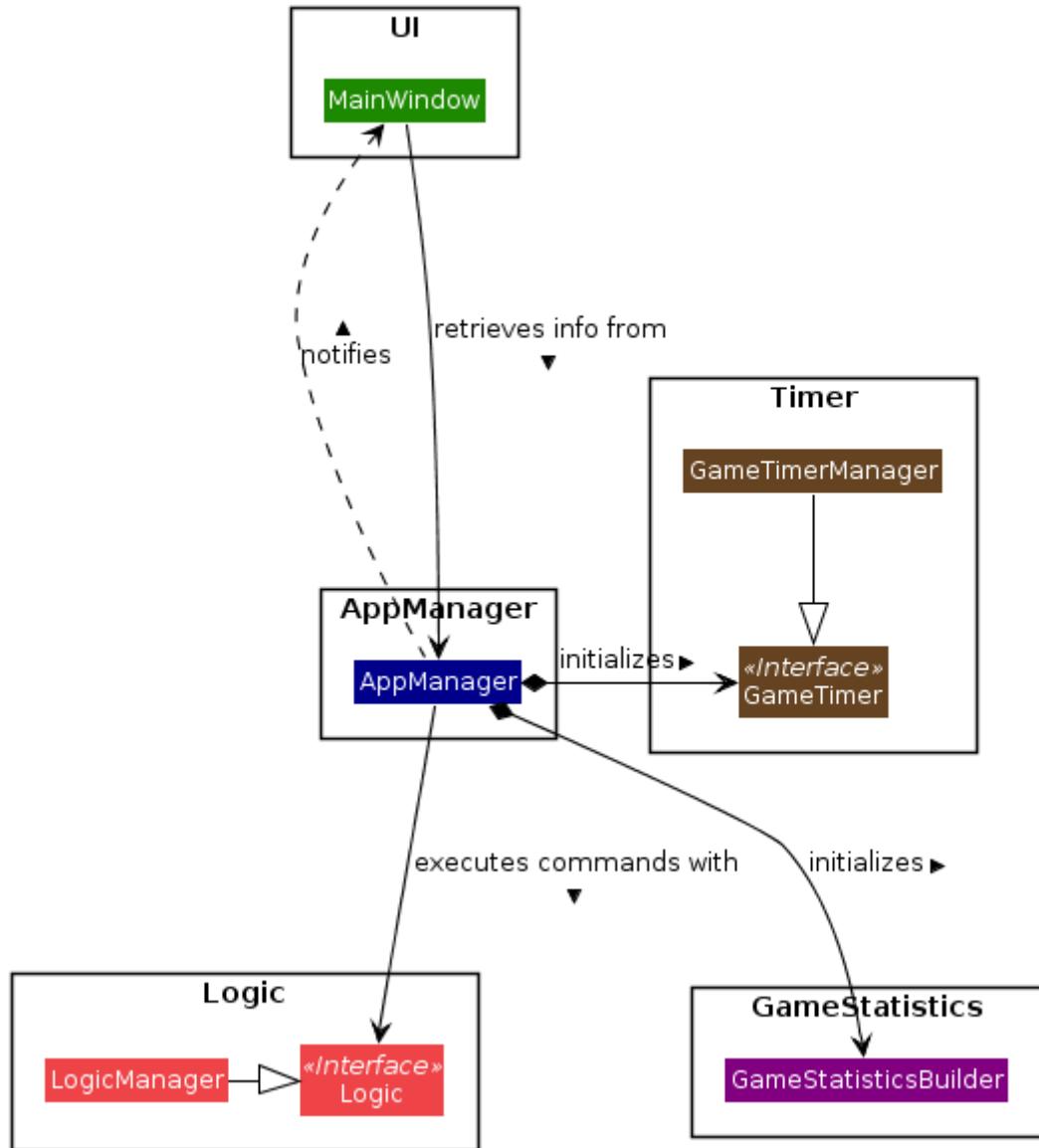


Figure 15. Structure of the *AppManager* Component

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**.

