

**Draw It or Lose It**  
CS 230 Project Software Design  
Version 2.0

## 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 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 |
| --- | --- | --- | --- |
| 2.0 | 08/13/2025 | Kyle Dauk | Final submission of software design document for CS 230 Project 3 |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room has requested the development of a web-based version of its existing Android game, Draw It or Lose It, to make it accessible across multiple platforms. The web application must accommodate multiple teams per game, with each team consisting of multiple players. Game and team names must be unique to avoid conflicts, and only one instance of the game management system should exist in memory at any time. To meet these technical and business requirements, a Java-based solution is proposed, leveraging key object-oriented programming principles. A Singleton pattern is used to enforce the single-instance requirement, while the Iterator pattern ensures uniqueness of names by allowing the application to search through existing entities before creating new ones. This design lays the groundwork for a scalable, maintainable, and secure multi-platform game application.

## Requirements

The application must allow one or more teams to participate in a game, with each team having multiple players. Every team and game must have a unique name, and the application must allow users to check whether a name is already in use before selecting it. A Singleton pattern is required to ensure that only one instance of the GameService class exists in memory, managing all entities within the game. Additionally, the application must incorporate an Entity base class from which the Game, Team, and Player classes will inherit common attributes, such as ID and name. The Iterator pattern must be used in all entity creation processes to enforce name uniqueness. These requirements support the client’s business goal of offering a robust, scalable multiplayer experience across web and mobile platforms.

## [Design Constraints](#_2et92p0)

Developing a web-based, distributed version of Draw It or Lose It introduces several design constraints. First, the application must be platform-independent to support different operating systems and browsers, which is addressed by implementing the core logic in Java. Second, it must operate in a stateless, distributed architecture, making it essential to maintain data consistency and enforce concurrency controls — especially in the Singleton Game Service implementation. Thread safety becomes critical in this scenario to avoid the creation of multiple instances under concurrent access. In addition, the need for name uniqueness among teams and games requires robust iteration and validation mechanisms, which introduces performance considerations when scaling to many users. Finally, the web-based environment also implies that security, latency, and eventual persistence of data must be planned for even if they are not implemented in this initial prototype.

## [Domain Model](#_8h2ehzxfam4o)

A screenshot of a computer

AI-generated content may be incorrect.

## [Evaluation](#_2o15spng8stw)

Using your experience to evaluate the characteristics, advantages, and weaknesses of each operating platform (Linux, Mac, and Windows) as well as mobile devices, consider the requirements outlined below and articulate your findings for each. As you complete the table, keep in mind your client’s requirements and look at the situation holistically, as it all has to work together.

In each cell, remove the bracketed prompt and write your own paragraph response covering the indicated information.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Mac provides a UNIX-based system with excellent stability and security features. However, it is rarely used as a hosting environment due to high hardware costs and limited adoption in enterprise-level hosting. | Linux is the most widely used platform for web hosting because it is secure, stable, scalable, and open-source, making it highly cost-effective and ideal for server deployment. | Windows servers integrate well with Microsoft technologies and enterprise tools but are expensive to license and require frequent maintenance compared to Linux. | Mobile devices cannot serve as dedicated servers because of hardware limitations and operating system restrictions. They are limited to client functionality only. |
| **Client Side** | Applications designed for Mac must maintain consistent UI/UX quality. Development for macOS may require extra time for compatibility testing and optimization. | While strong on servers, Linux desktops are less common, so supporting them requires additional testing and handling of dependencies. | Windows remains the most widely used desktop OS, making it essential for client-side compatibility to reach the largest audience. | Mobile platforms require responsive design and optimization for both Android and iOS. This adds complexity and development time but is critical for user accessibility. |
| **Development Tools** | Fully supports Java, Eclipse IDE, and JDK, enabling efficient cross-platform development and testing. | Linux supports Java, Eclipse, and other open-source tools, offering excellent integration for both development and hosting environments. | Windows supports Java and Eclipse IDE and integrates well with enterprise tools like Visual Studio, making it a versatile development platform. | Development for mobile requires Android SDK for Android and Xcode for iOS, with React Native or similar frameworks for cross-platform capabilities. |

## ****Recommendations:** Operating Platform**

For Draw It or Lose It, Linux is the optimal operating (server) platform. It offers enterprise-grade stability, proven scalability, and robust security while being cost-effective due to its open-source nature. Its broad adoption in cloud computing environments (e.g., AWS, Azure, Google Cloud) ensures seamless deployment, horizontal scaling, and integration with modern DevOps pipelines. Linux also supports container orchestration platforms such as Kubernetes and Docker, enabling continuous integration/continuous deployment (CI/CD) workflows. These capabilities align with the client’s need for a highly available, responsive, and maintainable game service.

