

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.0

Kyler Legrone

## Table of Contents

[**CS 230 Project Software Design Template** 1](#_Toc115077317)

[**Table of Contents 2**](#_Toc115077318)

[**Document Revision History 2**](#_Toc115077319)

[**Executive Summary 3**](#_Toc115077320)

[**Requirements 3**](#_Toc115077321)

[**Design Constraints 3**](#_Toc115077322)

[**System Architecture View 3**](#_Toc115077323)

[**Domain Model 3**](#_Toc115077324)

[**Evaluation 4**](#_Toc115077325)

[**Recommendations 5**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 03/23/2025 | Kyler Legrone | Initial draft of the software design document. |

**Executive Summary:**The purpose of this document is to outline the design and architecture of the web-based game "Draw It or Lose It". The Gaming Room has requested a solution for their game that works across multiple platforms. This document addresses the key requirements, including the use of Singleton and Iterator design patterns, ensuring unique game, team, and player names, and providing a system architecture to ensure that only one instance of each game, team, and player can exist at any given time. The design focuses on scalability, maintainability, and user experience.  
  
**Requirements:**

1. The system should support one or more teams per game.
2. Each team must have one or more players, and players must be uniquely identified.
3. The game and team names must be unique.
4. Only one instance of a game, team, or player should exist at any time.
5. The system must provide a method to add games, teams, and players, with appropriate validation to ensure uniqueness of names.
6. The system should be able to list all games, teams, and players, using an Iterator pattern to iterate over collections.”

**Design Constraints:**

1. The game must be designed for a web-based environment, which imposes certain performance and user interface constraints.
2. The solution must use Java and follow object-oriented principles, as specified in the project guidelines.
3. The game should be able to scale for multiple users simultaneously, although there will be no backend database during the prototype phase.
4. Since the solution is in Java, memory management and efficient object handling are critical, especially when managing multiple game instances.”

**System Architecture View:**  
The architecture of the system follows a layered approach, consisting of:

* **GameService**: This class manages the creation and management of games, teams, and players. It ensures that only one instance of each entity exists in memory at any given time (Singleton pattern).
* **Entity Class**: This base class holds common properties such as id and name for Game, Team, and Player.
* **Game Class**: Responsible for managing teams within the game. It uses a list to store teams and ensures teams have unique names.
* **Team Class**: Represents a team, containing players. Players are added with unique IDs and checked for name uniqueness.
* **Player Class**: Represents individual players on a team, each with a unique ID and name.”

