

# Draw It or Lose It

# **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 7**

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 07/18/2025 | Jude Mensah Nana Baffoe | The Summary, Requirements, Constraints, and other information needed for this project have been updated |
| 1.1 | 07/29/2025 | Jude Mensah Nana Baffoe | Updated strengths, weaknesses, and characteristics of the various operating systems |
| 1.2 | 08/15/2025 | Jude Mensah Nana Baffoe | Updated Recommendations to accommodate multiple platforms |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room seeks to bring its popular Android game, *Draw It or Lose It*, to the web, enabling cross-platform play. As a technology consultant for Creative Technology Solutions (CTS), you are tasked with designing a scalable, maintainable architecture and a functional Java application prototype. The proposed design leverages industry-standard software design patterns to ensure that requirements such as unique naming, single-instance management, and extensible team/player logic are met efficiently. This foundational work will ease future adaptations and deployment decisions.

## Requirements

Using the Singleton pattern ensures only one instance of the game is maintained in memory, which reduces redundancy and improves consistency. The requirement for unique naming across instances is supported by using the Iterator pattern to traverse and validate lists efficiently. The web-distributed nature also demands secure data handling and session control, which is addressed by centralized game logic.”

## [Design Constraints](#_2et92p0)

* **Distributed Web Environment:**  
  The application is intended to be web-based, introducing requirements for remote accessibility, possible concurrent users, and statelessness within individual client sessions.
* **Single Game Instance:**  
  Only one game instance should reside in memory at any one time. This constraint is addressed using the Singleton pattern, ensuring data consistency and preventing duplication.
* **Unique Naming:**  
  Both games and teams require globally unique names. This not only helps with user experience (avoiding confusion) but also with backend management and data validation.
* **Extensibility:**  
  The system must allow for future enhancements, such as additional game modes or new team/player features, with minimal impact on the core architecture.
* **Performance:**  
  Response times must remain low, even as the number of teams and players rises, necessitating collections optimized for searching and adding unique entries.

## [System Architecture View](#_ilbxbyevv6b6)

To satisfy these constraints, the application requires robust state management, efficient lookup structures for uniqueness, and maintainable, object-oriented code to foster scalability and maintainability over time.

## [Domain Model](#_8h2ehzxfam4o)

The software design leverages a clear Domain Model reflected in the UML diagram:

* **Entity (Abstract Base Class):**  
  Contains the attributes id (unique identifier) and name. This centralizes common features for all domain objects.
* **Game (Subclass of Entity):**  
  Represents the game instance. Contains a collection of Team objects, ensuring only one active game at a time via the Singleton pattern.
* **Team (Subclass of Entity):**  
  Contains a unique name and a collection of Player objects. Responsible for managing its players and ensuring no duplicate team or player names exist within its scope.
* **Player (Subclass of Entity):**  
  Represents an individual participant on a team, inheriting unique identification and naming from Entity.

