Koh Yi Da  - Project Portfolio

Project: Dukemon Github: Dukemon RepoSense: Code

1. Overview of *Dukemon*

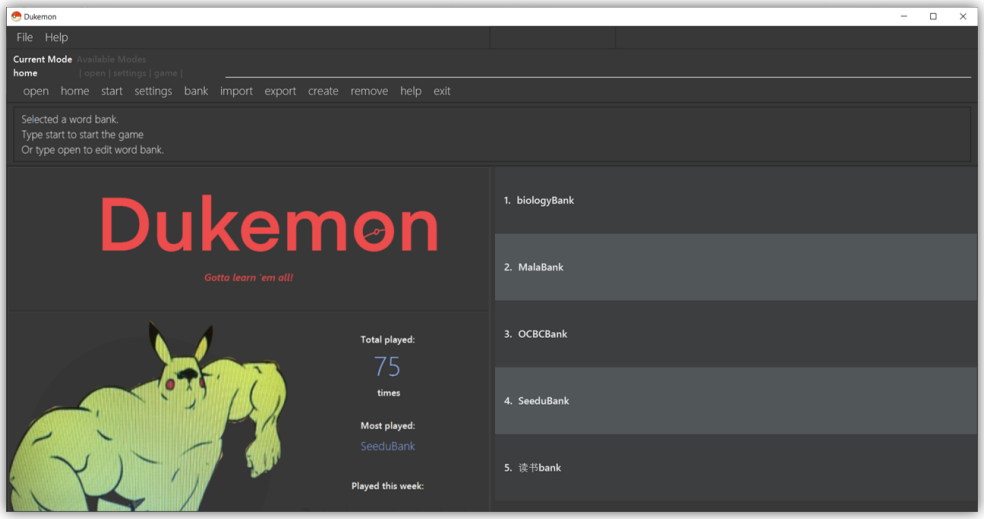

Figure 1. The Home Page of Dukemon upon initial start-up.

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

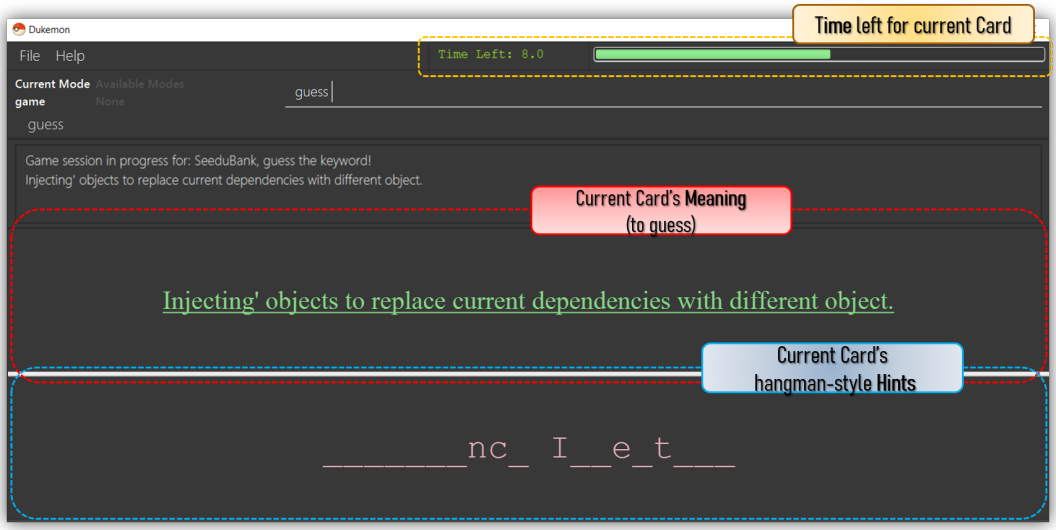

Figure 2. UI of Dukemon in Game Mode, showing the dynamic Timer and Hint features I implemented.

- **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 of the app 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` - `v1.4` (3 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 navigability: #137
    - Edited and wrote Introduction, Installation, and Quickstart sections in **User Guide**: #149
    - Drew and explained overall architecture of *Dukemon* in **Developer Guide** #94
    - Oversaw and ensured quality and cohesiveness of **User Guide** and **Developer Guide**. #226
- *Community*:
    - Reviewed PRs (with non-trivial review comments): #49, #71
    - Contributed to the module forum (example: 1)
- *Tools*:
    - Integrated external testing library (TestFX) to the project (#79) and Travis CI builds (#113).

