

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 | 09/22/2024 | Elizabeth Walko | Initial write up |

**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 seeks to extend their existing Android-based game, *Draw It or Lose It*, into a web-based platform capable of serving multiple users across various devices. The game must support multiple teams and players, ensure unique names for teams and games, and limit game instances to one in memory at any given time. To meet these requirements, we propose a client-server architecture that allows for scalable, real-time interaction while maintaining uniqueness and game integrity. Our solution includes a backend that enforces unique identifiers for games, teams, and players, and a frontend that provides a smooth cross-platform experience. By following this approach, we can facilitate the development of a flexible, scalable, and secure gaming platform that meets the client’s expectations.

## Requirements

* *A game will have the ability to have one or more teams involved.*
* *Each team will have multiple players assigned to it.*
* *Game and team names must be unique to allow users to check whether a name is in use when choosing a team name.*
* *Only one instance of the game can exist in memory at any given time. This can be accomplished by creating unique identifiers for each instance of a game, team, or player.*

## [Design Constraints](#_2et92p0)

Developing this game application in a web-based distributed environment introduces several design constraints:

1. **Concurrency & Synchronization**: Since multiple players and teams will interact with the game simultaneously, the system must efficiently handle real-time updates and synchronization of game state across platforms. This requires the use of real-time technologies like WebSockets or server-sent events to ensure low-latency communication.
2. **Scalability**: The web application should be scalable to support a growing number of players and games. This constraint affects the choice of database (e.g., choosing a scalable NoSQL or SQL solution), server architecture (cloud-based deployment), and design patterns (microservices, load balancing).
3. **Uniqueness & Identifier Management**: The need for unique names and identifiers for games, teams, and players requires strict enforcement at the database level. This impacts database schema design and query efficiency, as the system must quickly validate uniqueness in real time.
4. **Session Management**: Since only one instance of the game can be in memory at a time, session management is critical. The system must ensure that game sessions are properly tracked and handled to prevent multiple active instances.

The implications of these constraints are that the system must be built with performance, concurrency, and scalability in mind. These design considerations will dictate the choice of technologies and impact the development process.

## [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 for *The Gaming Room* system showcases a structure that efficiently supports the development of the game *Draw It or Lose It*. At the core is the Entity class, a superclass for Game, Team, and Player, which ensures that each game element has a unique id and name. This design allows for shared functionality and code reuse through inheritance. The GameService class is responsible for managing games and teams, using the singleton pattern to ensure only one game instance exists in memory, fulfilling the client’s requirement for game uniqueness.

The relationships between classes are primarily composed of inheritance and composition. A Game contains multiple Team objects, and each Team manages a list of Player objects. This allows for multiple teams and players to participate in a single game instance, supporting flexible and scalable gameplay. The singleton design pattern in GameService ensures that game state management is centralized, preventing multiple games from running simultaneously.

Object-oriented principles such as encapsulation, inheritance, and abstraction are leveraged to create a modular and efficient system. Encapsulation ensures that class data is protected, while inheritance promotes code reuse by sharing common properties like id and name. By using these principles, the system can efficiently meet the software requirements of unique game names, multiple teams, and a single active game instance.

**"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 is a viable option for hosting web-based applications, offering a stable Unix-based environment similar to Linux. However, macOS is not widely used in server environments, and hosting options are limited compared to Linux. Macs are often more expensive, and while they support standard web technologies (Apache, Nginx, Node.js), they are less optimized for large-scale web servers. | Linux is the most widely used operating system for hosting web-based applications due to its open-source nature, scalability, and flexibility. Linux distributions (such as Ubuntu, CentOS, or Debian) are highly optimized for server environments, offering excellent performance, security, and compatibility with most web technologies. It’s the most cost-effective platform and is ideal for scalable and cloud-based solutions. | Windows is a strong contender for hosting applications, particularly when using Microsoft technologies such as ASP.NET or IIS. It provides excellent integration with enterprise solutions and tools like SQL Server. However, it tends to be more resource-heavy and expensive, especially for larger deployments. Windows Server versions are commonly used, but may require more management and licensing costs compared to Linux. | Mobile devices are not typically used to host server-side applications. They lack the processing power, storage, and network capabilities necessary for efficient server hosting. However, they can act as thin clients or interface with cloud-based servers, enabling mobile access to web-based applications. Hosting on mobile devices would not be suitable for a production environment. |
| **Client Side** | Developing for macOS clients requires expertise in Swift or Objective-C for native applications, as well as knowledge of cross-platform frameworks like React Native or Electron for hybrid apps. macOS is typically well-suited for creative and high-performance applications but comes with a higher cost of development due to niche expertise. Supporting macOS is necessary for companies targeting creative or tech-focused user bases. | Linux has a lower user base for desktop applications, but its open-source nature allows for flexible development. Applications for Linux clients can be built using tools like GTK, Qt, or Java. Linux requires more expertise due to the diversity of distributions and user environments. However, supporting Linux is essential for server administrators and developers who prefer open-source software. | Windows has the largest market share among desktop OSes, making it essential to support. Developing for Windows clients is cost-effective due to a large talent pool familiar with C#, .NET, and other Microsoft technologies. Tools like Visual Studio offer robust development environments, but Windows applications can be resource-heavy and have higher hardware requirements compared to other platforms. | Developing for mobile devices requires understanding of both iOS and Android ecosystems. Native development (Swift for iOS, Kotlin/Java for Android) or cross-platform solutions like Flutter or React Native are crucial for supporting the diverse mobile market. Mobile apps need to be optimized for battery, memory, and network constraints, adding complexity and testing requirements. However, mobile devices offer widespread reach and are critical for user engagement. |
| **Development Tools** | macOS developers typically use Xcode for building native applications. For web-based applications, standard tools like VS Code, Git, and Docker are widely available. macOS supports all major programming languages (e.g., JavaScript, Python, Ruby) and development frameworks. The platform’s Unix base makes it a good choice for development but less so for deployment due to its cost and limited server adoption. | Linux offers a vast range of development tools, both native and third-party. Popular IDEs like VS Code, Eclipse, and PyCharm are commonly used. Linux is ideal for backend and server-side development due to its compatibility with programming languages like Python, PHP, JavaScript, and its support for containers (Docker, Kubernetes). The open-source nature and flexibility of Linux make it a top choice for web-based application development. | Windows development is driven by Microsoft’s ecosystem, particularly Visual Studio and .NET. Developers working in C#, ASP.NET, or other Microsoft technologies find Windows to be the most compatible environment. Windows also supports JavaScript, Python, and other web technologies, making it flexible for full-stack development. However, development tools may require more system resources compared to Linux. | Mobile app development requires platform-specific tools. Xcode is used for iOS development, and Android Studio for Android development. Cross-platform tools like React Native, Flutter, and Xamarin enable simultaneous development for both platforms. Additionally, tools like Figma for design and Firebase for backend integration are widely used in mobile development. These tools optimize app creation but introduce complexity due to fragmentation in OS versions and device types. |

