

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.1

## 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 |
| --- | --- | --- | --- |
| 1.1 | 10/18/25 | Daniel Dyall | Completed Recommendations section |

**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)

Draw It or Lose It is being redeveloped for web-based, multi-platform deployment to broaden its accessibility beyond Android. The client, The Gaming Room, requires a scalable and secure application supporting teams, unique naming for games and teams, and reliable real-time gameplay. The proposed solution leverages modular object-oriented design, the Singleton and Iterator patterns, and industry-standard security and web technology to meet these requirements efficiently while ensuring cross-platform support and maintainability. This document outlines all key requirements, system designs, and recommendations necessary for The Gaming Room to confidently proceed with development.

## Requirements

Support multiple, concurrent game sessions with teams and players

Enable unique game and team names for easy user validation

Ensure only one game instance exists in memory at a time

Robust data security for user and gameplay information

Performance suitable for real-time gaming and team guessing

Scalable web-based architecture to serve desktop and mobile clients

## [Design Constraints](#_2et92p0)

Developing Draw It or Lose It as a web-based distributed application introduces constraints including:

Network Reliability: Dependence on stable connections, requiring error handling for outages or latency.

Cross-Platform Compatibility: Code and frameworks must support deployment to Linux, Mac, Windows, and mobile devices.

Scalability: Architecture must handle increasing users without performance loss.

Security: Sensitive user data, including game states and names, must be protected using encryption and best practices.

Resource Management: Ensuring only one game instance is active in memory through the Singleton pattern.

These constraints inform technology choices (e.g., frameworks, databases) and development priorities.

## [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 illustrates:

GameService: Central class using the Singleton pattern to ensure exactly one active game instance, responsible for adding and retrieving games.

Game: Holds game state and references to participating teams.

Team: Manages players and their unique names.

Player: Each instance holds a unique name and participates in a team.

Entity (base class): All core classes inherit from Entity, standardizing identifiers and naming attributes.

Object-Oriented Principles:

Encapsulation: Attributes (id, name) kept private; accessed via getter/setter methods.

Inheritance: Game, Team, and Player inherit from Entity, minimizing code repetition.

Singleton Pattern: GameService uses a static instance and private constructor to permit only one object in memory.

Iterator Pattern: addGame(), getGame(), addTeam(), and addPlayer() use iteration to validate unique names and retrieve instances.

This structure supports all business requirements and allows for efficient code reuse and future extensions.

**"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** | Rarely used for large scale hosting. Good server admin tools. Higher hardware/software costs. | Industry standard, high stability, scalability, security, minimal/no licensing cost. Full enterprise/community support. | Robust, seamless Microsoft stack (.NET/IIS), strong enterprise support. Significant licensing cost if using Windows Server. | Relies on backend web servers running on desktop/server OS. Not suitable for direct server hosting. |
| **Client Side** | High usability, Safari/Chrome support. Device/OS version differences require careful testing. | Modern browser support (Chrome, Firefox). Flexible desktop env, robust compatibility. | Largest desktop base, broad browser support, strong accessibility. | Responsive web UI frameworks ensure compatibility; native apps need separate builds. Optimized for touch, battery, performance. |
| **Development Tools** | Xcode (free), Swift, Java, Node.js. No Xcode licensing cost when Macs available. Specialized Apple dev skills/teams needed for native work. Apple hardware may increase cost. | Eclipse, VS Code, Java, Python, Node.js—open source, multi-platform. No licensing cost. Unified web/backend teams. Open-source tools enable full-stack, unified teams. No significant licensing costs. | Visual Studio Community (free), Enterprise (paid), C#, Java, Node.js, .NET. Licensing may be required for Enterprise or Windows Server. Team may split for .NET specialization. | Android Studio (free), Xcode (free), Flutter, React Native—open source. Cross-platform frameworks unify mobile/web teams. Native development requires specialized teams. Apple developer fees for App Store submissions. |

## 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 server hosting, I recommend deploying the application on Linux. Linux offers proven scalability, high reliability, and low operational costs, making it ideal for supporting a web-based distributed game platform that needs to scale as user volume grows. Linux is widely adopted for enterprise gaming architectures, with robust community and vendor support, strong security modules, and minimal licensing expenses. For client-facing components, web and mobile frameworks such as React, Flutter, and Kotlin are recommended. These enable seamless cross-platform support and rapid development, helping to ensure that both web and mobile users enjoy consistent functionality and performance regardless of their operating system. This combination addresses the client’s need for broad accessibility while maintaining efficiency and cost control.
2. **Operating Systems Architectures**: I recommend utilizing a three-tier architecture that separates the user interface, business logic, and data persistence layers. The presentation layer will be built using responsive web frameworks, ensuring compatibility across desktop and mobile platforms. The business logic is deployed as stateless services over secure TCP/IP networking, with built-in load balancing and redundancy to ensure reliability and uptime. This architecture enables the system to scale horizontally and maintain high performance as user loads increase. Stateless communication also supports efficient error recovery and fast failover, which is vital for real-time gaming environments.
3. **Storage Management**: For storage management, I recommend implementing cloud-based relational databases such as PostgreSQL or MySQL. These platforms offer scalable storage solutions and support the unique constraints required for this project, such as enforcing uniqueness for team and game names. The database should be deployed with automated backup routines, encrypted storage, and disaster recovery capabilities to safeguard critical gameplay and user data. Utilizing cloud resources ensures minimal downtime and rapid data restoration in case of failure. This approach supports high transaction volume and maintains data integrity across distributed systems.
4. **Memory Management**: Effective resource management is paramount for game performance and reliability. The GameService central class will leverage the Singleton pattern to maintain only one active instance of each game, preventing redundant resource allocation and reducing the risk of leaks or stale data. Each game object and associated team/player instances should include explicit cleanup methods for memory deallocation when sessions end or objects are no longer needed. Regularly scheduled audits for memory usage and garbage collection will ensure the game remains responsive and efficient, even as multiple sessions run concurrently.
5. **Distributed Systems and Networks**: I recommend implementing robust RESTful APIs for all communication between platforms and device types. These APIs facilitate interoperability between web browsers, mobile devices, and backend servers, allowing the game to operate fluidly across multiple environments. The networking layer should be designed with strong error detection, retry logic, and session validation to maintain consistent gameplay even during transient connectivity issues. Load balancing strategies and distributed caching will further enhance performance and scalability, while encrypted connections protect data in transit.
6. **Security**: Security must be multilayered to defend both user identities and sensitive game data. I recommend integrating platform security modules such as OAuth for authentication, enforced encrypted storage using SSL/TLS protocols, and rigorous input validation throughout the application. Authentication should be role-based, ensuring that only authorized users can access administrative functions. All user and gameplay data must be encrypted both at rest and in transit. Regular vulnerability assessments and automated security updates are critical for protecting against emerging threats. These measures will collectively safeguard user trust and compliance with best practices for online gaming environments.