6.3. Timer component

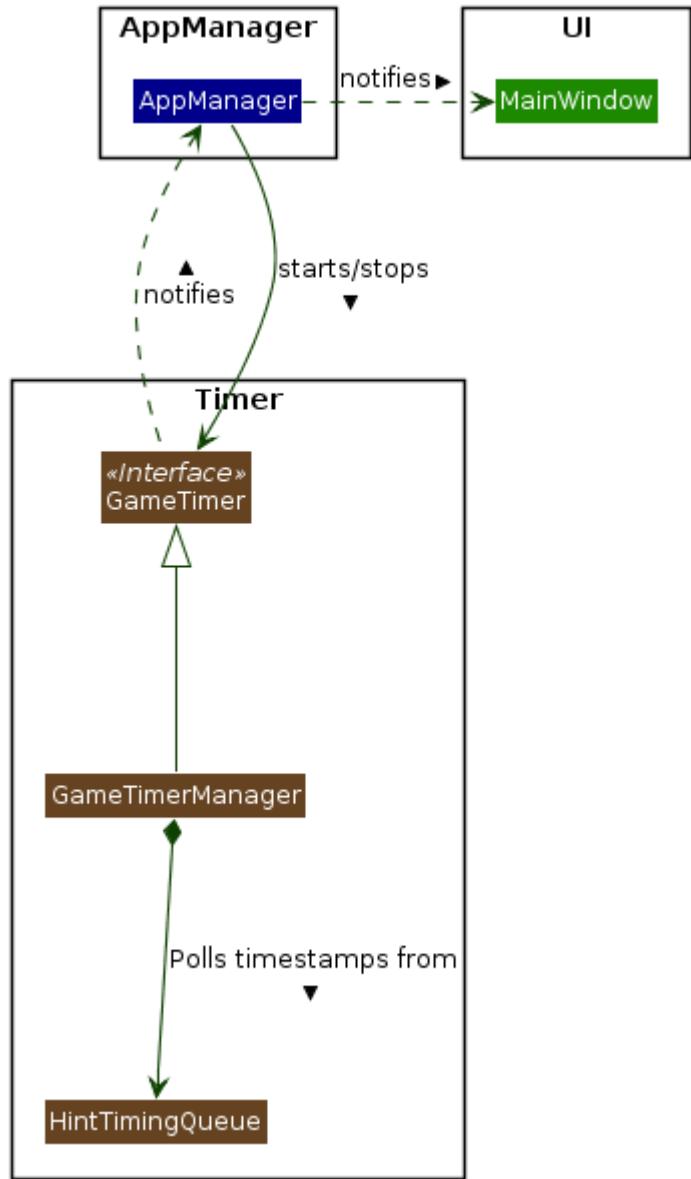


Figure 16. Structure of the Timer Component

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.*

6.4. Game component

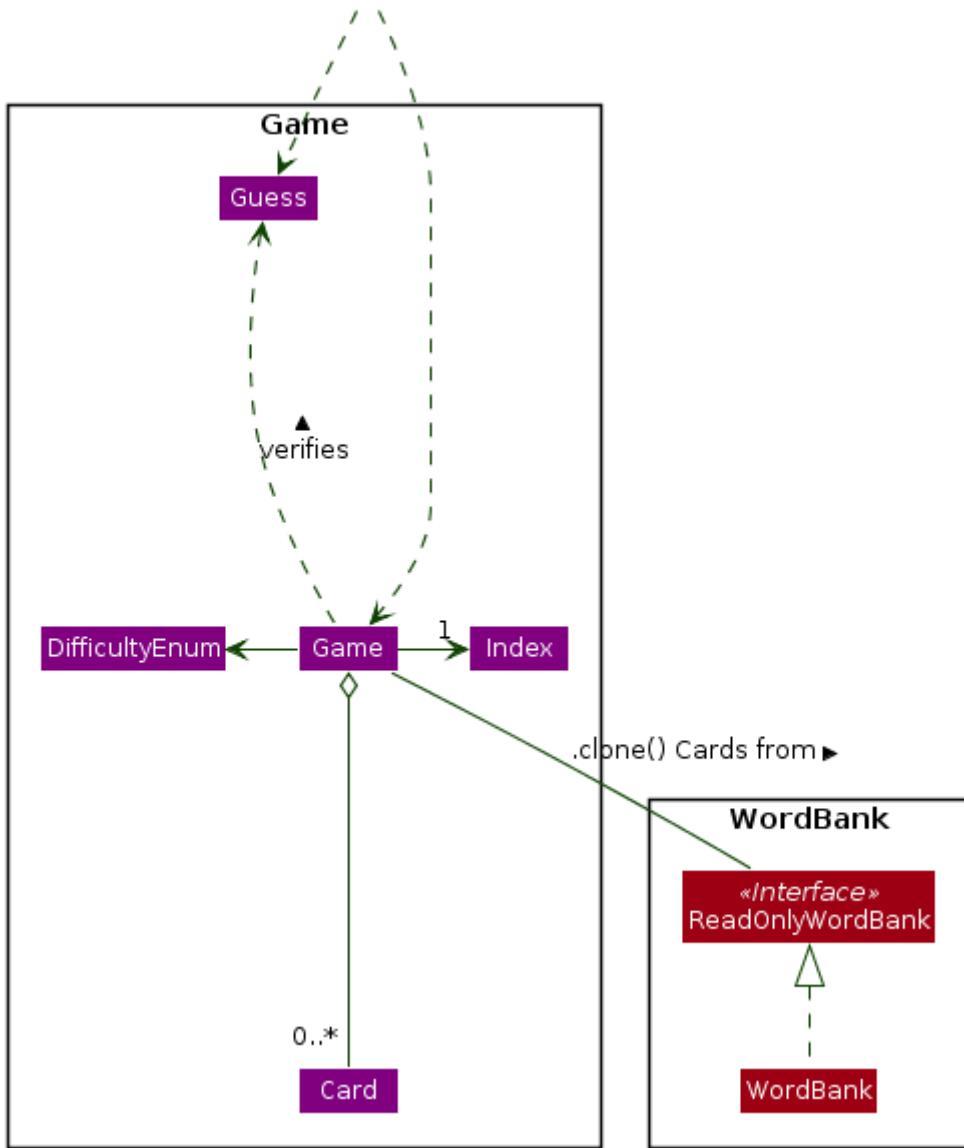


