Technical Documentation

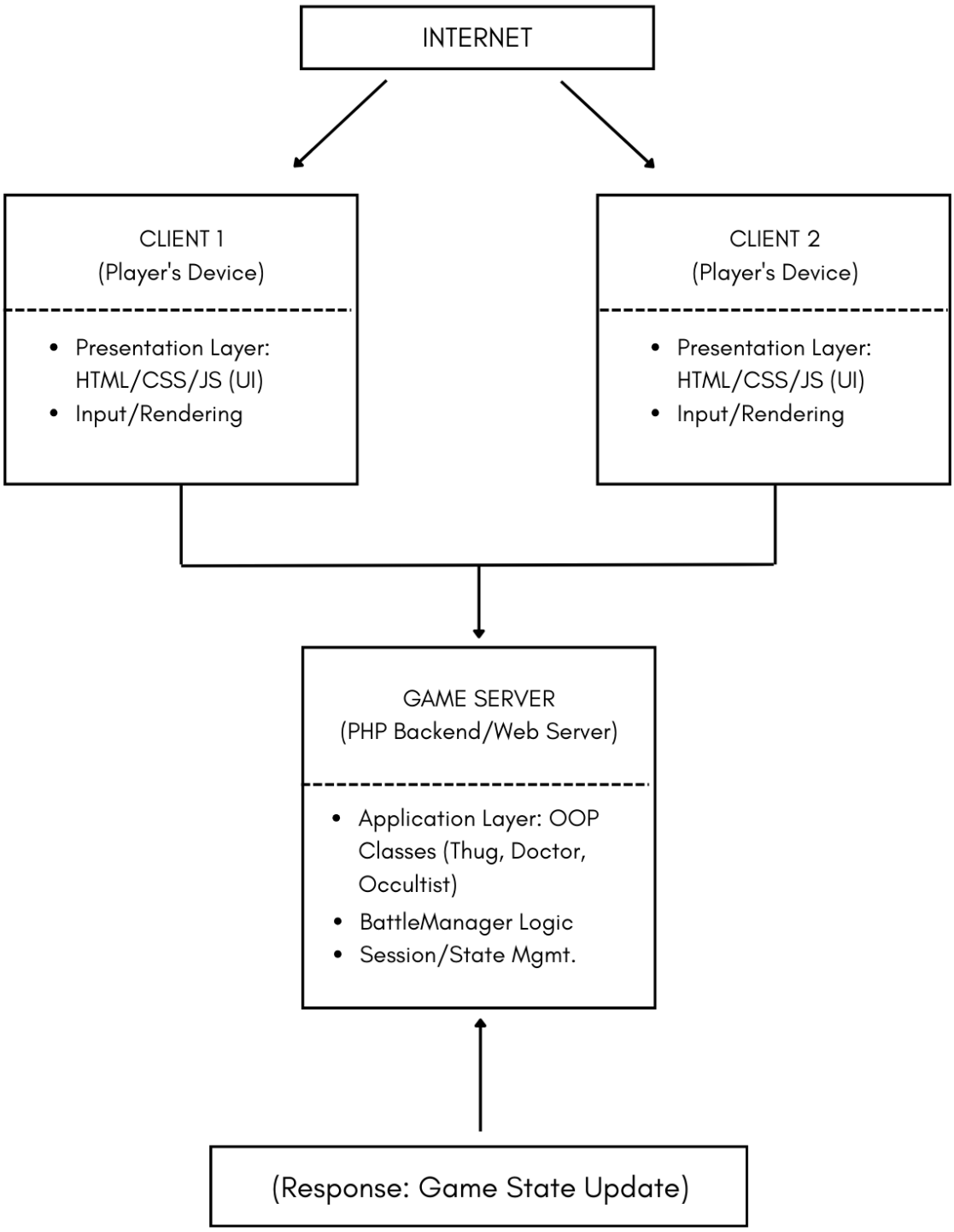
OOP Implementation Guide

This guide details the effective application of Object-Oriented Programming (OOP) principles within the PHP backend to ensure a structured, flexible, and maintainable codebase.

1. **Encapsulation**
    - **Principle:** Game state integrity is maintained by encapsulating critical data within class boundaries.
    - **Implementation:** Core attributes (e.g., hp, stats, turns) are contained within PHP classes such as Player, Skill, and BattleManager. Access and modification are strictly controlled via **public getters and setters**.
    - **Rationale:** This prevents direct, unauthorized manipulation of game variables, enforcing a predictable flow of data and enhancing code security and organization.
2. **Inheritance**
    - **Principle:** Promotes code reusability through a defined class hierarchy.
    - **Implementation:** A base Character class is defined to store shared properties (name, hp) and common methods (attack()). Specialized derived classes (**Thug**, **Doctor**, **Occultist**) inherit these properties.
    - **Rationale:** Each derived class overrides or extends specific methods, such as useSkill(), to implement its unique mechanical identity. This simplifies future feature expansion.
3. **Polymorphism**
    - **Principle:** Enables different classes to respond distinctly to the same method call, achieving flexible, class-based behavior.
    - **Implementation:** Combat functions (e.g., executeAction()) are called identically across all character classes. However, the resulting effect is class-specific:
        - An Occultist's executeAction() triggers complex spell logic and elemental damage.
        - A **Thug**'s executeAction() triggers physical strike calculations.
4. **Abstraction**
    - **Principle:** Hides complex implementation details behind simple interfaces.
    - **Implementation:** Core systems (e.g., skill processing, combat flow) are defined using abstract classes (AbstractSkill) or interfaces (ICombatAction).
    - **Rationale:** This design enforces modularity and scalability. New skills, effects, or future character classes can be integrated simply by implementing the defined abstract contracts, without requiring modifications to the core battle engine.

Architecture Diagram

The diagram illustrates a client–server architecture for an online turn-based game. Two players (Client 1 and Client 2) interact with the game through their devices using HTML, CSS, and JavaScript for the presentation and input layers. Their actions are sent through the internet to the Game Server, which handles all core game logic. The server processes requests using PHP, object-oriented classes (such as Thug, Doctor, and Occultist), the BattleManager, and session/state management. After computing the results, the server sends back a Game State Update to both clients, ensuring synchronized and authoritative gameplay**.**



Design Decisions and Rationale

Architecture Rationale

The project employs a standard Client-Server Architecture.

- **Decision:** All critical game logic resides on the PHP backend. The client (browser) is solely responsible for rendering the UI and relaying player input.
- **Rationale:** This fundamental separation prevents client-side cheating and ensures the server maintains the single, authoritative source for the game state (e.g., HP, damage calculations), guaranteeing fairness and security.

Gameplay Structure

- **Decision:** Implementation of a strict turn-based combat system with predefined roles (**Thug, Doctor, Occultist**).
- **Rationale:** The turn-based nature simplifies the challenge of synchronizing player actions over a web connection and allows development effort to prioritize deep, strategic combat mechanics. The predefined roles serve as immediate proof points for the core OOP principles (Inheritance and Polymorphism).