

GamingRoom

# **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 | 06/22/25 | Sephane Nidri | GamingRoom Revision 2 |

**Module 7 work Added to page 10 to page 14**

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room has requested a solution to expand the reach of their game, *Draw It or Lose It*, originally developed for Android platforms, to a broader audience by deploying it as a modern web-based application accessible across desktop (Windows, Mac, Linux) and mobile (iOS, Android) platforms. The goal is to create a responsive, browser-based version of the game that supports real-time play and can scale to thousands of concurrent users, all while leveraging existing code structures and incorporating effective software design patterns.

## Requirements

#### Business Requirements

1. **Platform Expansion**: The game must expand beyond Android to support iOS, Windows, macOS, and Linux users via a responsive web-based interface.
2. **Cost Efficiency**: The solution should leverage cost-effective platforms and open-source tools wherever possible to minimize long-term expenses.
3. **Scalability**: The application must support scaling to thousands of concurrent users, especially during peak play hours.
4. **Cross-Team Collaboration**: The development process may involve multiple teams (front-end, back-end) and must support effective collaboration.

#### User Requirements

1. **Accessibility**: Users must be able to access the game from any modern web browser on desktop or mobile devices.
2. **Consistent Gameplay**: The experience, features, and performance must be consistent across all supported platforms.
3. **Responsive Design**: The user interface must adjust automatically to different screen sizes and orientations.
4. **Touch and Mouse Input**: The application must support both touch (mobile) and mouse/keyboard (desktop) inputs seamlessly.

#### Technical Requirements

1. **Web-Based Delivery**: Use HTML5, CSS3, and JavaScript for the front-end, served via a web server on Linux or another platform.
2. **Backend Implementation**: Retain the existing Java codebase for business logic, deployed in a web service framework (e.g., Dropwizard or Spring Boot).
3. **Design Patterns**: Continue applying the Singleton pattern for game service management and Iterator pattern for accessing game elements safely.
4. **Platform Compatibility**: Ensure compatibility with major web browsers (Chrome, Firefox, Safari, Edge) and mobile platforms (iOS Safari, Android Chrome).
5. **Licensing and Tools**: Use free/open-source IDEs like IntelliJ IDEA Community, Eclipse, or VS Code. Apple Developer Program is required for iOS testing/distribution.
6. **Security and Data Integrity**: Implement secure communication (HTTPS), input validation, and proper user session management.

## [Design Constraints](#_2et92p0)

#### 1. Platform Independence (Browser-Based Delivery)

* **Constraint**: The application must run in all modern browsers across multiple platforms (Windows, macOS, Linux, Android, iOS).
* **Implication**: Developers must use platform-neutral technologies like HTML5, CSS3, and JavaScript for the front-end. Native mobile or desktop features (e.g., file system access, local storage APIs) may have limited or inconsistent behavior across platforms, requiring fallback mechanisms and extensive browser testing.

#### 2. Responsive Design

* **Constraint**: The user interface must dynamically adapt to different screen sizes and device orientations.
* **Implication**: Designers must use flexible layouts (e.g., CSS Grid, Flexbox) and test UI responsiveness thoroughly. Additional UI/UX design time is needed to ensure accessibility and ease of use on both desktop and mobile devices.

#### 3. Stateless Communication (Web-Based)

* **Constraint**: Web applications operate over HTTP(S), which is stateless.
* **Implication**: Developers must implement session management techniques (e.g., JWT, cookies, or server-side sessions) to maintain game state across multiple requests and users, adding complexity to authentication and game continuity logic.

#### 4. Real-Time Interactivity

* **Constraint**: Multiplayer features and time-based rounds require low-latency communication and real-time updates.
* **Implication**: Developers may need to integrate technologies like WebSockets or server-sent events to provide bi-directional, real-time data exchange, especially for managing turns and countdown timers.

#### 5. Security Requirements

* **Constraint**: The application must ensure secure access, data privacy, and prevent cheating or unauthorized access.
* **Implication**: Secure communication channels (HTTPS), input sanitization, authentication, and role-based access control are required. Developers must follow security best practices, increasing overall development effort and testing.

#### 6. Server Load and Scalability

* **Constraint**: The game must scale to support thousands of concurrent users.
* **Implication**: The server-side architecture must be designed to scale horizontally (e.g., via cloud hosting, load balancers, microservices). Developers must implement performance optimizations and monitoring tools, possibly increasing infrastructure and deployment complexity.

#### 7. Code Reuse and Maintainability

* **Constraint**: The existing Java-based game logic must be reused for back-end services.
* **Implication**: Developers must integrate the existing code into a web framework (e.g., Dropwizard or Spring Boot) and expose it via REST APIs. This requires refactoring and separating UI logic from business logic, which adds to initial setup and testing.

