CITX1401 – Computational Thinking with Python

Project Documentation

**Game Title: Echoes of the Nebula**

**Student Name: Md Salman Ali**

Contents

[Overview 1](#_Toc1170930853)

[Design 2](#_Toc771658385)

[Story Design 3](#_Toc1995666062)

[Program Structure 3](#_Toc321661421)

[Algorithm Design 5](#_Toc107606017)

[Additional Features 12](#_Toc62614930)

[Missing Functionality 13](#_Toc1392469457)

[Extra Features 13](#_Toc259480574)

# Overview

A brief description of the game’s concept and theme

"Echoes of the Nebula" (EON) is an interactive, text-based adventure game with a science-fiction theme. The player assumes the role of a lone technician who awakens from cryo-sleep aboard a silent, adrift research vessel. The ship is trapped within a mysterious, living cosmic nebula that has seemingly caused the disappearance of the crew.

The game's core concept is one of isolation, mystery, and discovery. The player must restore the ship's systems, explore the deserted decks, and confront strange entities corrupted by the nebula's energy. Through their choices, the player will unravel the central mystery and ultimately decide the fate of both themselves and the vessel, leading to multiple distinct endings. The game is built entirely in Python 3 using only the standard library, featuring a dynamic terminal-based UI with ASCII art, colored text, and a responsive design to create an immersive experience.

The project is version-controlled with Git, and the [full source code can be viewed on a private GitHub repository](https://github.com/immortal-forest/eon). Access can be provided for marking and review purposes, and the repository is scheduled to be made public after the submission due date.

# Design

## Story Design

A design for the story that the game will tell.

The story framework, including the title, setting, and the main branching paths, was devised from the initial project brief. This framework was then fleshed out with original dialogue, detailed scene descriptions, and the specific choices available to the player.

The atmosphere of isolation aboard a derelict craft is drawn from classic Western-style sci-fi, such as the movie "Alien" and the videogame "System Shock", and the social deduction game "Among Us". The high-concept themes of encountering an immense, non-human intelligence were influenced by the philosophical implications of Orson Scott Card's "Ender's Game". In addition, the nebula as a treacherous, transformative environment that affects all who enter it has strong thematic connections to the titular chasm in the Japanese manga, “Made in Abyss”. These sources informed the game's tone, pacing, and the design of its user interface.

**1. The Beginning (Cryo-chamber and Terminal):** The game opens with the player awakening in the cryo-chamber. The initial objective is to restore power by accessing a nearby terminal, which introduces the player to the game's choice-based interaction system.

**2. The Main Hallway (First Branching Choice):** After exiting the cryo-chamber, the player enters a central hallway, which serves as a hub. Two paths are presented: the Maintenance Deck and the locked Bridge, forcing the player to explore to find a way forward.

**3. The Maintenance Deck (Item Collection):** To progress, the player must explore the Maintenance Deck to find a *data\_key*. This section introduces the item collection mechanic, a core requirement for solving puzzles and accessing new areas.

**4. The Bridge and Combat:** Using the *data\_key*, the player gains access to the Bridge, where they encounter the first hostile "Glitch-Entity". This scene introduces the turn-based combat system (attack/defend), another of the game's advanced features.

**5. The Major Choice (The FTL Drive vs. The Nebula):** After defeating the entity, the player reaches the ship's command console and is faced with the story's major branching point:

* **The Escape Path:** Prioritize survival by repairing the ship's FTL drive.
* **The Discovery Path:** Investigate the nature of the nebula to understand what happened to the crew.

**6. The Escape Path:** This branch focuses on item collection and a random combat encounter in the Cargo Bay to find a *power\_regulator*. The path culminates in a final choice that determines whether the escape is successful.

**7. The Discovery Path:** This branch is narrative-focused, leading the player to establish communication with the nebula, which is revealed to be a conscious entity. The player is presented with a final choice that determines their ultimate fate.

## Program Structure

Pseudocode and/or Diagrams showing the design of the structure of the program’s code (e.g. functions and modules).

**Diagram of Modules:**

EON/

.

