

RUST RUNNER [V.0.5]

System Design Document (SDD)

This is an small snapshot of the current design of RUST RUNNER, as of the 5th December, 2025. This document is a summarized understanding of the architecture, classes, and how the system interact with itself done by us, **Group 16**.

Developers

- Rayan Ahmad : RayanAhmad123
 - Philip Hasson : ChangIkJoong
 - Nadir Morabeth : nadirmorebytes
 - Oscar Bergdahl : jojk1
 - Jannah Francine Rosales Beato : wthjaaa
-

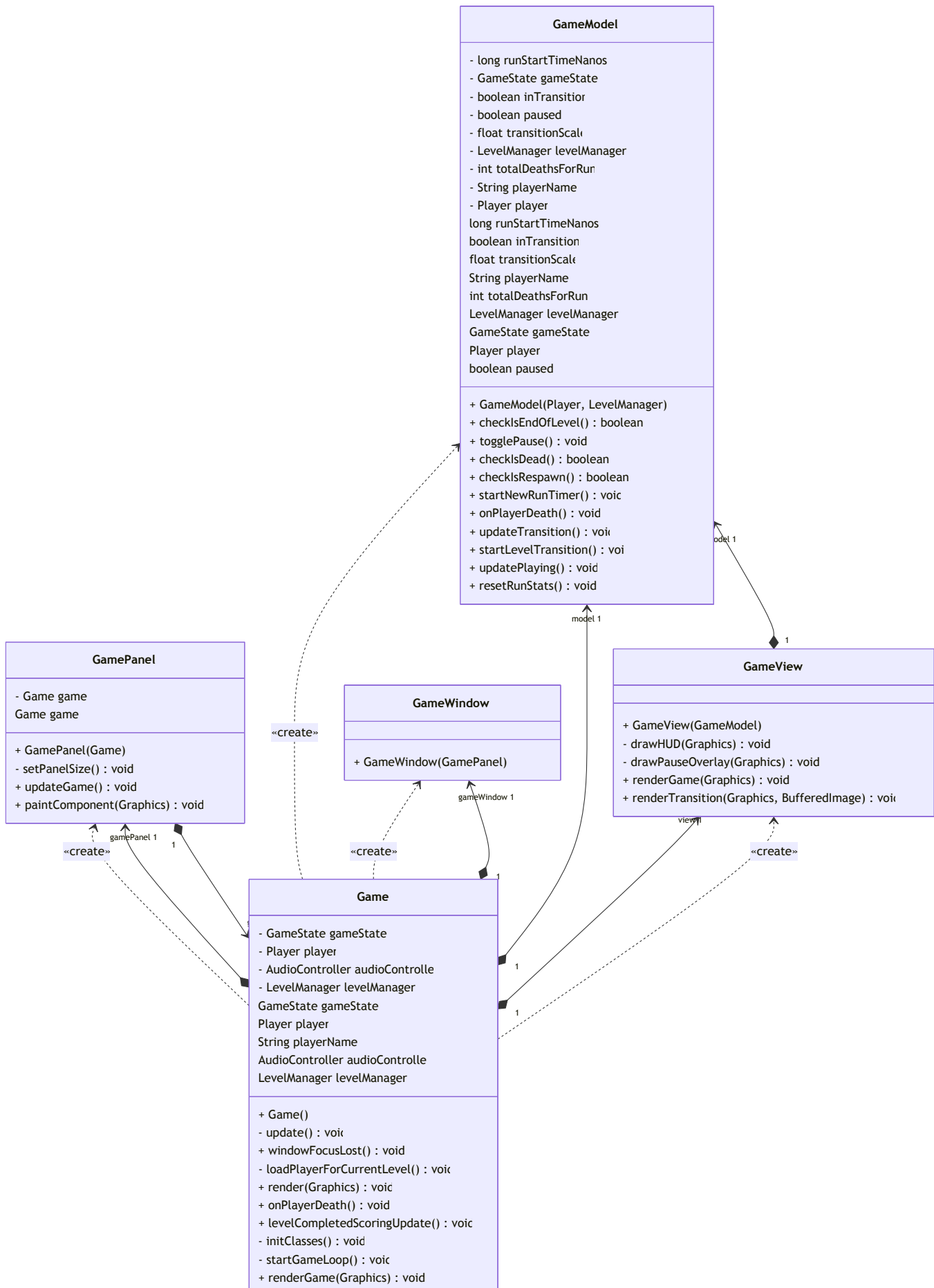
1. Architectural Overview

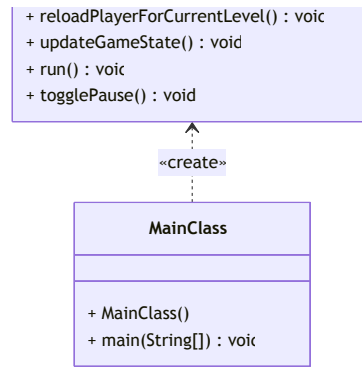
The game is meant to be structured around **MVC architecture** with additional patterns. The **MVC** pattern in our architecture consiting of:

- **Model**
 - **GameModel** : The central **GAME** model for our project, implementing the model for our game.
 - **Player**, **Level**, **LevelManager** are also part of it, with entity classes under **entities** directory and the level design and management in the **Levels** directory.
 - **View**
 - **GameView** : Renders only the game environment, HUD, pause overlay, and transition.
 - **GamePanel**, **GameWindow** : The Swing UI components, and window setup for the game.
 - Main Menu, leaderboard and level-selection are renderers under **main.states**, abstracted with the **GameBaseState** using the **State pattern**.
 - **Controller**
 - **Game** : The main **Controller** unit and game loop, controlling and composing together the **Model** and **View**, along with the different states and observers in the game.
 - **GameBaseState** : An abstract of the concrete states implemented including the **PlayingState**, **MenuState**, **LeaderboardState**, **LevelSelectState**. These emulate the **State pattern**, also communicating directly with other Objects such as the **Singleton Pattern** of **AudioController**, enriching the system with audible haptic feedback and sound.
 - **KeyboardInputs**, **MouseInputs** and **inputs.commands.*** use a **Command pattern** for input handling.
 - **events** : Directory includes several **Observer Pattern** Event Listeners, which are meant to be implemented to act as interfaces between **Game** and different sub-classes.
-

2. Core Class Diagram (Overview)

Here are a few UML-style class diagrams for the core architecture. They are not complete and have been abstracted to easier understand the key responsibilities and relations in our application software.