## 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 **Linux** as the primary operating platform for *The Gaming Room* to expand *Draw It or Lose It* to other computing environments. Linux is highly scalable, open-source, and compatible with a variety of hardware. It supports a wide range of software stacks, making it ideal for developing web-based applications that need to run across multiple platforms, including mobile, desktop, and cloud environments. Additionally, its cross-platform support allows easy adaptation for other environments such as Windows and macOS through containerization and cloud technologies like Docker and Kubernetes.
2. **Operating Systems Architectures**: Linux utilizes a modular architecture that allows for significant flexibility and control over system resources. It follows a monolithic kernel design but supports a variety of architectures, including x86, ARM, and more. For *The Gaming Room*, this allows for adaptability whether the game is running on servers, desktops, or mobile devices. Linux's POSIX-compliant architecture is highly efficient in managing processes, memory, and storage, and the platform’s support for multi-threading and asynchronous I/O makes it ideal for real-time gaming environments. Moreover, Linux distributions offer stability and security, which is critical for a web-based game.
3. **Storage Management**: For storage management, I recommend using a **cloud-based solution** like **Amazon S3** or **Google Cloud Storage** in conjunction with **Linux file systems** such as EXT4 or XFS. These storage systems are scalable, ensuring that as *Draw It or Lose It* grows, the storage infrastructure can scale accordingly. Cloud storage provides robust data redundancy, backups, and disaster recovery mechanisms. Additionally, leveraging Linux’s built-in file systems allows for efficient management of local data and cached files, while cloud storage can serve as the main repository for user data, game assets, and session information.
4. **Memory Management**: Linux employs several advanced memory management techniques, such as **virtual memory**, **paging**, and **swapping**. These allow for efficient handling of memory resources, even when the physical RAM is limited. The use of memory caches and buffers ensures fast access to frequently used game assets, minimizing latency during gameplay. For *Draw It or Lose It*, this memory management is crucial, as the game must handle multiple players and teams, loading and rendering images in real-time. The use of memory paging allows the system to handle a high number of concurrent users without overburdening physical memory, maintaining performance during peak usage.
5. **Distributed Systems and Networks**: To enable communication between platforms, *Draw It or Lose It* can be deployed as a **distributed system** using **microservices** and containerized applications (Docker). Distributed systems would allow different components of the game—such as user authentication, game logic, and image rendering—to run independently, ensuring scalability and resilience. The system can be deployed across cloud servers (AWS, Google Cloud, or Azure) and can use **RESTful APIs** or **WebSockets** to enable real-time communication between clients (mobile, web, desktop). This setup reduces dependencies between components, meaning the game can stay online even if one service fails. Utilizing **CDNs** (Content Delivery Networks) can further improve game performance by reducing latency for users across various geographical locations.
6. **Security**: Security is critical for *The Gaming Room*, particularly with sensitive user information and cross-platform communication. Linux offers strong built-in security features, such as **SELinux** (Security-Enhanced Linux) and **AppArmor**, which enforce mandatory access controls to protect system resources and user data. Additionally, securing the network through **TLS/SSL** encryption ensures that all data transferred between users and servers is protected. For cross-platform communication, using **OAuth 2.0** for secure authentication and **JWT** (JSON Web Tokens) for session management will protect user data across mobile, desktop, and web clients. Regular patching, firewalls, and network intrusion detection systems (NIDS) should be employed to mitigate vulnerabilities and protect against cyber-attacks.