3. 📖 Contributions - User Guide

Below are the highlights of my contributions to the User Guide, showcasing my ability to write documentation targeting end-users.

3.1. Game Commands 🎮

(Available in Game mode)

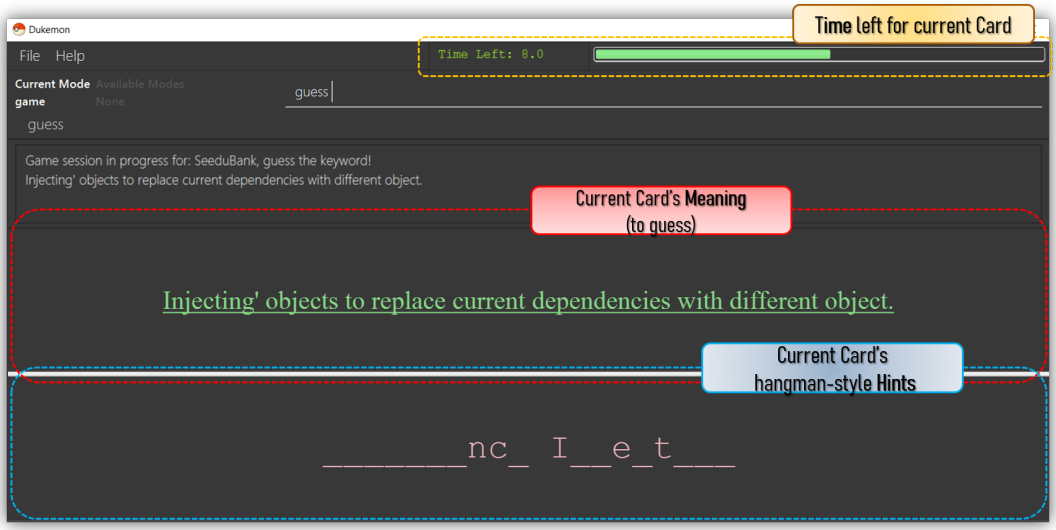

Figure 3. UI regions that are relevant when a Game session is in progress.

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*.

3.1.1. 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*. (*WordBank* selection is done in *Home* mode.) If no *Difficuty* is specified, the default *Difficulty* in *Settings* will be used.

3.1.2. Game Mode - Playing

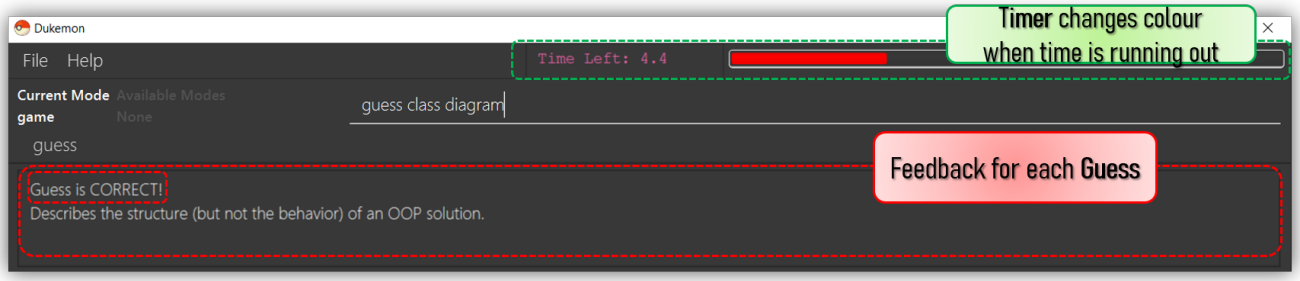

Figure 4. Timer and feedback for each Guess during a Game session. (Timer changes color based on time left).