**"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)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | macOS can host web-based applications but is rarely used in production due to licensing costs, hardware requirements, and limited server management tools. Its Unix base ensures reliability and supports most open-source stacks; however, fewer cloud providers offer macOS VMs, and enterprise support is limited compared to Linux and Windows. | Linux is the industry standard for web app hosting, offering stability, scalability, robust security, and low resource costs. Its vast community, open-source nature, and support for virtually all modern web tech make it ideal for flexible, efficient web servers. Weaknesses include a steeper learning curve for administrators unfamiliar with Unix-like systems. | Windows servers are widely used in enterprises requiring integration with Microsoft technologies (like .NET or SQL Server). They offer strong UI management tools and broad vendor support, but tend to have higher licensing and maintenance costs. Windows also has more frequent updates and is potentially higher vulnerable to attacks targeting its user base. | Mobile devices are not suitable for hosting web-based applications due to limited resources, battery dependence, and a lack of server-oriented management features. They are designed for client-side app consumption, not for reliable, long-running server operations. |
| **Client Side** | Development for Mac requires access to macOS hardware (for Xcode and iOS testing). Costs are higher due to Apple's premium pricing. Expertise in Swift/Objective-C is often needed for native apps. Supporting multiple client types (Mac, iOS) can require more development time unless a cross-platform framework is used. | Linux desktops are less common among consumers but are favored in developer and academic settings. Supporting Linux clients often involves additional testing for compatibility, but can be straightforward for web-based applications accessed via browsers. Cost is minimal, but Linux-specific desktop app development requires expertise in tools like GTK or Qt. | Windows dominates the desktop market, making Windows client support essential for broad reach. Development tools (Visual Studio, . .NET) are well-supported, and cost is usually lower than Mac, as inexpensive Windows hardware is widely available. Development time may increase when ensuring compatibility with legacy systems. | Client-side development for mobile devices involves targeting iOS and Android, which often require separate codebases (Swift/Objective-C for iOS, Kotlin/Java for Android) unless using cross-platform tools like Flutter or React Native. Additional time and expertise are required to ensure optimal usability and function on diverse devices. Testing costs can be significant due to OS fragmentation. |
| **Development Tools** | Key languages: Swift, Objective-C, Java (for cross-platform). IDEs: Xcode (for iOS/macOS), Eclipse/IntelliJ (for Java web apps). Web/distributed app dev is well supported via Docker, Node.js, and Java. Limited cloud deployment options for macOS. | Most programming languages and frameworks are first-class citizens on Linux: Java, Python, Node.js, PHP, Ruby, Go, etc. Tools: Eclipse, IntelliJ, VS Code, Git, Docker. Linux servers are central to cloud-native and containerized app development. | Languages: C#, .NET, Java, Python. IDEs: Visual Studio, Eclipse, IntelliJ, Visual Studio Code. Some open-source tools are less mature on Windows, but Windows supports a broad array of enterprise tools and cloud integration features. | Languages: Swift, Objective-C (iOS); Java, Kotlin (Android); Dart (Flutter), JavaScript/TypeScript (React Native). Tools: Android Studio, Xcode, Flutter, React Native CLI. Emulators and device managers are needed for testing. Cross-platform frameworks can reduce code duplication. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Criteria | Linux | Mac | Windows | Mobile (Android & iOS) |
| **Server Hosting** | Highly stable; widely used in web hosting; robust security features | Possible, but less common for large-scale production web servers; server-class hardware rare | Supported, but less common for large-scale production hosting; more often used in business environments | Not commonly used for server hosting; more suited to client-side application |
| **Server Licensing Costs** | Generally free (open source distros, e.g., Ubuntu Server, CentOS, Debian); commercial support (Red Hat, SUSE) costs extra | macOS Server has licensing fees; runs only on Apple hardware; limiting/costly | Windows Server OS requires per-instance licensing; Client Access Licenses (CALs) add cost | N/A for server hosting |
| **Server Support & Scalability** | Extremely scalable; strong community/commercial support; container (Docker, Kubernetes) friendly | Limited scalability in enterprise server environments | Scalable, though historically less efficient for web scale; Windows containers/Kubernetes available | N/A for server hosting |
| **Client Delivery (Web App)** | Standard browsers (Chrome, Firefox); responsive HTML works well | Standard browsers (Safari, Chrome, Firefox); supports modern HTML/CSS/JS | Standard browsers (Edge, Chrome, Firefox); supports modern HTML/CSS/JS | Mobile browsers (Safari iOS, Chrome Android); touch optimization essential; responsive design required |
| **Software Development Considerations** | Dev testing on Linux needed; ensures server app consistency; cross-platform browser testing | macOS testing needed for Safari quirks and Apple-specific behaviors; limited to Apple hardware | Windows testing essential (Edge-specific quirks); prominent in schools/business | Must ensure responsive design, touch input support, compatibility with both iOS and Android browsers |
| **Development Tools** | Programming: Node.js, Python, Java, etc. IDEs: VS Code, JetBrains, Eclipse. Most are free/open-source. | Programming: Same languages as previously stated; Xcode required for iOS native dev; IDE licensing for JetBrains; Xcode is free | Programming: Same; Visual Studio (free/community & paid/enterprise), Visual Studio Code, JetBrains | Android Studio, Xcode (free, macOS only for iOS builds), cross-platform frameworks (React Native, Flutter, etc.) |
| **Development Team Impact** | One full-stack team can build browser-based client/server, but thorough cross-platform/browser testing required | May require Mac hardware/team for iOS builds and Safari testing | Windows experts for Windows-specific testing and deployment, though basic dev is cross-platform | Android/iOS experts for testing; may need Mac hardware for App Store submission if native clients are later considered |
| **Development Tool Licensing Costs** | Mostly free/open source; JetBrains, enterprise support, or commercial tools incur costs if needed | Xcode free; JetBrains paid if used; Mac hardware costs higher | Visual Studio free or paid; non-trivial for enterprise; Windows OS licenses if not already owned | Android Studio free, Xcode free (but Mac needed), possible costs for cross-platform tool licenses |
| **Strengths** | Cost-effective, scalable, robust, secure, ideal for high-traffic web applications | Excellent for iOS development; strong integration with Apple ecosystem | Familiar to many developers/businesses; powerful dev tools; widely used in enterprise | Reach broad user base; designed for mobile experience; supports web-oriented deployment to avoid native-only development |
| **Weaknesses** | Smaller pool of desktop users, steeper CLI/server admin learning curve | High hardware costs; server deployment not enterprise standard; limited hardware choices | Licensing costs, lower server usage share, complexity of Windows server management | Device/browser fragmentation; requires careful touch/responsive UX; Apple restricts iOS build/deploy to Macs |

