

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 5**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 07/19/25 | Samuel Billingsley | Initial version of software design document for Draw It or Lose It web application. |

**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 requires assistance in transitioning their Android game *Draw It or Lose It* to a web based cross platform application. The game involves teams competing to guess puzzles revealed through stock images across four timed rounds. Challenges include enforcing unique game and team names, maintaining a single game instance in memory, and supporting real time multiplayer interactions.

The solution for this involves developing a Java based web application using Spring Boot for the backend, ensuring compatibility across Windows, Mac, Linux, and mobile browsers. We would also need to implement the singleton pattern to manage game instances, the iterator pattern for validating unique names, and WebSocket technology for real time gameplay synchronization. The client should note that this approach requires HTTPS for security and relies on browser based delivery to avoid platform specific native apps.

## Requirements

*<* Please note: While this section is not being assessed, it will support your outline of the design constraints below. *In your summary, identify each of the client’s business and technical requirements in a clear and concise manner.>*

## [Design Constraints](#_2et92p0)

Developing a web based game introduces several constraints. The requirement for web deployment necessitates using HTTP/HTTPS protocols, addressed through a Spring Boot backend with embedded Tomcat. Cross platform compatibility demands a responsive HTML5 frontend to ensure consistent performance across desktop and mobile browsers. The unique naming requirement for games and teams requires implementing name validation checks, solved through the iterator pattern when adding new entities.

The single instance constraint mandates strict control over game object creation, enforced through the singleton pattern in the GameService class. Real time gameplay synchronization for timed rounds and drawing reveals requires WebSocket support to broadcast game state updates to all connected clients. These constraints collectively guide the architecture toward a Java or Spring backend with a JavaScript frontend, using PostgreSQL for relational data storage given its robustness in handling team and player relationships.

## [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)

The UML class diagram represents the application's core structure through several key relationships. The Entity class serves as the base class, providing common attributes like ID and name to its subclasses: Game, Team, and Player. This inheritance hierarchy promotes code reuse and consistent behavior across all entity types. The GameService class maintains a collection of Game objects through a composition relationship, reflecting its role as the central manager for game instances.

Object oriented principles are demonstrated in the design. The singleton pattern in GameService ensures only one instance exists, meeting the client's single instance requirement. Encapsulation is enforced through private fields with public accessors in all classes. The iterator pattern supports unique name validation by iterating through existing entities before allowing new additions. Polymorphism allows Game, Team, and Player objects to be treated uniformly through their shared Entity superclass when performing common operations.

**"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** | Mac servers aren’t used in production environments very often due to limited hardware options and higher costs. While macOS can run web servers like Apache or Nginx, its primary advantage is the Unix based foundation that provides stability similar to Linux. The main weaknesses include expensive Apple hardware requirements and lack of support compared to Linux or Windows Server. | Linux is a better choice for server deployment, offering stability, security, and cost effectiveness as an open source platform. Its advantages include cloud compatibility, lightweight operation, and strong community support. The only potential weakness is that some proprietary software may require additional configuration compared to Windows environments. | Windows Servers provide a robust environment with excellent .NET framework support and familiar management tools. Advantages include seamless integration with other Microsoft products and strong commercial support. However, licensing costs are higher than Linux, and resource requirements are heavier which makes scaling more expensive. | Mobile devices are not suitable for hosting server side applications due to hardware limitations, restricted background processes, and battery life concerns. While possible to run lightweight servers on jailbroken devices, this approach would be impractical for production use, with severe performance constraints and security risks. |
| **Client Side** | Mac clients require Safari compatibility testing and optimization for trackpad gestures. Development considerations include ensuring proper font rendering and testing Retina display support. The relatively small market share reduces priority compared to Windows, but Mac users expect premium UX, increasing design time requirements. | Linux desktop clients present minimal special requirements beyond standard web compatibility testing. The main consideration is supporting multiple browser variants across different distributions. Development costs are low as no platform specific code is needed beyond standard responsive web practices. | Windows clients represent the largest desktop user base, requiring thorough testing across multiple operating systems. Considerations include legacy system support and touchscreen compatibility for convertible devices. Development time is highest for Windows due to the need for extensive browser version testing. | Mobile clients require more adaptation for touch interfaces, limited screen sizes, and intermittent connectivity. Both Android and iOS need dedicated testing for mobile browsers. Progressive Web App development adds approximately 30% more time compared to desktop only development but avoids native app store requirements. |
| **Development Tools** | Xcode provides excellent tools for Safari debugging and iOS simulator testing. Cross platform IDEs like VS Code and JetBrains products work well on Mac. The Unix environment allows seamless compatibility with Linux server tooling. No additional licensing costs beyond standard Apple hardware requirements. | Linux offers the most cost effective development environment with free access to all necessary tools like VS Code, Eclipse, and Docker. The native environment matches production servers exactly. Weaknesses include occasional driver compatibility issues and less commercial IDE support compared to Windows. | Windows provides the broadest IDE selection including full Visual Studio capabilities. WSL2 enables Linux compatibility for backend development. Licensing costs apply for premium tools, but the platform offers the most comprehensive debugging and profiling capabilities for complex applications. | Mobile development requires browser dev tools plus device emulators or simulators. Android Studio and Xcode are essential. Physical device testing adds hardware costs. Cross-platform tools like Flutter or React Native may be considered but add complexity versus pure PWA approach. |

