

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 | <10/13/24> | Colten Moore | Operating System comparison outline and recommendations made |

**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 objective of this project is to develop a web-based game application called "Draw It or Lose It," inspired by the classic 1980s game show *Win, Lose or Draw*. This is designed to be a multiplayer game, allowing one or more teams to compete in guessing what is being drawn. Each game round lasts around one minute, with clues provided with the use of rendered images. The goal is to provide fun and interactive experiences for players while also making sure that the application meets crucial software requirements regarding team structure, unique identifiers, and session management.

The solution is to build a robust, scalable web application using object-oriented programming principles. This will support a structured environment where teams, players, and instances are clearly defined and efficiently managed. To meet the client’s requirements, we will develop unique identifiers for each game, team, and player and enforce uniqueness for the names. Only one active instance of the game should be allowed in memory at any time, which will be achieved through singleton design patterns for game management.

## [Design Constraints](#_2et92p0)

1. **Distributed Web-Based Environment**  
   The application is intended to be web-based, with a strong consideration for network latency, security, and scalability. As a result, the design needs to support efficient data synchronization to ensure that the game states are updated in real-time across multiple users’ screens without delays.
2. **Unique Identifiers and Name Uniqueness**  
   Unique identifiers are critical for each instance of a game, team, and player. This requirement implies that we must implement a mechanism to check and reserve names efficiently to prevent duplicate team and game names.
3. **Single Instance Constraint**  
   Only one instance of the game should be active at any time in memory, making it necessary to use a singleton pattern or a similar control structure to manage the game state centrally. This design constraint might have implications for how resources are allocated and managed on the server-side.
4. **Multiplayer Support**  
   The application needs to support multiple teams and multiple players within each team. This kind of distributed environment requires efficient session management to handle multiple concurrent users smoothly.

**Implications on Development:**  
To address these constraints, we can implement a backend infrastructure capable of handling high concurrency and real-time updates. Using more modern technologies like WebSockets or server-sent events, we can achieve minimal latency and responsive experience for all players. We can also enforce unique names and single-instance restrictions programmatically and through the database.

## 

## [Domain Model](#_8h2ehzxfam4o)

The provided UML diagram represents the core entities and their relationships within the "Draw It or Lose It" application. The diagram’s primary classes are:

1. **Game**  
   this is the central entity managing the game’s logic, rounds, and interactions. The **Game** class was designed as a singleton, making sure that only one instance can be active at any time. It also contains logic for things like starting and ending rounds, control of time limits, and rendering images for clues.
2. **Team**  
   Each **Team** object represents a group of players. It contains unique identifiers for name checking and an array or list of **Player** objects associated with it. The Team class contains functionality to manage player interactions and handle team-specific game data.
3. **Player**  
   The **Player** class holds information on individual users for the game. It stores player’s name and could potentially store scores or other individual stats.
4. **Entity (Base Class)**  
   An **Entity** base class was introduced to handle shared attributes and behaviors for common properties, such as things like unique identifiers and names. This base class promotes code reusability, reducing redundancy across the **Game**, **Team**, and **Player** classes by centralizing shared attributes and methods.