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**)

3.1.3. Game Mode - Terminating & Statistics

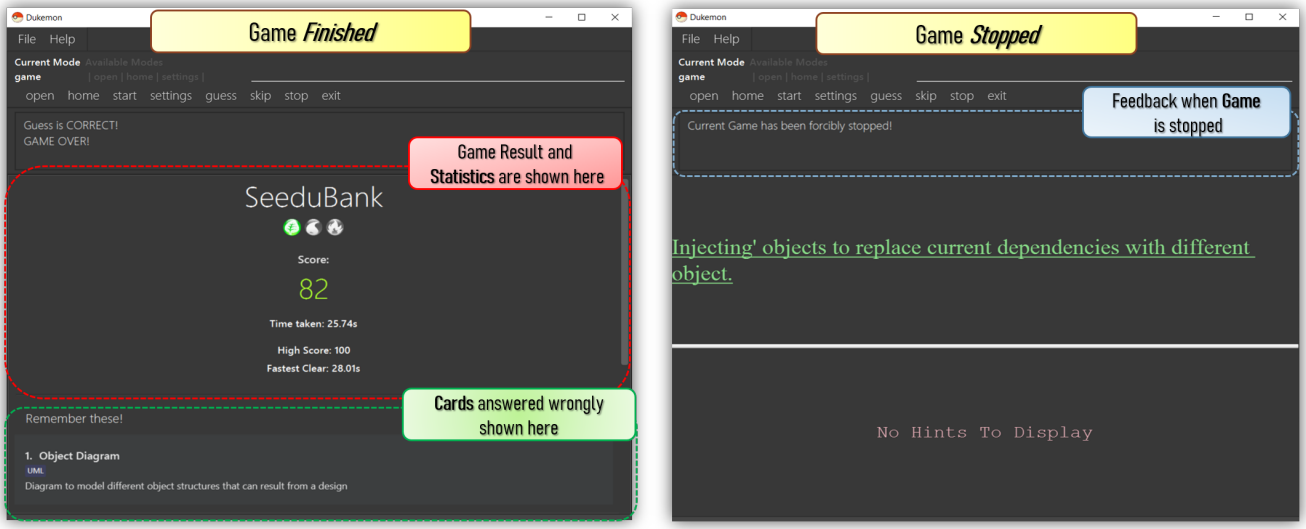

Figure 5. Comparison of UI Regions between Game finished vs. Game forcibly stopped.

A *Game* finishes when **all *Cards* have been attempted**. *Statistics* are **automatically shown** upon completion of a *Game* (see Fig. 6 above). The user can choose to `stop` a *Game* before it has finished- **all current *Game* progress is lost**, and **no *Statistics* are collected** (see Fig. 7 above).

The relevant command(s) are:

1. **Stopping a *Game* (before it has finished)**:
   Format: `stop`
   - Forcibly terminates the current active *Game* session.

4. Contributions - Developer Guide

Below are the highlights my contributions to the Developer Guide, showcasing my ability to write technical documentation and the technical depth of my contributions to the project.

4.1. Timer-based Features

4.1.1. 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 relies on using *Functional Interfaces* as *callbacks* for the TImer to periodically notify other components in the system without directly holding a reference to those components.

Internally, the `Timer` works by using the method `java.util.Timer.schedule()` that schedules `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 by the `Timer`.

NOTE	To avoid synchronization issues, all *callbacks* to change `UI` components are forced to run on the **JavaFX Application Thread** using `Platform.runLater()`.

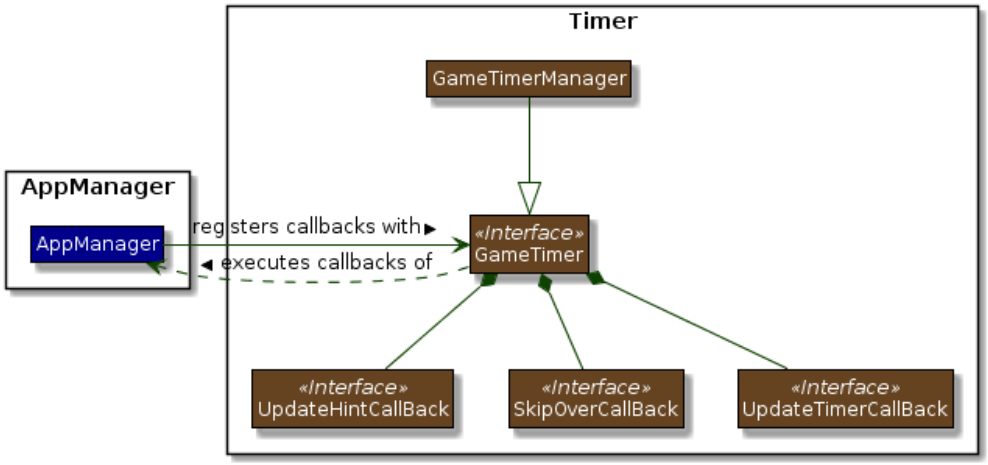

Figure 6. Class diagram reflecting how the callback-functions are organized in the Timer component.