Figure 17. Structure of the Game Component

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)

6.5. Timer-based Features

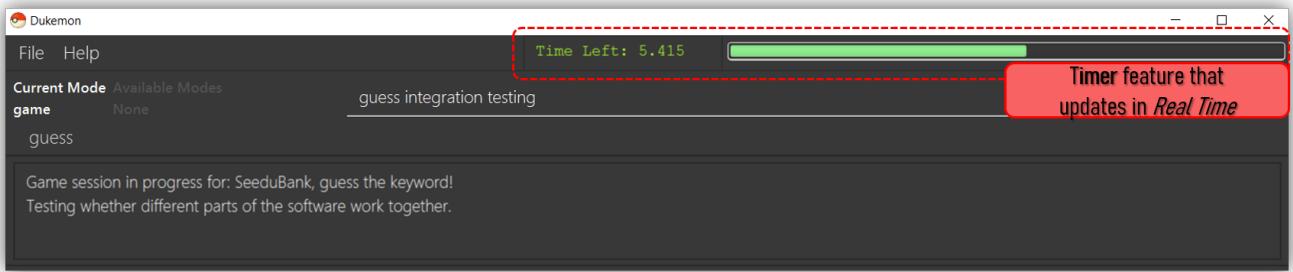


Figure 18. Screenshot of the **Timer** component in action.