├── data # Static game assets and data models

│ ├── exceptions.py

│ ├── game.json # Game script

│ ├── handler.py # Functions for loading data files

│ ├── init.py

│ ├── models.py # Player, Enemy, Entity classes

│ ├── README.md

│ └── state.json # Saved game data

|

├── main.py # Main game loop and orchestration

|

├── player # Dynamic player state management

│ ├── handler.py # Save/load game functions

│ └── init.py

|

├── pyproject.toml

├── README.md

|

└── term # Terminal display and styling package

├── colors.py # Raw ANSI escape codes

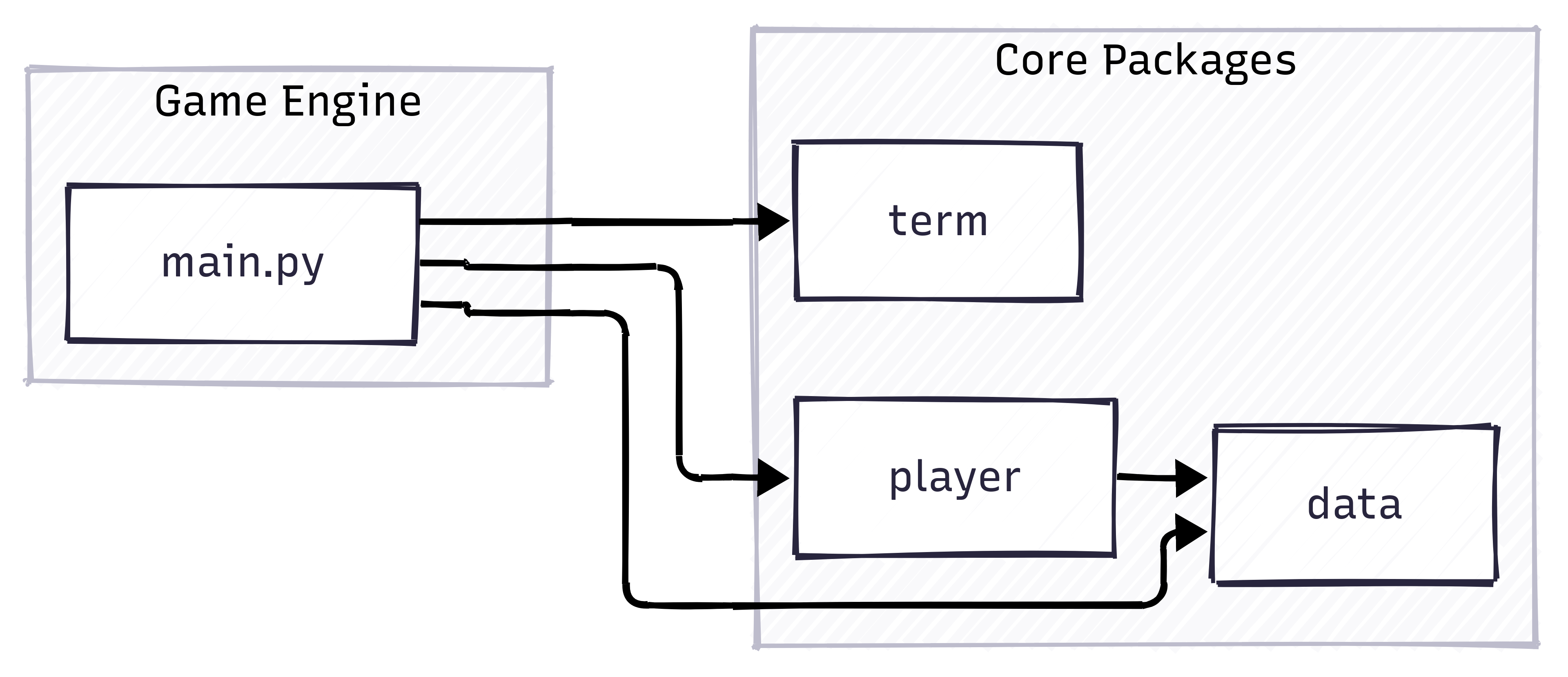
├── init.py

├── sequences.py # Raw ANSI escape codes for cursor

└── terminal.py # High-level display functions