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`.

4.1.2. Implementation Overview - Hints

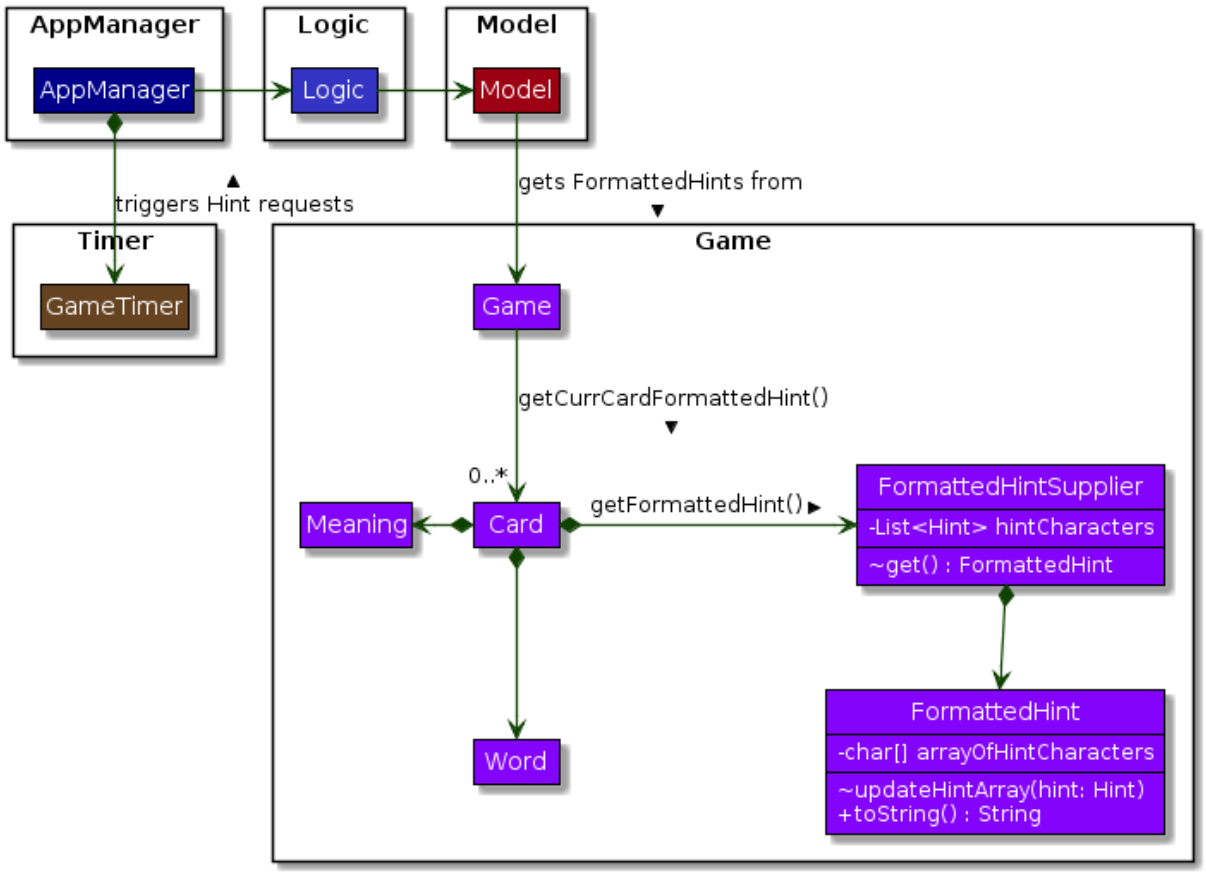

Figure 7. Class Diagram showing structure of `Hints` and its relationships to other components. (Some details omitted)

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 `FormattedHint` maintains a `char[]` array that it's `toString()` method uses to format the output `Hint` string 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 update `Hints`** to the `AppManager`, who then updates and retrieves the updated `FormattedHint` from the current `Game` via the `Logic` component.