6.5.1. Implementation Overview - Timer

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 50ms).

An *Observer Pattern* is loosely 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()`.

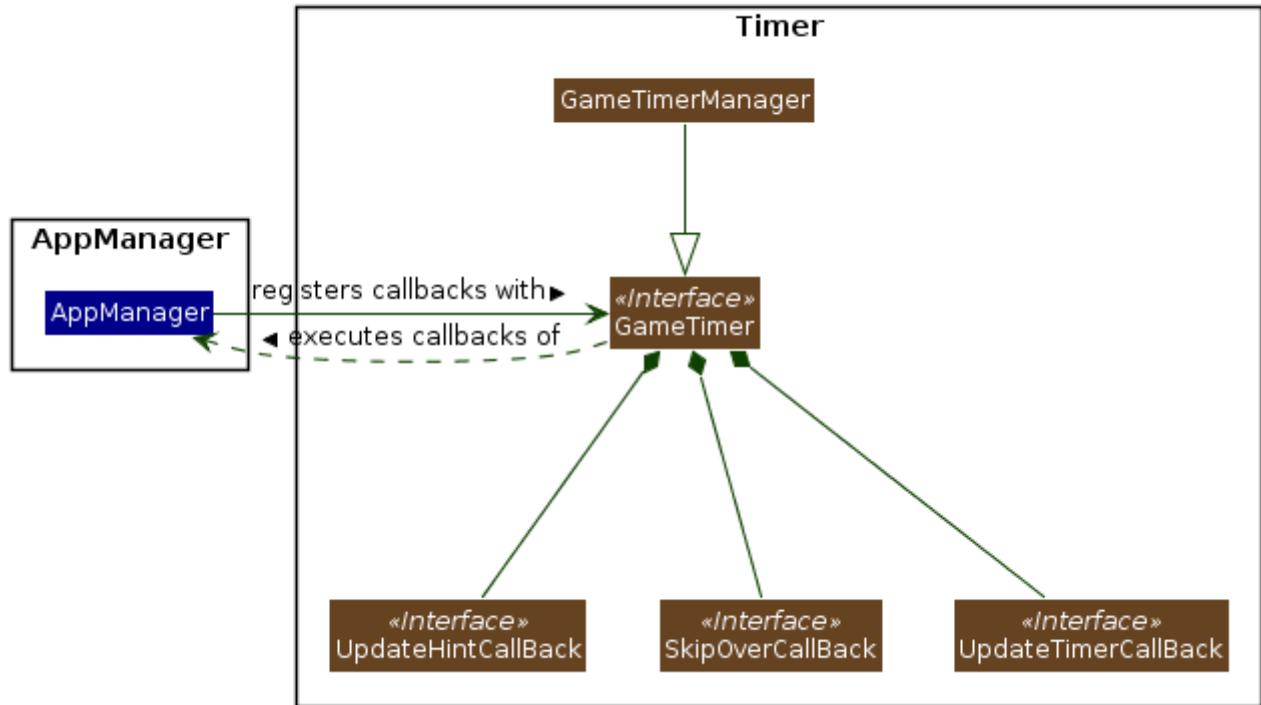


Figure 19. Class diagram reflecting how the callback-functions are organized in the **Timer** component.

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`.