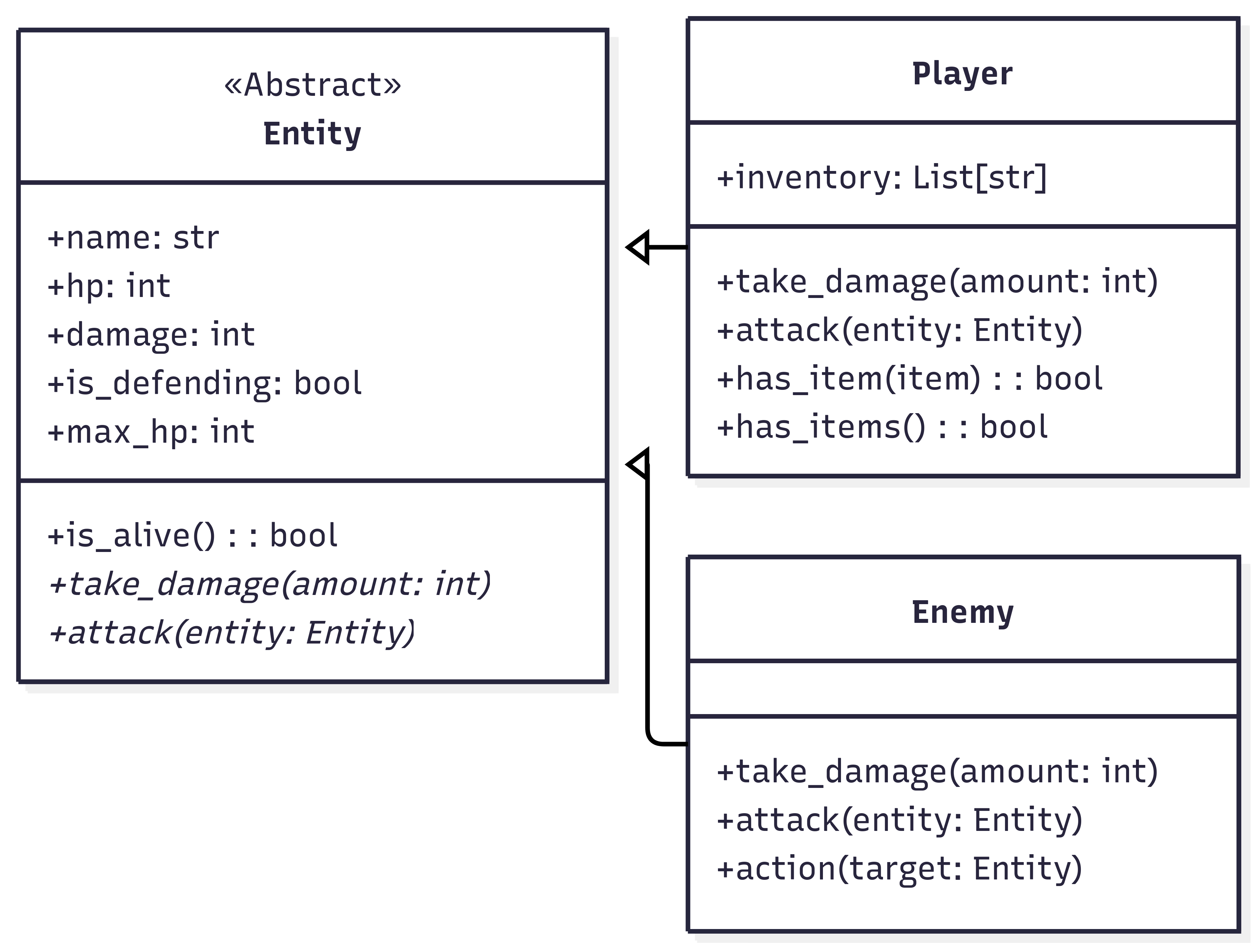
**Component Diagram**

This diagram shows the high-level dependencies between the main packages. The main game engine relies on the three core packages to function. (Made using [mermaid.js](https://mermaid.js.org/))



**Class Diagram**

This diagram illustrates the relationship between the core data models used for combat and player state. Both Player and Enemy are specialized versions of the abstract Entity class. (Made using [mermaid.js](https://mermaid.js.org/))



## Algorithm Design

Analysis, Specification and Pseudocode and/or Diagrams showing the design of any important algorithms used in the program’s code.

