

Draw It or Lose It Web Application

# **CS 230 Project Software Design Template**

Version 1.0

## 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 4**](#_Toc115077322)

[**System Architecture View 4**](#_Toc115077323)

[**Domain Model 5**](#_Toc115077324)

[**Evaluation 6**](#_Toc115077325)

[**Recommendations 7**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 04/24/2025 | James Brewer Jr | Initial Project One submission. Added Executive Summary, Requirements, Design Constraints, and Domain Model. |
| 1.1 | 04/25/2025 | James Brewer Jr | Added Evaluation section for Project Two, including server-side and client-side platform analysis. |
| 1.2 | 04/26/2025 | James Brewer Jr | Added Recommendations section for Project Three (operating platform, OS architecture, storage,  memory, distributed systems, and security) |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room is transitioning their successful mobile game, Draw It or Lose It, into a scalable, web-based application that supports multiple operating platforms. This project presents a solution for developing a backend structure that allows the game to be hosted on a server and accessed by desktop and mobile clients. The proposed solution leverages object-oriented programming (OOP) principles, design patterns like Singleton and Iterator, and a modular class hierarchy to ensure that only one instance of a game runs at any time, teams and players can be efficiently managed, and game-related data is handled in a reliable, scalable manner. This foundational software design allows for future enhancements, including graphical interfaces, persistent storage, and multiplayer networking.

## Requirements

The client, The Gaming Room, has requested a scalable, cross-platform game application based on their current Android title, Draw It or Lose It. The application must support players across Linux, Mac, Windows, and mobile devices by delivering a web-based experience. The game should include the following functional and technical requirements:

Functional Requirements:

• Allow users to create and manage games with unique names

• Support nested structures including teams and players within games

• Prevent duplicate names for games, teams, and players

• Provide a centralized GameService to control all game-related logic

Technical Requirements:

• Use the Java programming language

• Follow object-oriented programming principles (encapsulation, inheritance, abstraction)

• Implement the Singleton design pattern for the GameService class

• Enforce unique entity constraints using iteration and comparison logic

• Ensure compatibility across platforms (no OS-specific logic)

• Allow for future integration with front-end interfaces and persistent storage

These requirements form the foundation for the game architecture and ensure that the project can evolve into a production-ready, multi-platform gaming environment.

**Design Constraints**

This game application must function in a web-based distributed environment, which introduces several key constraints:

1. Singleton Limitation: Only one instance of the GameService class should exist in memory at any time, requiring use of the Singleton pattern to prevent accidental duplication.

2. Name Uniqueness: Game and team names must be unique to prevent duplication and conflicts in a multi-user environment. This constraint is enforced using the Iterator pattern to search for duplicates before allowing creation.

3. Scalability: The application must scale to support thousands of concurrent users, which means no hard-coded limits or in-memory assumptions that don't scale well in production.

4. Platform Compatibility: The solution must be portable across Linux, Windows, and Mac OS server environments, which limits OS-specific functionality or dependencies.

5, Volatile Memory: The current implementation does not persist data between sessions. All instances are lost if the application is stopped, limiting long-term usage until a database layer is added in the future.

## [System Architecture View](#_ilbxbyevv6b6)

Please note: There is nothing required here for these projects, but this section serves as a reminder that describing the system and subsystem architecture present in the application, including physical components or tiers, may be required for other projects. A logical topology of the communication and storage aspects is also necessary to understand the overall architecture and should be provided.

## [Domain Model](#_8h2ehzxfam4o)

Domain Model

The UML class diagram for the game application demonstrates the use of inheritance, encapsulation, and modularity. At the core is an abstract Entity class, which contains shared attributes (id and name) and is extended by Game, Team, and Player. This use of inheritance reduces code duplication and allows polymorphic behavior. The GameService class acts as a Singleton that controls the lifecycle of all games, teams, and players.

Object-oriented principles applied:

• Encapsulation: All class fields are private with public accessors (getters), protecting data from unauthorized access.

• Inheritance: The Entity class is extended by multiple classes, promoting code reuse.

• Abstraction: The Entity class is abstract and never instantiated directly, encouraging proper OOP hierarchy usage.

• Iterator Pattern: Used to traverse collections to enforce unique names and search for existing entities.

• Singleton Pattern: Ensures that only one GameService instance manages the application’s game data in memory.

This model supports flexible and maintainable code, forming a solid base for future expansion of features and platform integration.