4.1.3. 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**.

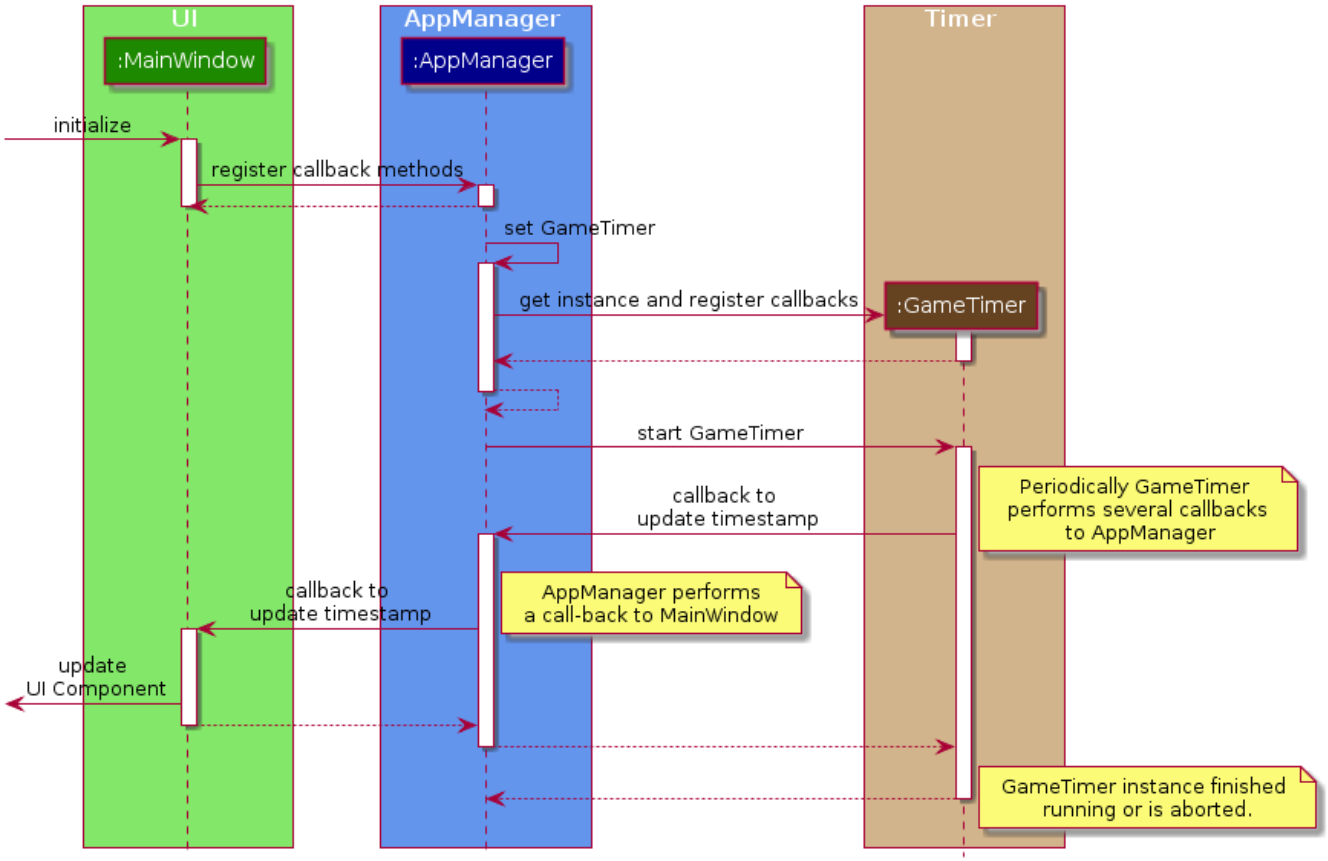

Figure 8. Sequence diagram (with some details omitted) describing the flow of registering and executing callbacks between the different components

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

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.

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`.