**Main Game Loop (main)**

* **Analysis:** The game requires a persistent loop that can handle different states (exploration, combat, menus) without becoming overly complex. The loop must manage rendering, input, and state updates in a consistent cycle.
* **Specification:** A single *while True* loop serves as the game's "heartbeat". The game's state is determined by variables like *enemy* and *scene\_name*. An *if/else* block within the loop checks these variables each turn to decide whether to run exploration logic or combat logic. This avoids code duplication and keeps all special commands (save, load, quit) functional at all times.
* **Pseudocode:**

PROCEDURE main()  
 // --- Initialization Phase ---  
 INITIALIZE Player object  
 LOAD game\_script from JSON  
 SET initial scene\_name from game\_script  
 INITIALIZE empty combat\_log, empty random\_combat list, and set enemy to None  
  
 // --- Main Game Loop ---  
 WHILE True DO  
 current\_scene = GET scene data using scene\_name  
  
 // Check for Random & Fixed Combat Triggers  
 IF scene can trigger combat AND no enemy is active THEN  
 CREATE Enemy object from scene data  
 INITIALIZE combat\_log  
 END IF  
  
 // --- Render Phase ---  
 CLEAR the terminal screen  
 RENDER the top status line (HUD)  
 line\_num\_after\_description = RENDER the main window (description or combat log)  
 valid\_options, line\_num\_after\_options = RENDER the choices for the player  
  
 // --- Input Phase ---  
 player\_action = GET validated input from the user (number or command string)  
  
 // --- Update Phase ---  
 IF player\_action is a command string (e.g., "save") THEN  
 result = HANDLE the command  
 IF command was "load" THEN  
 UPDATE all game state variables from result  
 END IF  
 CONTINUE to next loop iteration  
 END IF  
  
 IF enemy is active THEN // Combat Logic  
 chosen\_action = GET combat action (Attack/Defend)  
 combat\_result = HANDLE one turn of combat  
 IF combat\_result is a victory THEN  
 UPDATE scene\_name to the 'on\_victory' scene  
 RESET enemy to None and clear combat\_log  
 END IF  
 ELSE // Exploration Logic  
 IF player's choice is NOT valid THEN  
 DISPLAY error message and PAUSE  
 CONTINUE to next loop iteration  
 END IF  
 chosen\_option\_data = GET selected option from scene  
 scene\_name = UPDATE game state based on chosen\_option\_data  
 END IF  
  
 // --- End Condition Check ---  
 IF player is not alive THEN  
 DISPLAY "Game Over" message  
 BREAK loop  
 END IF  
 IF scene\_name is a final end state THEN  
 DISPLAY the ending message  
 BREAK loop  
 END IF  
 END WHILE  
END PROCEDURE

**Responsive Status Line (render\_status\_line)**

* **Analysis:** The status line must display information in left, center, and right-aligned sections that adapt to any terminal width. A key challenge is that styled text contains invisible ANSI escape codes that interfere with standard length calculations, causing misalignment.
* **Specification:** The function must calculate positions based on the length of the **plain, un-styled text content** only. It gets the terminal's full width using *os.get\_terminal\_size(**).columns*. The left section is aligned at column 1. The center position is calculated by subtracting the plain text length from the terminal width and dividing by two. The right position is calculated by subtracting the plain text length from the terminal width. The styling is only applied just before displaying.
* **Pseudocode:**

PROCEDURE render\_status\_line(player, scene\_name, enemy)  
 width = GET terminal width  
  