6.5.2. Implementation Overview - Hints

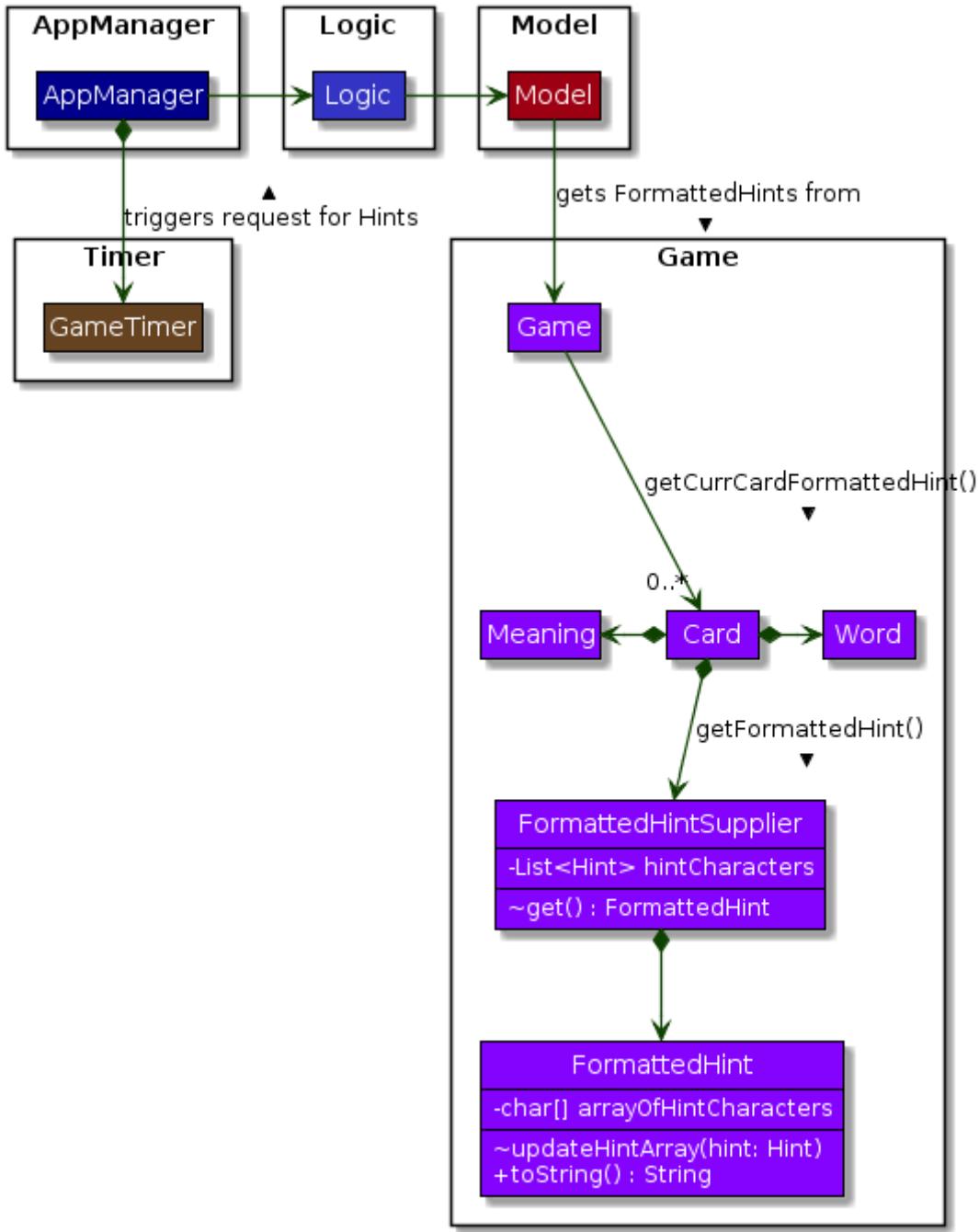


Figure 20. Class Diagram showing structure of **Hints** and its relationships to other components. (Some details omitted)

In order to display the **Hints** component to the user in a *Hangman-esque* style, **string formatting** has to be performed.

- Each **Card** contains a **FormattedHintSupplier** that supplies **FormattedHints** ready to be shown to the user.
- Each **FormattedHintSupplier** contains a **FormattedHint** that is periodically updated.
- Each **FormattedHintSupplier** contains a **java.util.List** of **Hint** to update the **FormattedHint** with.
- Each **Hint** encapsulates a **Character** and an **Index** which the **Character** is to be shown in the **FormattedHint**.