**"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** | macOS is a Unix-based system supporting web servers like Apache and Nginx, but it isn’t typically used for production environments. While it offers amazing developmental tools, like Xcode, and integration with the Apple ecosystem, its high costs and limited server-side adoption make it a less favorable option for large-scale hosting. | Linux is highly popular for web hosting due to its flexibility, scalability, and lack of licensing fees. It supports major web servers like Apache and Nginx and is favored for its strong community support. However, it requires skilled administrators and developers with Linux experience. | Windows Server is widely used in enterprise environments and integrates well with Microsoft technologies, particularly with .NET applications. Its user-friendly GUI and integration with Visual Studio make it appealing for developers, but it comes with higher licensing fees and resource demands compared to Linux. | Mobile devices do not host web applications but act as clients connecting to server-side applications. Therefore, server hosting considerations for mobile platforms are not applicable. |
| **Client Side** | To support Mac clients, the application needs to be optimized for browsers like Safari, Chrome, and Firefox. This requires compatibility with Apple-specific features, increasing development time and costs. Proficiency in HTML5 and JavaScript is critical for ensuring cross-browser functionality on macOS. | Linux desktop environments require compatibility with browsers like Firefox and Chrome. While it’s cost-effective and open-source, developing for Linux requires additional testing across different distros, adding time to the development process. | With Windows being the most widely used Operating System, the game must support major browsers like Chrome, Firefox, and Edge. Windows environments are familiar to most developers, speeding up the process, though licensing costs for tools like Visual Studio can add up fast. | Supporting both iOS and Android requires responsive web design and knowledge of mobile-specific optimization. Knowledge in Swift (iOS) and Kotlin/Java (Android) is needed, or cross-platform frameworks like Flutter can reduce development complexity. Development is time consuming due to device fragmentation and OS versioning. |
| **Development Tools** | Relevant tools include Xcode, Swift, Objective-C, and JavaScript, alongside IDEs like VS Code. Development teams need Apple-specific expertise, especially for iOS integration. While Xcode is free, macOS hardware and Apple's Developer Program fees can increase costs. | Common languages include Python, Java, JavaScript, and PHP, with tools like VS Code and Eclipse. Linux developer teams typically have strong open-source expertise, and licensing costs are minimal since most development tools are free. | Windows development typically uses C#, .NET, and JavaScript, with Visual Studio as the primary IDE. Teams with Microsoft expertise can work efficiently, but the cost of Windows Server and premium development tools may be higher. | Key tools include Android Studio for Android and Xcode for iOS, along with cross-platform frameworks like React Native. Mobile development teams must be skilled in mobile optimization, and while Android development is free, Apple’s Developer Program costs $99/year. |

## 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**: I recommend using Linux as the primary operating platform for The Gaming Room to expand *Draw It or Lose It* to other computing environments. Linux offers a flexible, scalable, and cost-efficient option, making it ideal for web-based applications that need to scale to thousands of users. It is also compatible with major web hosting services and can integrate seamlessly with other operating systems like Windows and macOS, as well as mobile platforms (iOS and Android). Additionally, Linux’s open-source nature greatly reduces licensing costs.
2. **Operating Systems Architectures**: The recommended platform will leverage the Unix-based **architecture** of Linux, which is suited towards high-traffic web applications. This architecture supports multi-user and multi-tasking operations, ensuring that multiple players can access the game simultaneously with minimal latency. Linux also allows lightweight virtual environments and containers (example - [Docker](https://www.docker.com/)) that help isolate processes, optimize resource allocation, and enhance scalability as the game grows.
3. **Storage Management**: For storage management, I would recommend using cloud-based storage solutions such as [Amazon S3](https://aws.amazon.com/s3/) or [Google Cloud Storage](https://cloud.google.com/storage/) that integrate well with Linux servers. These services provide scalable, high-availability storage with built-in redundancy and automated backups. Additionally, Linux natively supports a wide range of file systems such as EXT4 and ZFS, which offer excellent performance for handling large volumes of game data, including player records and multimedia assets.
4. **Memory Management**: Linux uses advanced memory management techniques such as virtual memory, paging, and memory swapping. It efficiently manages both physical and virtual memory, ensuring that critical processes like rendering game graphics and processing user inputs are prioritized. The operating system’s ability to allocate memory dynamically to processes based on need allows *Draw It or Lose It* to run smoothly even during peak traffic. Additionally, Linux's memory management reduces the likelihood of memory leaks, helping maintain long-term performance.
5. **Distributed Systems and Networks**: To enable *Draw It or Lose It* to communicate across platforms, I recommend deploying the game using distributed systems architectures like microservices hosted in cloud environments ([AWS](https://aws.amazon.com/s3/), [Azure](https://portal.azure.com)). Each service (such as rendering, user authentication, or puzzle management) can operate independently, enhancing the system's flexibility and fault tolerance. A thorough network architecture, supported by [RESTful APIs](https://aws.amazon.com/what-is/restful-api/) or WebSockets, will enable real-time communication between the server and various client devices (desktop and mobile). Load balancers will manage traffic and ensure seamless gameplay, even in case of network outages or device disconnections.
6. **Security**: Security must be a priority for protecting user data on and between various platforms. Linux offers strong built-in security features, including access control via [SELinux](https://www.redhat.com/en/topics/linux/what-is-selinux) and [AppArmor](https://www.apparmor.net/). I recommend implementing SSL/TLS encryption for all communication between the game server and client devices, ensuring that user data is protected from interception. [OAuth2](https://oauth.net) or [JWT](https://jwt.io) can be used for secure user authentication across devices, while firewalls (such as iptables on Linux) can protect the game servers from unauthorized access. Additionally, data stored in cloud services should be encrypted both at rest and in transit.