3. MVC Responsibilities

Repeating a bit but a more in-depth of why we consider and what we consider for each of the Model-View-Controller components. Adding a bit of examples as well.

3.1 Model Component

Key classes:

- **GameModel**
 - Stores and process most of our game states, the model of the MVC pattern.
 - It handles and communicates with the other classes:
 - Death and respawn detection (`checkIsDead()`, `checkIsRespawn()`).
 - End of a level detection: (`checkIsEndOfLevel()` via `player.hasReachedLevelEnd()`).
 - Level transitions: (`startLevelTransition()`, `updateTransition()`).
 - Pause state and routing of updates.
 - Run statistics used for other classes such as `playerName`, total deaths, total spent time.
- **Player**
 - The **Player** class is meant to encapsulate the movement, physics, collisions (via `HelpMethods` and current `Level`).
 - Has information for example about `hitbox`, velocities, jump and air state.
 - Uses `isOnLevelEnd(...)` and sets `reachedLevelEnd`. It itself does not trigger scoring or transitions directly.
 - Is meant to notify **Game** about deaths through a `PlayerEventListener` instead of holding a direct reference to **Game**, everything with **Observer Pattern** at this stage is however work-in-progress.
- **LevelManager**
 - Builds all levels from images and text resources using `LoadSave` and `LevelConfigLoader`.
 - Maintains `currentLevelIndex` and the list of levels `List<Level>`.
 - Uses `update()` that send the information to the current `Level` to update platforms, spikes, spawn platform.
 - Exposes `resetToFirstLevel()` used when returning to the main menu.

- **Level**

- Stores all of the tile data arrays for ground, obstacles and objects.
- Manages triggers, moving platforms, spikes, and death sprites.

Together these forms the model itself of the game world along with it's rules.

3.2 View Component

Key classes:

- **GameView**

- Responsible for rendering the GUI based on **GameModel**, parsing it downwards to it's respective subclasses:
 - Background, tiles, objects (**LevelManager.draw().***).
 - Currently renders player via the **Player** class: **Player.render(g)**.
 - HUD: current level index and player death count.
 - Pause overlay when **model.isPaused()**.
 - Transition effects using **model.getTransitionScale()**.

- **GamePanel**

- Swing **JPanel**: sets the size, attaches some of the input listeners, and currently communicates the **paintComponent** to **game.render(g)**.

- **GameWindow**

- Wraps the Swing **JFrame** creation and focuses on its handling.
- Menu and leaderboard views (**MainMenu**, **Leaderboard**, **LevelSelect**) as different states using State pattern, switching between these views interchangeably.
 - Render their own UIs and rely on **Game** | **LevelManager** | **GameModel** for data.
 - **Leaderboard** reads scores from **LoadSave.readScoreFile()** | **updateScoreFile()** and renders per-level the top 5 entries only to not clutter the leaderboards too much.

The view layer is meant to never change any core game rules or state, it is meant to only input the model or controller and draw this accordingly.

3.3 Controller Component

Key classes:

- **Game**: the central controller and application **lifecycle** owner.
 - **Responsibilities**:
 - Construct and tie together the full application with model, view, audio, input, and states.
 - Run the main loop (**run()**), calling **update()** and initializing the redrawing of the GUI.
 - Communicate updates to the current **GameBaseState**.
 - Bridge the model events (switching states) to "side effects" such as audio via **AudioController**, initiates the spawn platform reset, and to be querying leaderboard

entries by communicating for example level-completed event (via `LevelCompletedListener`) instead of calling hard-coded method.

- Manage the game's different states via the `setGameState(GameState newState)`.
- React to low-level model callbacks, for example implement `PlayerEventListener` so `Player` can report deaths without depending on `Game`.

- **States**

- `GameBaseState` abstract base: holds reference to `Game` and defines `update()`, `render(g)`, `onEnter()`, `onExit()`.
- `PlayingState`: calls `game.updateGameState()` | `game.renderGame(g)`, and is one of the responsables for switching background music (for the correct state) track using `onEnter()`.
- `MenuState`, `LeaderboardState`, `LevelSelectState`: does similar, switches menu/leaderboard/level-select views into the state machine. `MenuState` starts menu music using `onEnter()`, similarly to `PlayingState`.

- **Inputs & Commands**

- `KeyboardInputs`:
 - On key events, creates or dispatches `Command` objects (`MoveLeftPressCommand`, `JumpPressCommand`, `TogglePauseCommand`, etc.).
- Each `Command` implements an `execute()` that calls into `Game` or `Player`, serving as an abstraction layer in-between the component(s).
- `MouseInputs` also communicates mouse clicks to other UI elements (buttons, level select, etc.) via the relevant state, part of this implementation was due to the importance of the haptic feedback from our user stories.

And to add some more things,

OBSERVER PATTERN IS ABSOLUTELY NOT DONE, AND MOST OF THE ABSTRACTION IS 100% WORK IN PROGRESS.