## Recommendations

**1. Operating Platform:**

Linux is the best choice for hosting Draw It or Lose It due to its dominance in web server environments, scalability, robust security, low operational cost, and compatibility with all major development frameworks. For client-side mobile support, target both iOS and Android, utilizing cross-platform frameworks where possible.

**2. Operating Systems Architectures:**

Deploy the server using a Linux-based architecture, ideally hosted by a cloud provider such as AWS, Azure, or Google Cloud. The application should be built in Java, using object-oriented approaches and running in Docker containers for scalability and easy updates. Clients should connect via secure, modern technologies:

* Web: Modern browser-based clients (React, Angular, etc.)
* Desktop: Lightweight Mac/Windows apps when necessary, or Progressive Web Apps (PWAs)
* Mobile: Native or cross-platform apps via REST API

**3. Storage Management:**

Use a strong, relational database like PostgreSQL or MySQL for transactional data, and cloud storage (e.g., AWS S3) for static assets (such as images). Linux file systems (EXT4, XFS) support backup and recovery.

**4. Memory Management:**

Linux provides powerful memory management, including virtual memory, process/container isolation, and resource limits. The Java Virtual Machine (JVM) adds garbage collection and dynamic allocation, which supports both performance and stability, even under heavy concurrency.

**5. Distributed Systems and Networks:**

Use RESTful APIs or Web Sockets for real-time, cross-platform communication between clients and the server. Leverage cloud services for high availability, load balancing, and global distribution. The architecture should be tolerant of connectivity issues and include authentication and version control for distributed components.

**6. Security:**

Linux provides fine-grained user permissions, built-in firewalling, and robust encryption support. Use HTTPS for all communications, secure authentication (like OAuth2 or JWT) for users, and rigorous input validation. Data should be encrypted both in transit and at rest, taking full advantage of modern cloud security practices.

**Recommendation for Characteristics, advantages, and weaknesses of the various Operating Systems.**

**Server Side**:  
Linux is the industry standard for web servers due to its cost efficiency, stability, and scalability. Windows can work but it adds licensing and, sometimes, complexity. macOS is rarely used for server deployments due to hardware and cost limitations. Mobile is not a server OS.

**Client Side**:  
Delivering as a web app is the smartest move because it supports all platforms (desktop and mobile) via modern browsers, reducing development and maintenance efforts compared to native-only. Responsive web design ensures usability across devices. However, exhaustive cross-platform and browser testing is essential due to varying quirks (especially on Safari and Edge, as well as mobile browsers).

**Development Tools**:  
Most major languages, frameworks, and IDEs support all target platforms. To *submit* iOS apps to Apple’s App Store, a Mac is required (even if building a web app now, this will matter for any possible native app or push notifications in the future). Most essential tools are free, but some IDEs (like JetBrains products) and commercial support add cost.