## [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)

<Describe the UML class diagram provided below. Explain how the classes relate to each other. Identify any object-oriented programming principles that are demonstrated in the diagram and how they are used to fulfill the software requirements efficiently.>

**"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** | Android | **IOS** |
| --- | --- | --- | --- | --- | --- |
| **Server Side** Deployment | Can be used as a development server, but not commonly used for production hosting. MacOS Server is limited. | Excellent support for web hosting; preferred for scalable applications; supports Apache, NGINX, Docker, Kubernetes. | Windows Server supports IIS and can host Java applications, though less common for large-scale Java deployments. | Not applicable for server deployment. | None for open-source development. |
| Licensing Costs | High – requires Apple hardware & licensing | Low to none (open source). | Medium to high (Windows Server licenses). | None for open-source development. | equires Apple Developer Program ($99/yr). |
| Client-Side Considerations | Supports Safari, Chrome, Firefox. Less flexibility for server deployment but good for iOS simulator testing | Fully supports modern browsers (Chrome, Firefox). Supports web-based mobile emulation for testing. | Compatible with all major browsers. Slightly more complex to configure for Java web services. | Native support already exists. Web version will run on mobile Chrome/Firefox. Needs responsive UI. | Needs testing on Safari mobile and iOS simulators. Requires responsive, touch-friendly design. |
| Development Tools & Requiremen | Languages: Swift (iOS), JavaScript, Java. Tools: Xcode (iOS dev), IntelliJ. Requires Apple Developer account ($99/yr). | Languages: Java, JavaScript, HTML/CSS. Tools: Eclipse, IntelliJ IDEA, VS Code. No licensing costs. | Languages: Java, C#, JavaScript. Tools: Visual Studio (cost varies), IntelliJ, Eclipse. Some licensing costs apply. | Languages: Java/Kotlin, JavaScript. Tools: Android Studio. No licensing costs. | Languages: Swift (for native), JavaScript (for web). Tools: Xcode, browser dev tools. |

## 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**:

**Recommendation**: Adopt a **Linux-based cloud platform** (e.g., Ubuntu on AWS, Azure, or Google Cloud) as the primary operating platform for hosting the backend of *Draw It or Lose It*.

* **Reasoning**: Linux is open-source, cost-effective, secure, highly stable, and widely supported in cloud environments. It integrates well with Java-based web frameworks and supports scalability via containers (Docker) and orchestration (Kubernetes).

1. **Operating Systems Architectures**:

**Description**:  
 Linux follows a **monolithic kernel architecture** with modularity and efficient hardware access.

* Supports multi-threading and multi-user environments.
* Ideal for running web servers, REST APIs, and microservices.
* Compatible with various hardware (x86, ARM) and virtualization solutions.
* Easily deployable using **Docker containers** for consistent development, testing, and production environments.

1. **Storage Management**:

**Recommendation**: Use **cloud-managed object and block storage systems** alongside Linux’s file system management.

* **Example**: Amazon S3 (object storage) for image assets and logs, and Amazon EBS or Google Persistent Disks for application data and database volumes.
* **Database Recommendation**: Use **PostgreSQL** or **MySQL** for structured data (games, teams, players).
* **Backup and Redundancy**: Enable automated backups and geo-redundancy for high availability and disaster recovery.

1. **Memory Management**:

**Linux Memory Management Techniques**:

* **Virtual Memory**: Isolates application memory to prevent crashes from affecting others.
* **Paging & Swapping**: Manages RAM efficiently when under pressure by using disk space.
* **Caching**: Frequently accessed files and database results are cached to enhance performance.
* **Garbage Collection**: Java Virtual Machine (JVM) running the backend manages object memory cleanup automatically.
* **Implication for Draw It or Lose It**: Ensures smooth performance under varying load conditions, even with large numbers of players or concurrent sessions.

1. **Distributed Systems and Networks**:

**Communication Across Platforms**:

* Use **RESTful APIs** for communication between client applications (browsers, mobile devices) and the server.
* Employ **WebSockets** for real-time features (game rounds, timers, multiplayer interaction).
* Deploy a **Content Delivery Network (CDN)** for static assets to reduce latency globally.

1. **Security**:

* Ensure **network reliability** through redundancy (load balancers, failover nodes).
* Implement **health checks** and **auto-scaling** to handle usage spikes.
* Use **logging and monitoring tools** (like ELK stack or Prometheus + Grafana) to track distributed service performance.

Recommendations:

To support The Gaming Room’s goal of scaling *Draw It or Lose It* into a robust, platform-independent, multiplayer web game, I have assessed multiple system architecture strategies. The following recommendations prioritize flexibility, reliability, and performance across desktop, mobile, and web-based environments.

### **1. Operating Platform**

**Recommendation:**  
 Deploy the backend on **Ubuntu Linux (LTS)** using a cloud-native environment such as **Amazon Web Services (AWS)**, **Google Cloud Platform (GCP)**, or **Microsoft Azure**.