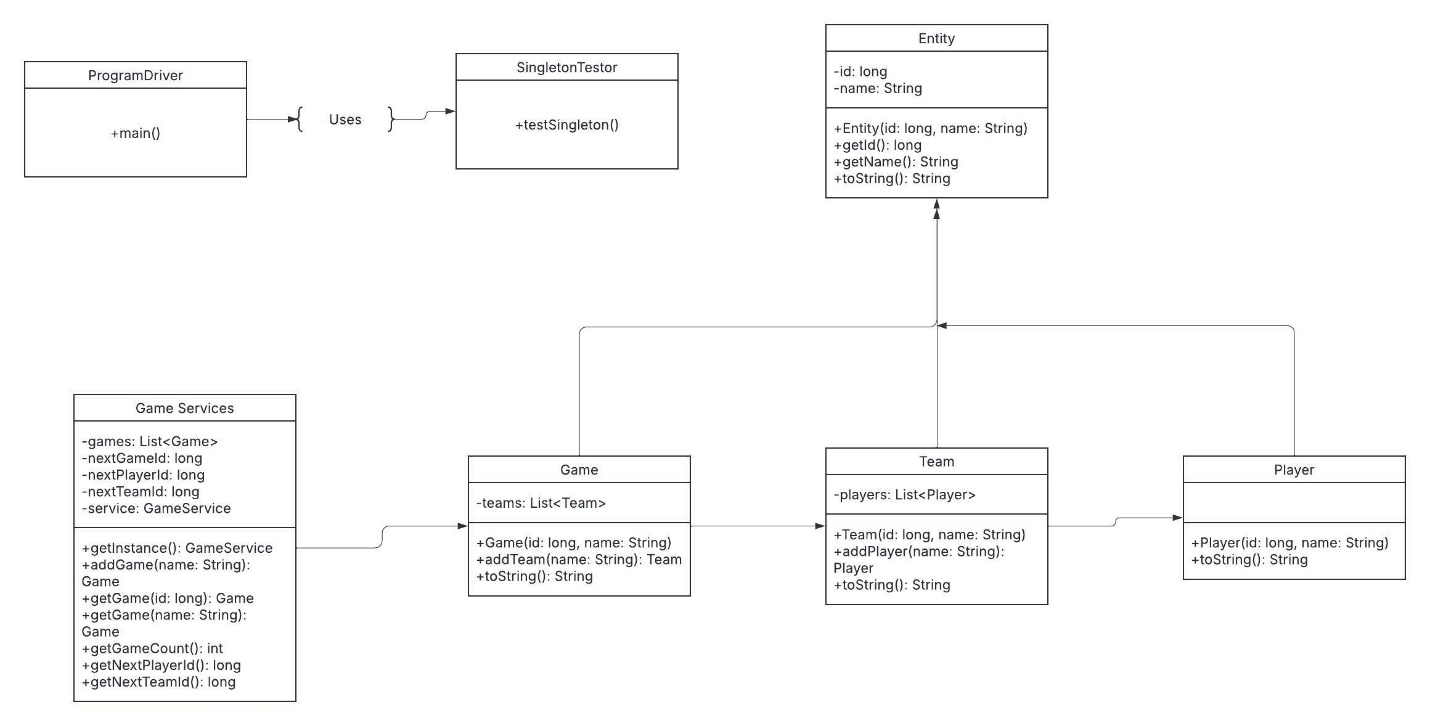
## [Domain Model](#_8h2ehzxfam4o)

**Summary of Relationships:**

1. **ProgramDriver <<uses>> SingletonTester**:  
   The ProgramDriver class uses the SingletonTester class, indicating that it calls methods or interacts with it during execution (i.e., testSingleton()).
2. ***GameService 0... → Game*\*:**  
   The GameService class manages multiple instances of the Game class. This relationship indicates that a GameService can contain many Game objects.
3. ***Game 0... → Team*\*:**  
   A Game can have multiple Team objects, representing that each game can have many teams.
4. ***Team 0... → Player*\*:**  
   A Team can have multiple Player objects, meaning each team is composed of many players.
5. **Game, Team, and Player ← Entity**:  
   These classes inherit from the Entity class, meaning they share common attributes such as id and name, and common methods like getId(), getName(), and toString().

**Key Points to Understand the Diagram:**

* Inheritance: The Game, Team, and Player classes inherit from the Entity class.
* Composition/Association: The GameService manages a collection of Game objects, which in turn manage collections of Team objects, and each Team manages collections of Player objects.
* Singleton Pattern: The GameService is a singleton, meaning there is only one instance of it throughout the program, accessible via getInstance().

****

## [Evaluation](#_2o15spng8stw)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Limited server hosting capability. Not commonly used for large-scale production web servers. Hardware is costly and server management options are limited. | Excellent choice for web-based server hosting. Supports Apache, Nginx, and scalable cloud deployments. Free, open-source, highly secure. | Fully capable with IIS and Windows Server. Integrates well with Microsoft technologies. Licensing costs are significant. | Not applicable for server hosting. Mobile devices function only as clients, not hosts. |
| **Client Side** | Fully supports modern web browsers (Safari, Chrome, Firefox). Compatible with HTML5 responsive design. Requires testing for Safari-specific rendering behaviors. | Strong browser compatibility (Chrome, Firefox). Supports responsive HTML5 web apps. Suitable for users comfortable with Linux environments. | Full compatibility with major browsers (Edge, Chrome, Firefox). Common user base ensures broad support and stable rendering. | Requires responsive design for various screen sizes. Android uses Chrome WebView; iOS uses Safari/WebKit. Must test thoroughly for browser quirks and device-specific UI issues. |
| **Development Tools** | Xcode is required for iOS development. Web development supported through VS Code, WebStorm, etc. Apple hardware is mandatory for iOS builds. | Wide range of free/open-source tools (VS Code, Eclipse, IntelliJ). Strong support for back-end dev (Node.js, Python, Java). | Supports major IDEs (Visual Studio, VS Code, etc.). Great for .NET or Windows-specific apps. Paid licenses may apply for some enterprise tools. | Android Studio (Java/Kotlin) and Xcode (Swift) required for native apps. Web-based app only needs responsive front-end frameworks (React, etc.). Platform-specific devs may be needed. |