### ****Operating Systems Architectures****

The recommended architecture is a **three-tier, stateless, distributed model**:

1. **Presentation Layer** – Web and mobile clients deliver the user interface, leveraging responsive design for consistent cross-platform experiences.
2. **Application Layer** – Core game logic, matchmaking, and game state management run on Java-based microservices. A Singleton GameService enforces the one-instance rule while being horizontally scalable across multiple nodes.
3. **Data Layer** – A relational database cluster (e.g., PostgreSQL) handles persistent storage with replication and automatic failover.

Using a stateless design ensures that user sessions and game states are maintained externally (in the database or an in-memory store like Redis). This allows rapid scaling and resilience against node failures. The architecture supports load balancers, API gateways, and service discovery, making it robust against traffic spikes and outages.

### ****Storage Management:****

A **PostgreSQL** relational database is recommended for its ACID compliance, advanced indexing, and JSON support for semi-structured data (e.g., player statistics). Features such as foreign key constraints and transaction isolation levels ensure consistency for critical game data (games, teams, players, match history).

* **Data Backup** – Daily incremental backups with weekly full backups stored securely in the cloud.
* **Growth Planning** – Implement partitioning/sharding if the player base scales significantly.
* **Caching** – Use Redis for frequently accessed data (leaderboards, active sessions) to reduce query load and latency.  
  This approach balances reliability with performance and prepares for long-term scalability.

### ****Memory Management:****

The Java Virtual Machine (JVM) handles memory through generational garbage collection (G1 or Parallel GC), ensuring unused objects are efficiently reclaimed without impacting performance. Proper heap sizing and tuning will be monitored via JMX (Java Management Extensions).

* **Object Lifetime** – Short-lived objects (e.g., request payloads) remain in the young generation for fast GC cycles, while persistent game objects live in the old generation until explicitly released.
* **Memory Leak Prevention** – Regular profiling with tools like VisualVM and automated testing for leaks will be part of CI/CD.
* **Caching Strategy** – A time-to-live (TTL) policy in Redis prevents stale or unnecessary data from consuming memory.  
  These practices prevent fragmentation, reduce GC pauses, and maintain high performance during peak loads.

### ****Distributed Systems and Networks:****

Cross-platform communication will occur via **RESTful APIs over HTTPS** for standard game operations and **WebSockets** for real-time updates such as turn notifications and chat.

* **Network Topology** – Cloud load balancers distribute incoming traffic evenly, and services are deployed across multiple availability zones for resilience.
* **Fault Tolerance** – Health checks, automatic failover, and retry/backoff mechanisms ensure continuity during service interruptions.
* **Content Delivery** – Static assets (images, scripts) are served via a Content Delivery Network (CDN) to reduce latency globally.  
  This setup ensures low latency, high availability, and seamless gameplay regardless of user device or geographic location.

### ****Security:****

Security will be integrated at every layer:

* **Data in Transit** – Enforced TLS 1.3 encryption for all network communication.
* **Authentication** – OAuth 2.0/OpenID Connect (via Auth0 or Keycloak) for secure, token-based login.
* **Authorization** – Role-Based Access Control (RBAC) to distinguish between user and admin privileges.
* **Data at Rest** – AES-256 encryption for all sensitive fields in the database.
* **API Protection** – Rate limiting and input validation to prevent injection attacks and abuse.
* **Monitoring** – Continuous logging and intrusion detection via tools like Wazuh or AWS GuardDuty.  
  Regular penetration testing and security audits will maintain compliance with industry best practices.