s

<The Gaming Room >

# **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 | <01/30/25> | <Christopher Duclervil > | <Brief description of changes in this revision> |

**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 aims to develop a web-based version of their game application, 'Draw It or Lose It.' The primary goal is to create a scalable, distributed, and platform-independent solution that ensures smooth gameplay across multiple environments. This document outlines the software design problem and provides a structured solution to meet business and technical requirements. Key considerations include the selection of an appropriate development stack, performance optimization, security measures, and compatibility with various operating systems. By leveraging modern web technologies and cloud-based infrastructure, we propose a solution that ensures accessibility, reliability, and efficient resource utilization while maintaining a seamless gaming experience for users.

## Requirements

The Gaming Room requires a web-based implementation of their game 'Draw It or Lose It,' designed to function efficiently in a distributed environment. The application must meet both business and technical requirements to ensure seamless gameplay and scalability.  
  
**Business Requirements:**   
1. The game must support multiple teams competing in a session.  
2. Each team should consist of multiple players, ensuring collaborative gameplay.  
3. Game and team names must be unique, preventing duplication and confusion.  
4. A central game instance must manage game flow, ensuring a consistent experience for all players.  
5. The game should be accessible from various platforms, including desktops, tablets, and mobile devices.  
  
**Technical Requirements**  
1. The application must maintain a single instance of the game in memory to prevent conflicts and redundancy.  
2. Unique identifiers must be generated for each game, team, and player to avoid duplication.  
3. The system should efficiently render and display drawings from a pre-existing library of images.  
4. The game must include a real-time mechanism for user interaction, such as WebSockets, to handle guesses and responses efficiently.  
5. The backend should be developed with scalability in mind, using a cloud-based infrastructure.  
6. Security features such as user authentication and data encryption must be incorporated.  
7. Performance optimizations should be implemented to ensure smooth gameplay, particularly in managing time-based game rounds.  
  
Meeting these requirements will ensure that 'Draw It or Lose It' provides an engaging and seamless experience for users across different devices and platforms.

## [Design Constraints](#_2et92p0)

Developing a game application in a web-based distributed environment presents several design constraints that must be addressed:  
  
1. **Scalability:** The application must efficiently handle multiple concurrent users without performance degradation. Implementing a scalable backend and load-balancing mechanisms is crucial.  
  
2. **Cross-Platform Compatibility**: The game should be accessible from different operating systems and devices, including Windows, macOS, Linux, and mobile platforms. This requires adopting web technologies such as HTML5, JavaScript, and responsive UI design.  
  
3. **Latency and Performance**: Real-time interactions in the game demand low latency and optimized network communication. The use of WebSockets or similar real-time communication protocols is necessary.  
  
4.  **Security:** Protecting user data and ensuring secure communication between clients and the server is a priority. Implementing authentication, authorization, and data encryption mechanisms is essential.  
  
5. **Data Consistency and Synchronization:** Given that the game state is shared across multiple users, data consistency should be maintained through reliable database management and synchronization strategies.  
  
6. **Cloud Infrastructure and Cost Efficiency**: Hosting the game on cloud platforms introduces cost constraints and requires strategic resource allocation to optimize expenses while maintaining high availability.

## [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)

This UML diagram effectively represents The Gaming Room system, providing a clear structure for how the different components interact. By applying object-oriented principles such as **inheritance, encapsulation, and polymorphism**, This design promotes reusability and maintainability. The relationships between `**Game`, `Team`, and `Player`** allow for a scalable and logical approach to managing game instances. Also implementing the singleton pattern in **`GameService**` ensures that game management remains centralized and efficient.