The **Timer** component **triggers a request** to the **AppManager**, who then updates and retrieves the

FormattedHint from the current Game via the Logic component.

6.5.3. 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.

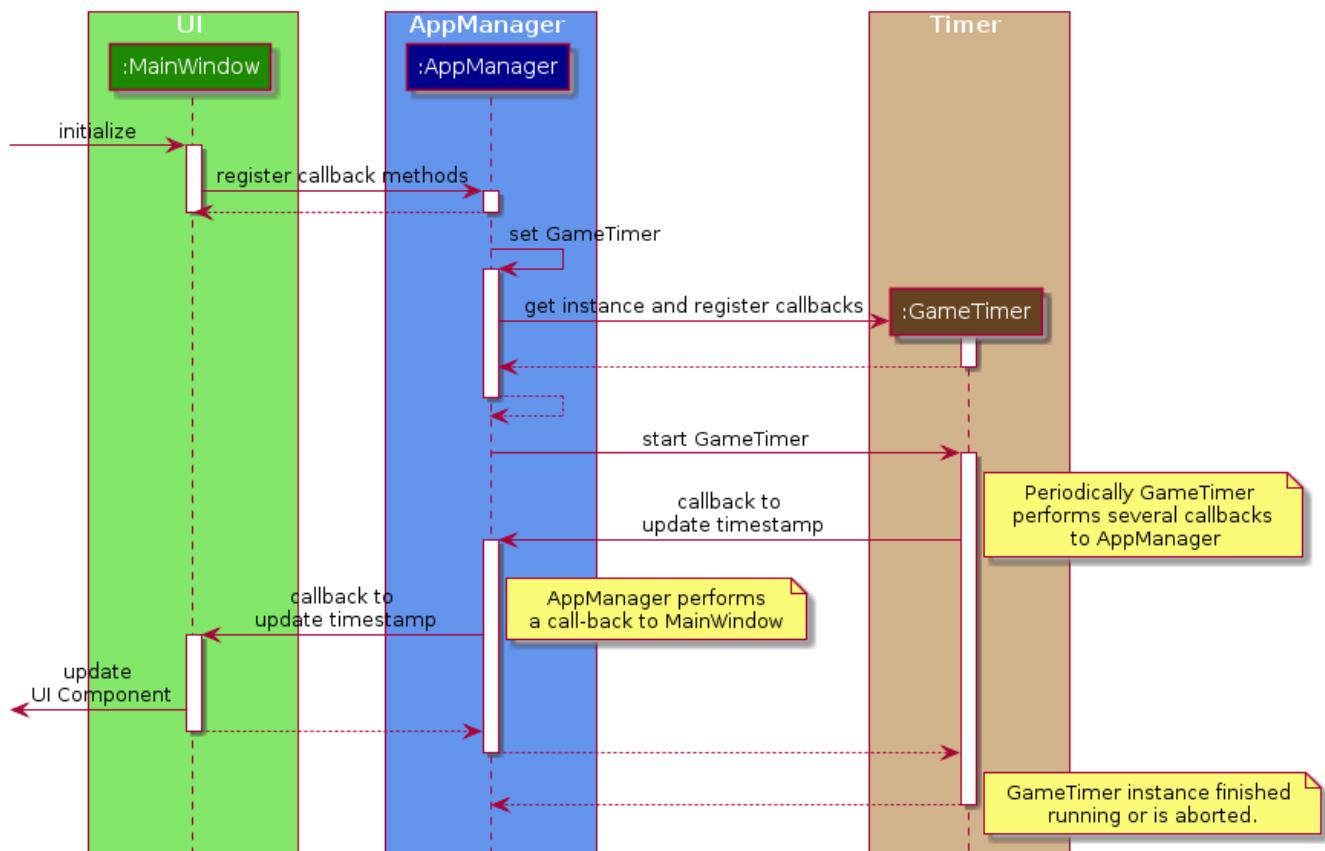


Figure 21. Sequence diagram (with some details omitted) describing the flow of registering and executing callbacks between the different components

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.