**Team Structure**:  
A unified full-stack development team can usually tackle a web app with server and browser clients, though additional quality assurance is needed for edge-case platforms. For any native mobile work, iOS developers will need Mac hardware.

**Key Recommendations**

* **Host game servers on Linux for cost, scalability, and reliability.**
* **Develop a responsive web app** to ensure the widest platform support and futureproofing.
* **Test extensively on major browsers and devices.**
* **Budget for macOS devices** if submitted to the iOS App Store or deep iOS feature integration is anticipated.
* **Consider the total cost of ownership**—Linux offers the lowest server costs, but integration and QA for multiple clients (desktop/mobile/browser) should not be underestimated.

**Recommendations for The Gaming Room – Software Design Document**

**Operating Platform**

For expanding "Draw It or Lose It" to multiple environments, it is recommended that The Gaming Room use a cloud-based, cross-platform server solution. A platform such as **Amazon Web Services (AWS)**, **Microsoft Azure**, or **Google Cloud Platform (GCP)** allows hosting game logic and data centrally while enabling access and synchronization across devices running different operating systems (Windows, macOS, Linux, Android, iOS). These platforms offer scalable resources, high availability, and robust security features essential for modern multiplayer games.

**Operating Systems Architectures**

The recommended operating platforms support several architectural models:

* **Monolithic Kernel (e.g., Linux, Windows):** Combines core OS services in one large kernel. Offers efficient communication between kernel components and is used widely for game servers.
* **Microkernel (e.g., macOS):** Separates basic services (memory, process management) from other system services, increasing stability.
* **Hybrid Architecture:** Many modern systems blend monolithic and microkernel advantages, improving performance and security.

For distributed gaming environments, **containerization** (e.g., Docker) and **orchestration** (e.g., Kubernetes) allow running lightweight, isolated environments on any OS, maximizing compatibility and maintainability.

**Storage Management**

A **cloud storage solution** (such as AWS S3 or Azure Blob Storage) is recommended. It:

* Supports large-scale multimedia storage.
* Allows fast retrieval and replication across data centers.
* Offers built-in redundancy and automatic backups.
* Can be paired with database services such as SQL for game state and user profiles.

**Memory Management**

Memory management for the recommended platforms is handled using:

* **Virtual Memory:** Allows running processes to use more memory than physically available; the OS exchanges unused data to disk.
* **Dynamic Allocation:** Modern programming languages and platforms handle memory allocation and garbage collection, reducing leaks.
* **Caching:** Frequently accessed assets or game states are held in memory (RAM or in-memory databases like Redis) for rapid access, improving responsiveness in multiplayer scenarios.

**Distributed Systems and Networks**

To enable "Draw It or Lose It" on multiple platforms:

* The game should use client**-server architecture** with RESTful APIs or Web Sockets for real-time data exchange.
* **Load balancing** distributes workload across servers for scalability.
* **Redundancy and failover strategies** (multi-region deployment) minimize impact from outages.
* **Service discovery and health checks** ensure only functioning components handle data.
* Dependencies such as latency and connectivity are managed through protocol optimization and server proximity to major user populations.

**Security**

Protecting player data and communication involves:

* **Encryption in transit and at rest:** Use HTTPS/TLS for data transmission and encrypt stored data.
* **User Authentication and Authorization:** Implement OAuth, security tokens, and role-based access controls.
* **Regular Security Audits and Patch Management:** Monitor for emerging vulnerabilities and apply updates.
* **Firewall and Intrusion Detection Systems:** Limit and monitor inbound/outbound connection attempts.
* **Monitoring and Logging:** Track user activity and access for rapid incident response.
* **Compliance:** Adhere to standards such as GDPR or CCPA, ensuring user protection globally.

A cloud-based, cross-platform server architecture leveraging containerization, robust storage and memory management, secure networking, and modern distributed system practices will enable The Gaming Room to reliably and securely expand "Draw It or Lose It" for users on any operating system. This solution ensures high performance, scalability, security, and data integrity as the game grows and evolves.