## 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 Draw It or Lose It's multi platform expansion across Windows, macOS, Linux, mobile devices, and future console support, we recommend deploying the game servers on a Linux based cloud platform using either Ubuntu Server or Amazon Linux, hosted on AWS or Microsoft Azure. This approach provides cross platform compatibility through standardized REST APIs while ensuring scalability to handle multiple concurrent game instances. The cloud based model offers cost efficiency through pay as you go pricing and high availability through redundant server deployments across regions. This foundation will support current requirements while maintaining flexibility for future platform expansions.
2. **Operating Systems Architectures**: The recommended Linux server environment utilizes a monolithic kernel architecture optimized for performance and stability, with components including the kernel for process and hardware management, system libraries for core functions, an efficient networking stack, and a Virtual File System for storage abstraction. Client platforms will interact with this server using their native OS architecture. Windows and macOS employ hybrid kernel designs balancing performance and modularity, Android uses a modified Linux kernel with mobile optimizations, iOS runs on Apple's XNU kernel, and gaming consoles utilize proprietary operating systems with specialized APIs. This diversity necessitates careful API design to ensure consistent functionality across all platforms.
3. **Storage Management**: The game's storage requirements, particularly the 200+ high definition image files totaling approximately 1.6GB, will be managed through a tiered storage architecture. Primary storage will utilize AWS S3 for cost-effective, durable cloud storage of all game assets, with frequently accessed images cached on local SSDs for rapid retrieval. Structured game data will reside in a PostgreSQL database, supplemented by Redis for in memory caching of active session data. The Linux servers will employ the Ext4 file system for reliable local storage operations, with image assets converted to WebP format to reduce storage footprint without compromising quality. This combination provides the necessary balance between storage capacity, access speed, and cost efficiency.
4. **Memory Management**: Memory optimization will employ several techniques to ensure smooth gameplay with rapid image rendering. Dynamic loading will ensure only currently needed images occupy memory, with assets purged after use. The Java and .NET client implementations will utilize garbage collection for automatic memory management, while the Linux servers will employ virtual memory through paging and swapping to prevent memory exhaustion. Frequently accessed assets will benefit from object pooling to minimize allocation overhead. Platform specific optimizations include aggressive memory trimming on mobile devices when backgrounded and direct memory access techniques for console versions to maximize rendering performance. These approaches collectively ensure efficient memory utilization across all supported platforms.
5. **Distributed Systems and Networks**: The cross platform multiplayer functionality will be enabled through a combination of RESTful APIs for standard game operations and WebSocket connections for real time updates like score tracking and game state changes. AWS Elastic Load Balancing will distribute traffic across multiple server instances, while CloudFront edge computing will ensure low latency content delivery globally. The system will incorporate redundancy through auto scaling server clusters, with client side caching allowing limited offline functionality during network interruptions. Continuous heartbeat monitoring will detect and mitigate network outages, maintaining service reliability. These distributed system components work together to provide a seamless multiplayer experience regardless of platform combinations.
6. **Security**: Comprehensive security protections will be implemented at multiple levels. All communications will use TLS 1.3 encryption, with sensitive user data encrypted at rest using AES-256 and passwords hashed with bcrypt. Authentication will leverage OAuth 2.0 for cross-platform identity management, integrating with platform specific services like Xbox Live and PlayStation Network where available, while JSON Web Tokens will handle session management. Platform native security features such as iOS Face ID and Android fingerprint authentication will be incorporated where supported. The system will maintain compliance with GDPR and CCPA regulations through data minimization practices and clear user consent mechanisms. Regular security audits and automated vulnerability scanning will ensure ongoing protection as the game evolves.