

<Name-of-Software-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 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 | 09/30/2025 | James Marcano | Initial submission of completed software design document for The Gaming Room project |

**Instructions**

Fill in all bracketed information on page one (the cover page), in the Document Revision History table, and below each header. Under each header, remove the bracketed prompt and write your own paragraph response covering the indicated information.

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room currently offers its popular Android app, Draw It or Lose It, but wants to expand into a web-based, multi-platform version that can support more players and devices. This project outlines a Java-based solution that implements core software design patterns to meet their goals. Using the Singleton pattern, the GameService class ensures that only one instance of the service exists in memory, maintaining consistent control over game data. The Iterator pattern helps manage collections of games, teams, and players efficiently, enforcing unique names and IDs. This design allows The Gaming Room to scale Draw It or Lose It for web environments while keeping performance, security, and user experience in mind.

## Requirements

*Business*:

(1) Deliver Draw It or Lose It as a web app accessible across platforms.

(2) Support one or more teams per game.

(3) Each team has multiple players.

(4) Allow users to check whether a game/team name is available

(5) Reflect the intended gameplay flow

*Functional*:

(1) Exactly one in-memory coordinator for IDs and uniqueness

(2) Unique names for games and teams (this prototype also enforces unique player names)

(3) Four rounds per game, each one minute; drawings complete at 30 seconds; if no correct guess before time expires, the remaining teams each get one guess within a 15‑second window

(4) Provide programmatic methods to add/find games, teams, and players

(5) Console driver to verify behavior

*Non‑Functional:*

(1) Cross-platform compatibility

(2) Input validation and basic security posture

(3) Readable, maintainable OOP code

(4) Foundation for horizontal scaling and persistence in future iterations

## [Design Constraints](#_2et92p0)

1) Web deployment: browser access across OSes; use platform-neutral tech.

2) Concurrency & scalability: many users simultaneously; central coordination in prototype, database and load balancing later.

3) Statelessness: minimize server session footprint; plan to externalize state.

4) Performance & memory: efficient threading and memory use for timed rounds and frequent updates. 5) Uniqueness & integrity: enforce name uniqueness centrally now; unique DB constraints later.

6) Security: input validation and HTTPS when networked; least-privilege configuration.

7) Portability: plain Java SE; avoid framework lock‑in for the prototype.

8) Testing/observability: deterministic driver output; unit tests/logging in later stages.

## [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)

Classes: Entity (base id/name), GameService (singleton coordinator), Game (owns Teams), Team (owns Players), Player (leaf). Inheritance reduces duplication (Entity → Game/Team/Player). Composition models real relationships (Game→Team, Team→Player). Encapsulation hides internal lists; access via methods/iterators. The Iterator pattern is used to traverse collections for lookups and uniqueness checks. ProgramDriver exercises the model and, per the UML, uses a small SingletonTester helper to confirm the single‑instance behavior of GameService.

**"The Gaming Room UML diagram. The top of the diagram is labeled as com dot gamingroom. Test boxes are placed in two layers. The first layer has three text boxes and the second layer has four of them. In the first layer, the 'ProgramDriver' textbox points to 'SingletonTester' textbox. The 'ProgramDriver' textbox contains the text 'asterisk main round brackets.' The 'SingletonTester' textbox contains the text 'asterisk testSingleton round brackets.' The arrow between these two text boxes are labeled 'open two angle brackets uses close two angle brackets'. In the second layer, there are 'GameService', 'Game', 'Team', and 'Player' text boxes. The 'GameService' textbox has texts arranged in two layers. The first layer contains games colon List open angle bracket Game close angle bracket, nextGamesId colon long, nextPlayer Id colon long, nextTeamId colon long, and service colon GameService. The second layer contains GameService round brackets, getinstance round brackets colon GameService, addGame open parenthesis name colon String close parenthesis colon Game, getGame open parenthesis id colon long close open parenthesis colon Game, getGame open open parenthesis name colon String close open parenthesis colon Game, getGameCount round brackets colon int, getNextPlayerID round brackets colon long, and getNextTeamId round brackets colon long. The 'GameService' box is connected with the 'Game' textbox with a line labeled 'zero dot dt dot asterisk'.  The 'Game' textbox also contains text in two layers. The first layers contains the text teams colon List open angle bracket Team close angle bracket. The second layer has Game open round bracket id colon long comma name colon String close parenthesis, addTeam open parenthesis name colon String close parenthesis Team, toString round brackets colon String. The 'Game' textbox is connected with the 'Team' textbox with a line labeled 'zero dot dt dot asterisk'. The 'Team' textbox also contains text in two layers. The first layers contains the text players colon List open angle bracket Player close angle bracket. The second layer has Team open parenthesis id colon long comma name colon String close parenthesis, addPlayer open parenthesis name colon String close parenthesis colon Player, and toString round brackets colon String. The 'Team' textbox is connected with the 'Player' textbox with a line labeled 'zero dot dt dot asterisk'. It contains the text Player open parenthesis id colon long comma name colon String close parenthesis and toString round brackets colon String. The 'Game', the 'Team, and the 'Player' boxes point to the 'Entity' textbox in first layer. The 'Entity' textbox contains text in two layers. The first layer has the text id colon long and name colon String. The second layer has Entity round brackets, Entity open parenthesis id colon long comma name colon String close parenthesis, getId round brackets colon long, getName round brackets colon String, toString round brackets colon String.**