 // Left Section  
 left\_text\_plain = FORMAT player stats (HP, Attack, Items)  
 left\_text\_styled = STYLIZE(left\_text\_plain)  
 DISPLAY left\_text\_styled at (line=2, column=1)  
  
 // Center Section  
 center\_text\_plain = FORMAT scene\_name  
 center\_column = (width - length of center\_text\_plain) / 2  
 center\_text\_styled = STYLIZE(center\_text\_plain)  
 DISPLAY center\_text\_styled at (line=2, column=center\_column)  
  
 // Right Section  
 IF enemy exists AND enemy is alive THEN  
 right\_text\_plain = FORMAT enemy stats (HP, Attack)  
 right\_column = width - length of right\_text\_plain  
 right\_text\_styled = STYLIZE(right\_text\_plain)  
 DISPLAY right\_text\_styled at (line=2, column=right\_column)  
 END IF  
END PROCEDURE

**State-Driven Main Window (render\_main\_window)**

* **Analysis:** The main content area needs to adapt its display based on the game's state. In exploration mode, it should show a narrative description with a typing effect. In combat mode, it needs to display ASCII art and a log of recent events.
* **Specification:** The function will check if a combat\_log is provided. If it is, the function renders the combat layout, including ASCII art for the player and enemy, and the last few messages from the log. If combat\_log is not provided, it renders the exploration layout, using Python's textwrap module to format the description and a typing animation for an immersive effect. The function returns the last line number it used for drawing to help position subsequent UI elements.
* **Pseudocode:**

PROCEDURE render\_main\_window(scene, combat\_log)  
 IF combat\_log is not None THEN // Combat State  
 DISPLAY player ASCII art on the left  
 DISPLAY enemy ASCII art on the right  
 DISPLAY "Combat:" header  
 FOR each of the last 2 logs in combat\_log  
 STYLE log based on content (e.g., red for attacks)  
 DISPLAY styled log message  
 END FOR  
 line\_num = last line number used  
 ELSE // Exploration State  
 DISPLAY "Description:" header  
 wrapped\_lines = WRAP scene.description to fit screen width  
 FOR each line in wrapped\_lines  
 MOVE CURSOR to the correct position  
 DISPLAY line with typing animation  
 END FOR  
 line\_num = last line number used  
 END IF  
 RETURN line\_num  
END PROCEDURE

**State-Driven Choice Rendering (render\_options)**

* **Analysis:** The choices available to the player change based on the game's state. In exploration, options may have item prerequisites. In combat, the options are always a fixed set of actions (Attack, Defend). The UI must reflect these different states and clearly indicate which options are currently selectable.
* **Specification:** The function will first check if an enemy object is active. If so, it will ignore the scene's JSON data and display a hard-coded list of combat actions. If not in combat, it will iterate through the scene's options, check the player's inventory against any prerequisites, and style the text differently (e.g., bright vs. dim) to show which options are available.
* **Pseudocode:**

PROCEDURE render\_options(start\_line, scene, player, enemy)  
 DISPLAY "Options:" header below start\_line  
  
 IF enemy is active AND enemy is alive THEN  
 // Combat State  
 DISPLAY hard-coded "1. Attack" and "2. Defend" options  
 RETURN list of combat options  
 ELSE  
 // Exploration State  
 valid\_options\_flags = []  
 FOR each option in scene.options  
 is\_valid = CHECK player.inventory against option.prerequisites  
  
 IF is\_valid THEN  
 STYLE option text as selectable (e.g., bold)  
 ADD True to valid\_options\_flags  
 ELSE  
 STYLE option text as unselectable (e.g., dim)  
 ADD requirement text (e.g., " (Requires: data\_key)")  
 ADD False to valid\_options\_flags  
 END IF  
 DISPLAY styled option text  
 END FOR  
 RETURN valid\_options\_flags  
 END IF  
END PROCEDURE