## [Evaluation](#_2o15spng8stw)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | macOS can support basic web server hosting (e.g., Apache), but it’s not typically used in production. Not ideal for scalable deployment. Limited support for enterprise-grade deployment tools. Licensing costs are tied to Apple hardware. | Linux is the most widely used server OS for scalable web-based apps. It supports Nginx, Apache, Docker, and Kubernetes. Excellent performance and no licensing fees. Ideal for hosting. | Windows Server can host web applications via IIS or Docker. Reliable but less performant than Linux for large-scale hosting. Licensing costs apply for Windows Server OS. | Mobile platforms are not used for server hosting. Backend services must be hosted elsewhere (e.g., Linux/Windows cloud servers). These devices act only as clients. |
| **Client Side** | macOS fully supports modern browsers and can render responsive web apps. Requires testing for UI consistency with Linux/Windows. Development tools are mac-compatible. | Linux supports Chromium-based browsers and web rendering, but testing for UI/UX consistency is needed. Popular for developers, but not a common end-user platform. | Most desktop users run Windows, making it essential for testing compatibility. Supports all modern web browsers. Development tools like Visual Studio work well here. | Apps must be mobile responsive. Requires testing on different screen sizes, touch support, and OS-specific behavior. Responsive HTML5, CSS, and JavaScript frameworks are key. |
| **Development Tools** | Supports Xcode, IntelliJ, Eclipse, and JetBrains IDEs. Java, Python, JavaScript widely supported. May need separate dev environment if targeting other OSs. No license fees. | Open-source tools are abundant: Eclipse, VS Code, IntelliJ, etc. Full support for Java and web technologies. Ideal for backend development. No licensing costs. | Visual Studio, Eclipse, IntelliJ, and Android Studio run on Windows. Java, C#, and JavaScript dev supported. Licensing may apply for Visual Studio Pro or Enterprise. | Android Studio (for Android) and Xcode (for iOS) are required for native builds. Web app compatibility also requires extensive mobile browser testing. iOS requires a Mac for publishing. |

Recommendations

Analyze the characteristics of and techniques specific to various systems architectures and make a recommendation to The Gaming Room. Specifically, address the following:

1. **Operating Platform**: For The Gaming Room’s expansion of Draw It or Lose It, I recommend adopting a Linux-based server platform, such as Ubuntu Server or Red Hat Enterprise Linux. Linux offers robust scalability, strong community and enterprise support, cost-effectiveness (most distributions are free), and compatibility with containerized deployments like Docker and orchestration platforms like Kubernetes. This makes Linux the ideal foundation for developing, deploying, and scaling web-based applications across distributed environments.
2. **Operating Systems Architectures**: The Linux operating system follows a modular, monolithic architecture. Its kernel provides high performance and reliability, while modular extensions allow for specific customization depending on system needs. The architecture supports multiple users and processes simultaneously, enabling efficient task management in multi-threaded environments. Furthermore, the POSIX-compliant architecture ensures high compatibility with development tools, containerization, and remote access, making it ideal for cloud-native applications.
3. **Storage Management**: For storage, a robust, scalable file system like ext4 or XFS can be used in conjunction with Logical Volume Management (LVM). These file systems provide journaling, fast recovery, and efficient space allocation, which are critical for supporting multiple users and sessions. LVM adds flexibility by allowing dynamic resizing of storage volumes without disrupting the application. For enterprise-level growth, cloud-integrated solutions like AWS Elastic File System (EFS) or network-attached storage (NAS) options can be layered in.
4. **Memory Management**: Linux uses advanced memory management techniques such as paging, virtual memory, and demand paging. The system automatically caches disk operations in memory to improve read/write performance. It also swaps less-used memory segments to disk when RAM is constrained. These strategies ensure that Draw It or Lose It remains responsive under high loads. Additionally, Java’s built-in garbage collection (used in this application) complements OS-level memory management to prevent memory leaks or overflow.
5. **Distributed Systems and Networks**: In a distributed architecture, Draw It or Lose It will consist of independent services (microservices or RESTful APIs) communicating over a network. Tools such as Docker and Kubernetes will help package and orchestrate services across multiple servers or cloud zones. Load balancers can manage traffic efficiently, and failover strategies ensure system availability even during outages. Secure APIs and asynchronous messaging patterns (e.g., message queues) help maintain communication between different platforms while ensuring data consistency.
6. **Security**: To protect user information, Linux supports robust security features, including user and group permissions, SELinux (Security-Enhanced Linux), and built-in firewall tools (like iptables or ufw). Data in transit can be protected using HTTPS/TLS encryption, and sensitive information can be stored using database encryption and access control layers. Additionally, deploying regular patch updates, intrusion detection systems (IDS), and secure authentication (OAuth2 or multi-factor authentication) can prevent unauthorized access and data breaches.