4.1.4. Flow of Events - `Hints` Enabled

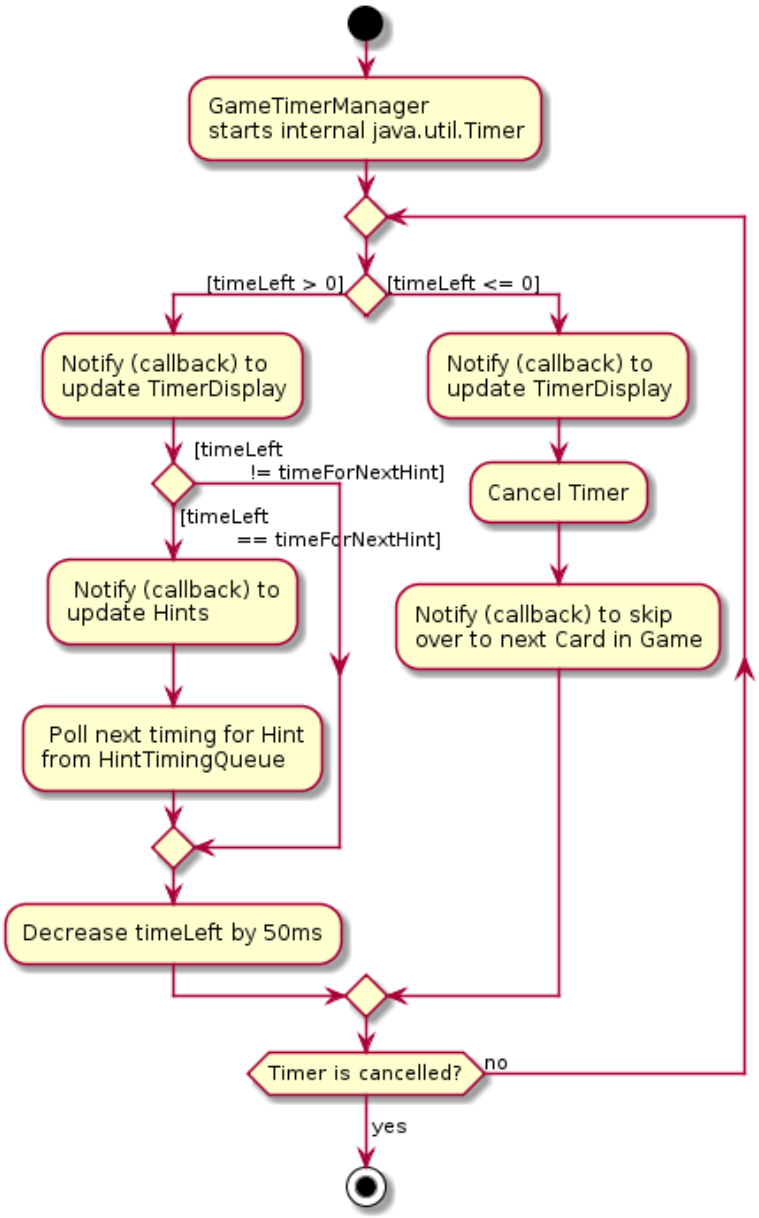

Figure 9. Activity diagram of the `run()` method of an instance of `GameTimerManager` when `Hints` are enabled.

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

In the 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.

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.

4.1.5. Design Considerations

There were a few reasons 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 we chose 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.		
	Alternative 1	**Alternative 2**
Aspect 2: Where and how to perform string formatting for `Hints` to be displayed.	**Move retrieval of individual Hint characters and all formatting outside of the Game component completely**: ***Pros:*** Maintains immutability of each `Card` inside `Game` component. ***Cons:*** Breaking abstraction as higher level components should not have to deal with string formatting.	**Perform formatting at the lowest level possible, using a `FormattedHint` class.** ***Pros:*** Higher level components need not know about string formatting at all, maintains good abstraction. ***Cons:*** Individual `Game` components like each `Card` become stateful, need to make deep copies to prevent state from carrying across `Game` sessions.
Why we chose Alternative 2: Implementing cloning of `Cards` affects other areas of code the least, and reduces unnecessary coupling. Since changes to higher level elements can potentially affect all other components, it was safer to modify more atomic areas of code.		