6.5.4. Flow of Events - **Hints** Enabled

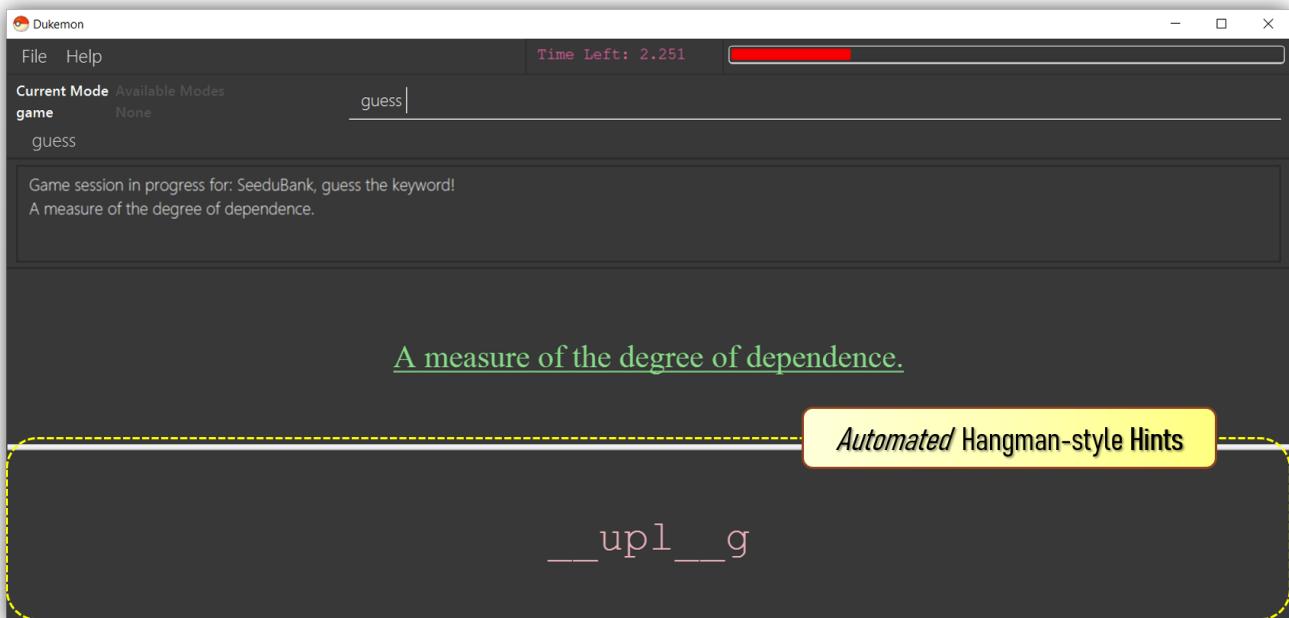


Figure 22. Screenshot of the automatic **Hints** feature in action.