**"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** | **Pros:** Stable Unix-based system, secure, good for local testing.    **Cons:** Expensive, limited enterprise hosting support, less flexible than Linux. | **Pros:** Preferred for web hosting, highly stable, secure, scalable, cost-effective.   **Cons:** Requires technical expertise for setup and management. | **Pros:** Strong support for .NET applications, good for enterprise use.   **Cons:** Costly licensing, resource-intensive, less scalable than Linux. | **Not suitable** for web hosting due to limited processing power and network stability. Best used as a **client platform** accessing a cloud-hosted service. |
| **Client Side** | **Pros:** Smooth UI experience, macOS development supported via Swift, Objective-C.  **Cons:** Requires macOS hardware, increasing costs. Cross-platform tools (Electron, React Native) offer alternatives. | **Pros:** Free licensing, strong open-source community support.   **Cons:** Compatibility issues across different Linux distributions, requires specialized expertise. | **Pros:** Supports various frameworks like .NET, Electron, and web-based apps.   **Cons:** Requires expertise in C#, JavaScript, or cross-platform tools. Compatibility issues across Windows versions. | **Pros:** Can be developed using native (Swift, Kotlin) or cross-platform tools (Flutter, React Native).   **Cons:** Native development is expensive and time-consuming; cross-platform tools may have performance trade-offs. |
| **Development Tools** | **Languages:** Swift, Objective-C, JavaScript, Python, Node.js.  **Tools:** Xcode, Visual Studio Code, Electron. | **Languages:** Python, Java, JavaScript, PHP, C++.  **Tools:** VS Code, JetBrains IDEs, Apache/Nginx for hosting. | **Languages:** C#, .NET, Java, JavaScript, Python.  **Tools:** Visual Studio, JetBrains IDEs, Eclipse. | **Languages:** Swift (iOS), Kotlin/Java (Android), JavaScript (React Native, Flutter).  **Tools:** Android Studio, Xcode, Flutter, React Native. |

## 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**: Linux is the most suitable server platform for *Draw It or Lose It* due to its stability, scalability, and cost-effectiveness. Unlike Windows and macOS, Linux provides a robust open-source environment with extensive community support and enterprise-level security features. It is widely used in cloud computing and offers better customization for web hosting. Deploying the game on Linux-based cloud servers like AWS EC2, Google Cloud Compute Engine, or Azure Virtual Machines ensures high availability and efficient resource management.
2. **Operating Systems Architectures:** A microservices-based distributed architecture is recommended to optimize performance, maintainability, and scalability. Instead of a monolithic structure, microservices allow independent deployment of game features such as authentication, real-time gameplay, and image processing. Containerization using Docker and Kubernetes will further streamline deployment and scaling, allowing for efficient resource allocation and fault tolerance.
3. **Storage Management**: Given that the game requires storing 200 high-definition images, totaling approximately 1.6GB, an optimized storage solution is critical. A hybrid storage approach is recommended, combining cloud-based solutions such as Amazon S3, Google Cloud Storage, or Azure Blob Storage with caching strategies using Content Delivery Networks (CDNs) like Cloudflare or AWS CloudFront. This will enable rapid retrieval of images while reducing latency and storage costs

**.**

1. **Memory Management**: Efficient memory allocation is essential to ensure smooth gameplay performance. Linux uses advanced memory management techniques such as virtual memory, paging, and buffer caching to optimize performance. Additionally, in-game assets can be managed using caching solutions like Redis or Memcached to reduce the frequency of database calls and optimize response times. Implementing proper garbage collection and dynamic memory allocation will also prevent memory leaks and ensure efficient resource utilization.
2. **Distributed Systems and Networks**: To enable seamless cross-platform communication, a distributed architecture leveraging RESTful APIs or WebSockets is recommended. Cloud-based load balancers, such as AWS Elastic Load Balancing or Google Cloud Load Balancer, will distribute traffic evenly, preventing bottlenecks and improving response times. Deploying redundant game servers across multiple geographic locations will minimize downtime and ensure high availability. Network monitoring tools like Prometheus or Datadog will help detect and resolve potential connectivity issues proactively.
3. **Security**: Security is a top priority to protect user data and prevent cyber threats. Implementing OAuth 2.0 authentication and TLS encryption will secure user credentials and data transmission. Role-based access control (RBAC) will restrict unauthorized access to game resources. Regular vulnerability assessments, firewalls, and intrusion detection systems (IDS) should be implemented to monitor and prevent attacks. Additionally, secure API endpoints and encrypted database storage will further protect sensitive user information.

**Summary**:

For server hosting, I believe Linux is the best choice because of its scalability, cost-effectiveness, and strong track record in web hosting. It’s stable, secure, and widely used for high-traffic applications, plus the availability of open-source tools helps keep costs down. On the client side, I prefer a web-based approach using HTML5, JavaScript, and responsive design to ensure the game runs smoothly across macOS, Windows, Linux, and mobile devices without platform-specific restrictions.

When it comes to development, I recognize that Windows and macOS require licensedtools for native applications, whereas Linux offers free, open-source options, making it a more budget-friendly choice. To simplify development across multiple platforms, I would use cross**-**platform frameworks like Flutter or React Native, which help reduce effort while maintaining broad compatibility. While native development in Swift (iOS) and Kotlin (Android) delivers the best performance, it’s more time-consuming and expensive, so I would consider cross-platform tools to balance cost and efficiency.