**Robust Input Handling (get\_player\_action)**

* **Analysis:** The game needs a robust way to get player input that handles both numbered choices and special string commands (like "save", "help"). It must validate input and provide clear error messages without crashing.
* **Specification:** The function will run an infinite loop. Inside, it will clear previous input/error lines and prompt the user. It will check if the input matches a special command. If not, it will try to convert the input to an integer and check if it's within the valid range of options. It provides feedback for invalid input and only breaks the loop when valid input is received.
* **Pseudocode:**

PROCEDURE get\_player\_action(line\_num, num\_options)  
 SPECIAL\_COMMANDS = ["save", "load", "quit", "inventory"]  
 WHILE True DO  
 CLEAR previous error and input lines  
 MOVE CURSOR to input position  
 user\_input = GET input from user  
  
 IF user\_input is in SPECIAL\_COMMANDS THEN  
 RETURN user\_input  
 END IF  
  
 IF user\_input is "help" THEN  
 DISPLAY list of special commands  
 PAUSE for user to read  
 CONTINUE loop  
 END IF  
  
 TRY  
 choice = CONVERT user\_input to integer  
 IF choice is between 1 and num\_options THEN  
 RETURN choice  
 ELSE  
 DISPLAY "number out of range" error and PAUSE  
 END IF  
 CATCH ValueError  
 DISPLAY "invalid command" error and PAUSE  
 END TRY  
 END WHILE  
END PROCEDURE

**Turn-Based Combat Logic (handle\_combat)**

* **Analysis:** A single round of turn-based combat needs to be processed. This includes the player's action, checking for enemy defeat, the enemy's counter-action, and checking for player defeat.
* **Specification:** The function takes the player, enemy, the chosen action, and the combat log. It resolves the player's action (attack or defend). If the enemy survives, the enemy AI performs its action. The function appends descriptive messages of all events to the combat log and returns a status code indicating victory, defeat, or continuation.
* **Pseudocode:**

PROCEDURE handle\_combat(player, enemy, player\_action, combat\_log)  
 // Player's Turn  
 IF player\_action is "attack" THEN  
 player ATTACKS enemy  
 ADD "You attack..." to combat\_log  
 ELSE IF player\_action is "defend" THEN  
 player DEFENDS  
 ADD "You defend..." to combat\_log  
 END IF  
  
 IF enemy is NOT alive THEN  
 ADD "You defeated..." to combat\_log  
 RETURN 1 // Victory code  
 END IF  
  
 // Enemy's Turn  
 enemy PERFORMS its action (attack or defend)  
 IF enemy is defending THEN  
 ADD "Enemy defends..." to combat\_log  
 ELSE  
 ADD "Enemy attacks..." to combat\_log  
 END IF  
  
 IF player is NOT alive THEN  
 ADD "You have been defeated..." to combat\_log  
 RETURN 0 // Defeat code  
 END IF  
END PROCEDURE

# Additional Features

## Missing Functionality

Any required functionality that is missing from the program

All basic and advanced requirements outlined in the project description have been successfully implemented, except the game map. This includes:

* A complete, branching storyline with over ten meaningful decision points.
* Item collection and prerequisite system.
* A turn-based combat system.
* Save and load functionality.
* A viewable ASCII map was designed but not implemented in the final version to focus on a more dynamic UI.

Aside from this, the game could be expanded by adding a difficulty setting (Easy, Normal, Hard). This would adjust the game's balance by changing the player's and enemies' combat stats (damage/defense percentages) and altering the chance of random encounters. This feature would provide replayability and cater to a wider range of players.

## Extra Features

Any extra functionality that may be worth bonus marks in the program

Several features were implemented beyond the core requirements, which may be worth bonus marks:

* **Advanced Terminal UI:** The game does not simply print text sequentially. It uses a custom term package with ANSI escape codes to create a persistent, "game-like" window with distinct, responsive sections for status, description, and choices.
* **ASCII Art and Styling:** ASCII art is used to enhance combat immersion. Text is styled with colors and effects (bold, dim) to improve readability and highlight important information.
* **Typing Animation Effect:** Scene descriptions and other text are printed with a "typewriter" effect to create a more engaging and paced narrative experience.
* **Random Combat Encounters:** One of the game's locations features a random encounter system, adding an element of unpredictability and replayability.