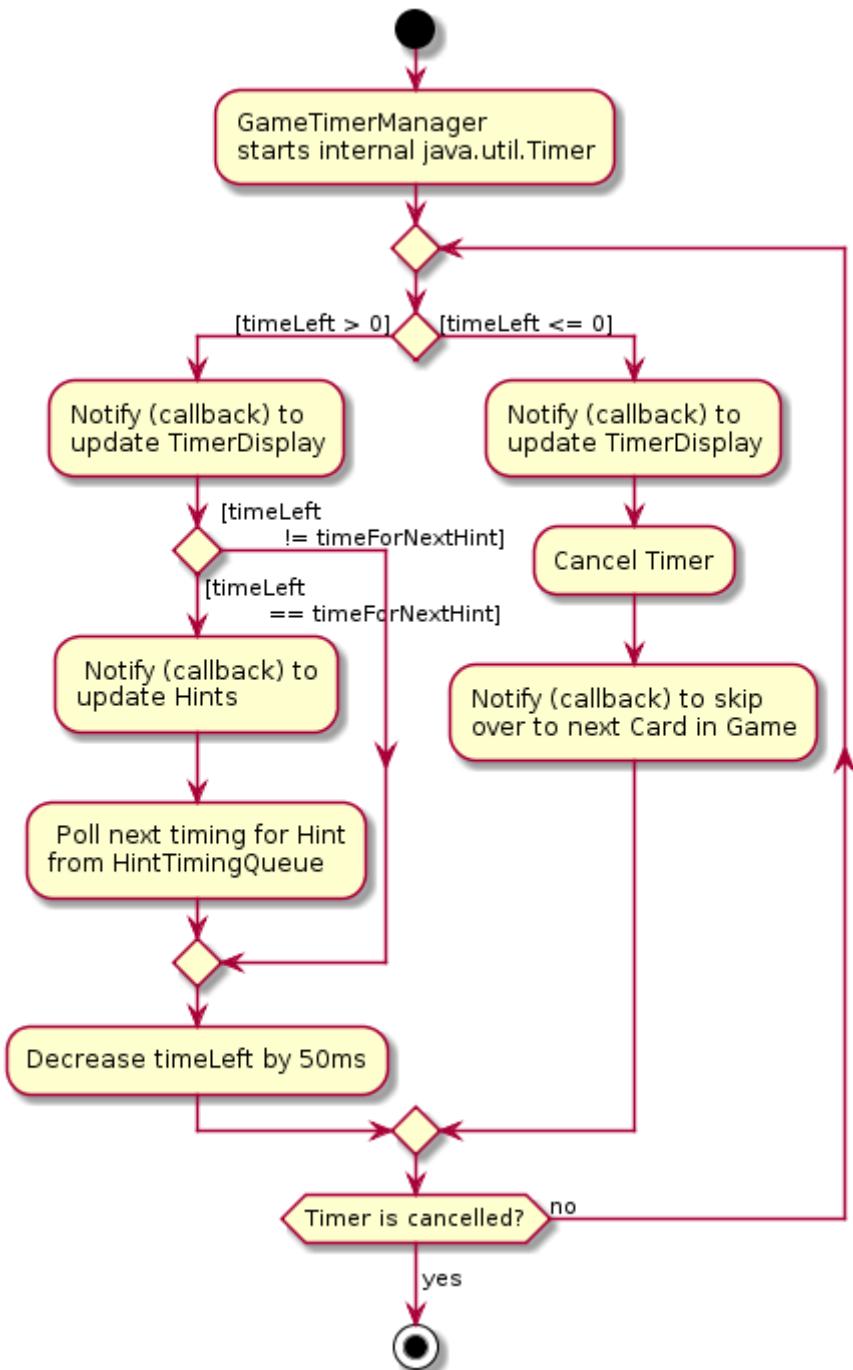


Figure 23. Activity diagram of the `run()` method of an instance of `GameTimerManager` when `Hints` are enabled.

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 50 milliseconds 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 50ms.

6.5.5. 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 <code>Ui</code> and other components when the <code>Timer</code> triggers an event. | Holding a reference to <code>Ui</code> and other components directly inside <code>GameTimer</code> itself: <i>Pros:</i> Straightforward and direct, can perform many different tasks on the dependent components. <i>Cons:</i> Poor abstraction and high potential for cyclic dependencies, resulting in high coupling. | Using <i>Functional Interfaces</i> as Call-backs to notify components indirectly. <i>Pros:</i> Maintains abstraction and minimal coupling between <code>Timer</code> and other components <i>Cons:</i> Relies on developer to register correct call-back methods with the <code>Timer</code> . 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.

7. Other Works

- [Linked-in Profile]