## [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** | macOS can technically host Java-based web apps using Apache or Nginx with Tomcat, but this setup is uncommon for production use. It performs well for local testing but becomes costly and restrictive due to Apple hardware and licensing. Best suited for development environments rather than long-term deployment. | Linux is the strongest choice for server deployment. It’s open-source, secure, and optimized for performance under heavy traffic. The Gaming Room would benefit from its stability, scalability, and zero licensing costs. Distributions like Ubuntu Server or CentOS integrate seamlessly with Java and Tomcat. Linux also has vast community and documentation support. | Windows Server can host Java-based web apps using Tomcat or IIS with Java support. It integrates well with corporate systems like Active Directory, but licensing and maintenance costs are higher. While reliable and enterprise-friendly, it’s less cost-effective than Linux for scaling web applications. | Mobile devices will not act as hosts. They will connect to the main web server through HTTPS, emphasizing responsiveness, low latency, and secure communication. |
| **Client Side** | Safari and Chrome both support HTML5 applications well, though minor CSS or JavaScript inconsistencies may occur. Apple’s consistent hardware ecosystem ensures a reliable, responsive experience across devices. | Linux users typically use Firefox or Chrome, which handle modern web technologies effectively. Its open-source environment ensures full compatibility and flexibility for browser-based games. | Windows is the most widely used desktop OS, so testing should prioritize Chrome and Edge browsers. Both deliver strong performance and broad compatibility for web-based gaming. | A responsive HTML5 design ensures the app automatically adjusts to different screen sizes. Testing on both Android (Chrome) and iOS (Safari) will confirm smooth touch interaction and performance over mobile networks. |
| **Development Tools** | Developers can use IntelliJ IDEA, Eclipse, or VS Code to build and test Java code locally. macOS’s Unix-based system mirrors Linux servers, minimizing deployment issues. Homebrew simplifies installing Java, Tomcat, and supporting libraries. | Linux supports full-stack development using open-source tools like Eclipse, IntelliJ, Docker, Jenkins, and Git. It’s cost-free and ideal for CI/CD pipelines, allowing multiple developers to collaborate efficiently without licensing constraints. | Supports Java IDEs like Eclipse or IntelliJ and integrates well with Azure DevOps or GitHub for CI/CD. Licensing and configuration requirements can be heavier but it’s a strong option for enterprise setups. | Developers should focus on HTML5, CSS, and JavaScript frameworks such as React or Vue.js. Testing can be done using Chrome DevTools mobile emulator, Android Studio, and Xcode Simulator to ensure full cross-platform performance. |

## 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**: After comparing server operating systems, I recommend deploying **Draw It or Lose It** on a **Linux-based cloud environment** using a **virtualized infrastructure** such as **Amazon EC2 (Ubuntu Server)** or **AWS Fargate** for containerization. Linux is the most cost-effective and performance-optimized choice, with robust Java support, secure configuration options, and long-term scalability. Its wide adoption across cloud platforms ensures compatibility, future-proofing, and seamless integration into CI/CD pipelines.
2. **Operating Systems Architectures**: The proposed solution will use a **multi-tiered architecture** on Linux that separates the application logic, database, and static content. Java will run in a **Tomcat servlet container**, enabling efficient **thread-based concurrency** and **process isolation**. The OS kernel will manage resources via **virtual memory**, process scheduling, and **I/O multiplexing**, all critical for supporting many simultaneous game sessions.