This evaluation provides a comparative analysis of various platforms to determine their suitability for supporting \*Draw It or Lose It\* in a distributed, web-based environment. The analysis considers each platform’s ability to host the application on the server side, deliver a consistent and responsive client-side experience, and support efficient development workflows using relevant tools and languages. The goal is to identify the strengths, limitations, and overall feasibility of deploying the application across Linux, macOS, Windows, and mobile platforms (iOS and Android), in alignment with The Gaming Room’s expansion goals.

**Recommendations**

**Operating Platform**

**Recommended Platform: Linux (Ubuntu Server) with Docker Containerization**

Linux is ideal for server deployment due to its open-source nature, cost-effectiveness, scalability, and robust support for web technologies. Using Docker enables containerized, consistent deployment across multiple environments, including development, testing, and production. This allows Draw It or Lose It to scale and run uniformly on Linux, Windows, and macOS systems.

**Operating System Architectures**

**Linux Architecture:**

- Monolithic Kernel: Linux uses a monolithic kernel with support for modularity. It provides direct access to hardware-level management of processes, memory, and I/O operations.

- Modular Design: Components such as drivers and services can be loaded/unloaded as needed, making Linux adaptable to evolving needs.

- Multi-threaded Process Management: Essential for handling multiple concurrent players, game sessions, and network requests in Draw It or Lose It.

**Cross-Platform Mobile Architecture:**

- React Native or Flutter: Enables shared codebase development for Android and iOS.

- Native Bridge Access: These frameworks support calling native modules to access device-specific hardware and optimize game performance.

**Storage Management**

**Recommended Storage System:**

- Cloud Storage: Utilize AWS S3 or Google Cloud Storage for storing game assets, user data, and logs.

- Relational Databases: Use MySQL or PostgreSQL for storing structured game data (user profiles, scores, sessions).

- Backups and Redundancy: Set up automated backups and data replication to prevent data loss and ensure high availability.

This architecture allows The Gaming Room to scale storage needs easily as the player base grows.

**Memory Management**

**Server-Side (Linux):**

- Virtual Memory Model: Provides an abstraction layer that efficiently manages RAM and swap space.

- Swapping: Ensures that inactive memory pages are swapped to disk, allowing for sustained performance under load.

- Caching: Frequently accessed files are cached to accelerate access times and reduce I/O overhead.

**Client-Side (Mobile):**

- Automatic Garbage Collection: Java (Android) and Swift (iOS) handle memory management internally, reducing risk of memory leaks.

- Efficient Resource Use: Optimize game assets and component loading to minimize memory consumption on mobile devices.

**Distributed Systems and Networks**

**Communication Architecture:**

- RESTful APIs: Facilitate standardized communication between client apps and server components.

- WebSockets: Provide real-time, bidirectional communication, critical for live game actions like drawing and guessing.

**System Design Considerations:**

- Service-Oriented Architecture (SOA): Each major function (game logic, user management, chat, etc.) can be handled as an independent service.

- Load Balancing & Fault Tolerance: Use tools like NGINX or AWS ELB to balance traffic and replicate services for high availability.

- Dependency Management: Plan for service isolation and use of message queues (e.g., RabbitMQ) for resilience during outages or delays.

**Security**

**Security Strategies:**

- Encryption in Transit: Use HTTPS with SSL/TLS for secure data transfer.

- Authentication: Implement OAuth 2.0 with JWT tokens to allow secure, stateless authentication across platforms.

- Data Encryption at Rest: Store sensitive data using AES-256 encryption.

- Access Controls: Implement role-based access for administrative functions to restrict unauthorized data access.

- Mobile Security:

- iOS: Enforce App Transport Security (ATS).

- Android: Use ProGuard and Network Security Configuration to prevent tampering and enhance runtime safety.

These strategies ensure data integrity, protect user privacy, and align with industry-standard security practices.