**Rationale:**

* **Stability and Performance:** Linux is the industry standard for web application servers, known for reliability and high uptime.
* **Cost Savings:** Open-source licensing eliminates the costs associated with proprietary platforms.
* **Cloud-Native Integration:** Offers seamless compatibility with tools like Docker, Kubernetes, and CI/CD pipelines, supporting efficient deployment in distributed environments.
* **Cross-Platform Accessibility:** Supports clients on Android, iOS, Windows, macOS, and web browsers without modification to backend services.

### **2. Operating System Architecture**

**Linux Architecture Summary:**

* **Monolithic Kernel + Modular Components:** Delivers efficient hardware control while allowing extensibility for drivers and system services.
* **POSIX-Compliant:** Facilitates consistency across scripts and development tools.
* **Supports Multithreading & Multiprocessing:** Handles simultaneous game sessions and concurrent user actions with minimal overhead.
* **Built-in Security:** Tools like AppArmor and SELinux offer process-level isolation, and Linux supports encrypted storage and strict file permissions.

**Relevance to Draw It or Lose It:**  
 The modularity and multitasking capabilities of Linux allow for independent services (game logic, scoring, session handling) to operate concurrently for a multiplayer, real-time experience.

### **3. Storage Management**

**Recommendation:**  
 Implement a **hybrid cloud storage approach** combining object storage, relational databases, and block storage.

**Configuration Details:**

* **Object Storage (such as AWS S3, Google Cloud Storage):**  
   Used for static resources such as game images, logs, and media assets such graphics, images, text, video and more. Offers scalability, geo-redundancy, and lifecycle versioning.
* **Relational Database (e.g., PostgreSQL, MySQL):**  
   Ideal for structured data like player profiles, team scores, and game states. Supports ACID (Atomic , Consistency, Isolation, Durability) transactions and complex queries for analytics and leaderboards.
* **Block Storage (e.g., AWS EBS):**  
   Attached to compute instances for fast access to cache snapshots or temporary runtime data.

**Advanced Features:**

* Automated backups
* Cost control through data lifecycle policies
* Seamless integration with Kubernetes volume mounts and serverless functions

### **4. Memory Management**

**Linux Memory Management Techniques for Draw It or Lose It:**

|  |  |
| --- | --- |
| **Technique** | **Purpose** |
| **Virtual Memory** | Isolates processes, preventing interference between game sessions. |
| **Paging & Swapping** | Frees physical RAM during peak usage by offloading to disk. |
| **Caching** | Stores frequently accessed game data (e.g., drawings, metadata) in memory for fast retrieval. |
| **JVM Garbage Collection** | Automatically reclaims unused memory in Java-based components like GameService, reducing leak risk. |

### **5. Distributed Systems and Networks**

**Recommended Architecture:**

* **Microservices backend** exposing RESTful APIs and WebSocket connections
* **Stateless frontend clients** (web or mobile) communicating securely via HTTPS and real-time protocols

**Key Technologies:**

* **RESTful APIs:** Handle requests for login, game creation, scoring, and player/team management
* **WebSockets:** Facilitate real-time communication for drawing updates, timers, and multiplayer events
* **CDNs (e.g., Cloudflare):** Accelerate global delivery of static assets and reduce latency
* **Service Discovery (e.g., Kubernetes Ingress):** Enables scalable service-to-service communication

**Network Resilience Tactics:**

* **Redundant Deployments:** Span services across multiple availability zones
* **Health Checks & Auto-restarts:** Detect and recover from failures
* **Fault Tolerance:** Implement fallback strategies such as retry mechanisms and connection reconnection logic for WebSocket outages

### **6. Security**

**End-to-End Security Strategy:**

|  |  |
| --- | --- |
| **Area** | **Mechanism** |
| **User Authentication** | Secure login using OAuth 2.0, JWT, or OpenID Connect |
| **Data in Transit** | Enforce HTTPS with TLS encryption |
| **Data at Rest** | Apply field-level encryption for sensitive values (e.g., emails, scores) |
| **Access Control** | Role-based access control (RBAC) for players, guests, and admins |
| **Threat Detection** | Continuous monitoring via AWS GuardDuty, Azure Defender, or GCP Security Command Center |
| **Logging & Auditing** | Centralized log aggregation using ELK stack with alerting for anomalies |

**Cross-Platform Security Considerations:**  
 Maintain consistent security policies across desktop and mobile environments:

* Use CORS headers for secure cross-origin requests
* Apply secure cookies and tokens
* Implement certificate pinning in native mobile apps

### **Conclusion**

By leveraging a Linux-based cloud architecture, containerized microservices, and robust distributed networking, *Draw It or Lose It* can evolve into a highly scalable, secure, and cross-platform game. This foundation enables The Gaming Room to support real-time multiplayer gameplay across regions and devices—without compromising performance, data integrity, or user trust.