

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 3.0

## 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 | 09/21/2024 | Niaz Khan | Initial version for submission |
| 2.0 | 10/05/2024 | Niaz Khan | Updated evaluation and platform analysis |
| 3.0 | 10/19/2024 | Niaz Khan | Added recommendations |

## [Executive Summary](#_sbfa50wo7nsh)

"Draw It or Lose It" is a web-based distributed game application inspired by the 1980s game "Win, Lose or Draw." The game allows teams to compete by guessing puzzles from progressively revealed stock drawings. The design ensures that the game supports multiple teams, unique identifiers, and only one game instance at a time. The client, The Gaming Room, has requested an evaluation of various operating platforms to expand the application across desktop and mobile systems.

## Requirements

The client requires a game application that supports the following:

* Involvement of one or more teams per game
* Multiple players per team
* Unique game and team names
* Only one instance of a game in memory at a time, managed by unique identifiers
* Four rounds of play, with each round lasting one minute, and a drawing fully revealed at the 30-second mark

These requirements are critical to delivering an interactive and competitive gaming experience while maintaining efficiency and scalability in a web-based environment.

## [Design Constraints](#_2et92p0)

The development of "Draw It or Lose It" is subject to several design constraints:

* **Scalability:** The system needs to scale to support potentially large numbers of simultaneous users across multiple games. It requires efficient management of memory and resources, especially since only one instance of a game can exist at any given time.
* **Unique Identification:** Each game, team, and player must have a unique identifier to ensure that only one game is active in memory at a time and that there are no duplicate names, which affects the database design and search optimization.
* **Cross-platform Compatibility:** The game must be designed to run smoothly on web browsers across different operating systems (Windows, macOS, Linux) and mobile platforms (iOS, Android). This necessitates using web standards and responsive design principles.
* **Security:** Ensuring the safety of user data is paramount, particularly when dealing with multiple teams, unique identifiers, and interactions between distributed systems. Strong authentication mechanisms and secure data transmission protocols (e.g., SSL/TLS) are required.

These constraints have direct implications on the development approach, requiring careful architecture selection and optimization for network performance and data handling.

## [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 provided here represents the domain model for "Draw It or Lose It." This diagram demonstrates how the game's core components are structured and how they interact with each other.

* **Entity Class:** This base class defines shared attributes such as id and name across the Game, Team, and Player classes. This is an example of the inheritance principle in object-oriented programming, promoting code reuse and simplicity.
* **GameService Class:** This follows the Singleton design pattern, ensuring that only one instance of the service exists in memory at any given time. This fulfills the requirement that only one game instance can be active at once, efficiently managing game states and operations.
* **Game, Team, Player Classes:** These are linked in a hierarchical relationship where a Game contains multiple Teams, and each Team contains multiple Players. These classes also follow the association relationship principle, with the GameService managing collections of Game objects and coordinating interactions between the teams and players.

This object-oriented design ensures that the game can efficiently manage multiple entities while enforcing unique identifiers and minimizing redundancy.

**"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.

**Server Side**

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server-Based Deployment** | macOS can serve web apps but is uncommon in large server environments. | Linux is the most popular for server hosting due to scalability, stability, and strong community support. | Windows Server is enterprise-friendly but resource-intensive and may incur higher licensing costs. | Mobile devices are limited in server-side capabilities but can connect to the server as clients. |
| **Licensing Costs** | macOS is free for personal use, but server-side deployment requires macOS Server, with potential additional costs. | Linux is open-source and free, making it cost-effective for scaling server needs. | Windows Server requires costly licenses, increasing the total cost of ownership. | Mobile devices typically don’t host servers but may incur app store fees. |

**Client Side**

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Browser Compatibility** | Requires compatibility with Safari and Chrome. | Must support Firefox, Chromium, and potentially other browsers depending on the distribution. | Supports Edge, Chrome, and Firefox; needs to be optimized for different screen resolutions. | Requires responsive design or native apps. Must support touch interfaces and multiple screen sizes (iOS, Android). |
| **Development Complexity** | Moderate complexity due to the need for testing in Safari and handling macOS-specific UI. | High complexity due to multiple Linux distributions and browser variations. | Lower complexity: Windows has standardized browser behavior but requires screen resolution adaptation. | High complexity; Mobile platforms require either responsive design or dedicated development using React Native, Android Studio, or Xcode. |

## 

## Programming Languages and Tools

| **Development Tools** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Programming Languages** | JavaScript (React, Node.js), Swift. | JavaScript (Node.js), Python, Java, C++. | C#, JavaScript, .NET Core. | Swift for iOS, Kotlin or Java for Android, or cross-platform tools like React Native. |
| **IDEs** | Xcode, Visual Studio Code, JetBrains IDEs. | Eclipse, IntelliJ IDEA, Vim, Emacs. | Visual Studio, Visual Studio Code, JetBrains IDEs. | Xcode for iOS, Android Studio for Android, or React Native for cross-platform. |
| **Development Team Impact** | The team will need expertise in macOS development tools. | Requires Linux expertise and experience with its toolchain; familiarity with different distributions is needed. | Easier for enterprise environments, with widely available Windows development tools. | Requires dedicated mobile developers or expertise in cross-platform frameworks like React Native. |
| **Licensing Costs** | Xcode is free, but Apple Developer Program requires a subscription for app store publishing. | Linux development tools are mostly open source, with no licensing costs. | Windows development tools like Visual Studio may have licensing costs. | iOS requires Apple Developer Program; Android may have Play Store fees. |

## 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**: Based on scalability, performance, and cost-effectiveness, I recommend using Linux as the primary operating platform for server-side deployment. Linux is highly suitable for cloud environments (AWS, Azure) due to its stability, vast support, and ability to handle large user bases without incurring significant licensing fees.
2. **Operating Systems Architectures**: The architecture should follow a microservices model using containerization tools like Docker. Each component of the game, such as game state management, player authentication, and drawing rendering, can be deployed as independent services. These microservices can be scaled separately depending on load, ensuring efficient use of server resources and maximizing performance.
3. **Storage Management**: For data storage, I recommend using a NoSQL database like MongoDB, which excels in handling dynamic data such as player information, game states, and logs. MongoDB allows for flexible schema design and scales horizontally, making it ideal for large-scale applications where data structures may evolve over time.
4. **Memory Management**: Linux provides robust memory management, utilizing virtual memory and paging techniques. These features help the system efficiently manage physical memory, ensuring that memory-intensive operations like drawing rendering and real-time game updates don’t consume excessive resources. Additionally, memory allocation for each microservice can be tuned via container limits in Docker, ensuring that each service has sufficient memory for its operations without overloading the system.
5. **Distributed Systems and Networks**: The game needs to be accessible across different platforms, so a RESTful API should be employed to allow communication between the server and client applications. Using REST over HTTPS ensures that game data, player interactions, and scores can be shared across different devices (web browsers, mobile apps) efficiently. To handle connectivity issues or outages, implement load balancers and failover mechanisms that automatically redirect traffic to healthy instances in case of system failures. Also, consider content delivery networks (CDNs) for distributing static assets (images, drawings) to minimize latency.
6. **Security**: Implement SSL/TLS encryption for all communications between the client and server to protect user data during transmission. Additionally, OAuth 2.0 should be used for secure user authentication, ensuring that only authorized users can access the game and sensitive data. Regular security audits and penetration testing should be conducted to identify vulnerabilities. Ensure that proper role-based access control (RBAC) is in place, especially for administrative functions, to minimize risks.