This architecture promotes:

* **Statelessness** at the application tier.
* **Process separation** for resilience.
* **Horizontal scalability** using container orchestration (Docker/Kubernetes in future iterations).
* **High availability** through auto-healing infrastructure and failover support.

1. **Storage Management**: To meet the performance and persistence needs of the game, the following storage architecture is recommended:

**PostgreSQL or MySQL** for relational data: game IDs, usernames, scores, session data.

Enforce **unique constraints** to prevent name collisions.

Support for **ACID-compliant transactions** to maintain integrity.

**Amazon S3** for scalable object storage: storing assets like pre-rendered drawings or future expansion features (e.g., user avatars, replays).

**Redis (in-memory store)** for volatile data like round timers, match queues, and live game states—offloading short-lived values from the primary database.

This hybrid approach balances reliability, speed, and cost-efficiency.

1. **Memory Management**: Memory usage will be optimized using a **JVM-based memory model** with the following techniques:

* **Garbage Collection (GC)**: The Java Virtual Machine will automatically reclaim memory using a generation-based GC strategy, such as G1GC, tuned for real-time performance.
* **Heap Size Management**: Set minimum and maximum heap limits based on server resources to prevent memory leaks or out-of-memory crashes.
* **Object Lifetime Minimization**: Avoid unnecessary object retention (e.g., long-lived static references); use **iterators** and **lazy loading** for entity traversal and lookups.
* **Thread Pooling**: Leverage Java’s ExecutorService to manage thread reuse, reducing memory pressure and improving concurrency.

These optimizations help support dozens or hundreds of concurrent game sessions without impacting server responsiveness.

1. **Distributed Systems and Networks** A successful deployment requires **distributed software architecture** that can communicate reliably across platforms. The following is proposed:
   * + **REST APIs over HTTPS**: Facilitate communication between mobile, web, and desktop clien
     + **Stateless API Design**: Keeps game logic modular and horizontally scalable.
     + **WebSockets**: For future real-time interactions (e.g., live guessing updates or team chat).
     + **Load Balancer (e.g., AWS ELB)**: Distributes traffic across multiple app servers.
     + **Multi-AZ Deployment**: Application instances span across availability zones to ensure fault tolerance and geographic redundancy.
     + **Retry Mechanisms and Circuit Breakers**: Protect against network failure, latency spikes, and cascading outages.
     + **Monitoring and Logging (e.g., CloudWatch, ELK Stack)**: Track performance, detect anomalies, and respond to outages rapidly.

This infrastructure will ensure cross-platform stability, reliability, and room for future multiplayer enhancements.

1. **Security**: Security is mission-critical for protecting player data and ensuring platform integrity. The architecture will implement a **defense-in-depth** approach:
   * + **Data Protection**
   * Use **AES-256 encryption** at rest (e.g., in RDS and S3) and **TLS 1.2+** in transit.
   * Input sanitization to prevent injection attacks.
   * Secure coding standards (e.g., OWASP Top 10 compliance).
     + **Authentication & Authorization**
   * Use **JWT (JSON Web Tokens)** or **OAuth 2.0** for user authentication.
   * Enforce **role-based access control (RBAC)** for administrative vs. player access.
     + **Infrastructure Security**
   * Deploy behind a **Virtual Private Cloud (VPC)** with strict firewall rules.
   * Implement **DDoS protection** using AWS Shield.
   * Apply **Linux file permissions**, **SELinux policies**, and **IP whitelisting** for admin tools.
     + **Future Enhancements**
   * Add MFA (multi-factor authentication).
   * Consider full audit logging and anomaly detection using machine learning models.

This security model ensures compliance, protects PII, and maintains trust across devices and regions.

This recommendation meets The Gaming Room’s goals of platform expansion, security, performance, and long-term maintainability. By leveraging Linux-based cloud architecture, modern memory and storage management, and secure distributed systems design, **Draw It or Lose It** is positioned to become a responsive, scalable, and trusted multiplayer experience on any platform.