

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 | 08/17/2025 | Erica Boterf | Initial version of software design |

**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 currently offers the game *Draw It or Lose It* as an Android application. They would like to expand the game to a web-based platform that can run seamlessly across multiple operating systems and devices. This shift will allow the game to reach a wider audience and provide consistent gameplay experiences regardless of the user’s device.

To achieve this, we will develop a web-based distributed application using industry-standard software development practices. The solution will ensure that:

* The game supports multiple teams and players.
* Game and team names are unique for easier identification and user experience.
* Only one active game instance exists in memory at a time, using the Singleton design pattern.
* Efficient management of players and teams through the use of object-oriented principles such as inheritance and encapsulation.

This document outlines the design constraints, domain model, platform evaluations, and recommendations to deliver a robust, scalable, and secure solution.

## Requirements

*The Gaming Room’s primary business and technical requirements include:*

* *Ability to support multiple teams and multiple players per team.*
* *Unique game and team names for clear identification.*
* *Restrict the system to one game instance in memory at any given time.*
* *Ensure compatibility across different operating systems and devices.*
* *Provide secure communication and user data protection.*
* *Enable future scalability for additional features and game modes.*

## [Design Constraints](#_2et92p0)

Developing a web-based, distributed application introduces several constraints:

* **Platform Independence:** The solution must run on multiple operating systems (Windows, Linux, macOS) and mobile devices. This requires web technologies like HTML5, CSS, JavaScript, and a back-end built on a platform-independent framework.
* **Scalability:** The application must support multiple concurrent users without performance issues. This requires load balancing and a scalable server environment.
* **Security:** User data must be protected through encryption and secure APIs. Since the application communicates across networks, data-in-transit security is critical.
* **Latency and Reliability:** Distributed environments can experience network delays or outages. The design should minimize latency and handle disconnections gracefully.
* **Unique Identifiers:** All game, team, and player names must be unique. Implementing the Iterator pattern for searching names before creation will prevent conflicts.
* **Single Instance Limitation:** The Singleton pattern will be implemented for the GameService class to ensure only one game instance exists in memory.

## [System Architecture View](#_ilbxbyevv6b6)

The system will use a **web-based, client-server architecture**. This approach allows players on different platforms—desktop (Windows, macOS, Linux) and mobile devices—to access the game through a web browser without requiring additional installations.

The architecture consists of the following layers:

* **Client Layer (Front-End):**  
  Runs on the user’s device through a standard web browser. It handles user interactions and communicates with the server using HTTPS. The front-end will be built using standard web technologies such as HTML5, CSS, and JavaScript, ensuring cross-platform compatibility.
* **Application Layer (Server-Side):**  
  Deployed on a Linux-based environment for reliability and cost-effectiveness. The server will manage game logic, enforce business rules (e.g., unique names, single game instance), and process all requests from clients. The Singleton pattern will ensure only one game instance runs in memory.
* **Data Layer (Database):**  
  A cloud-hosted relational database (e.g., MySQL or PostgreSQL) will store persistent data such as player information, team details, and game history. This enables scalability and high availability.
* **Communication:**  
  The system uses RESTful APIs over HTTPS to exchange data between clients and the server. This ensures secure and efficient communication.

This layered design promotes scalability, maintainability, and platform independence, making it an ideal choice for a distributed, web-based game application.

## [Domain Model](#_8h2ehzxfam4o)

The UML diagram illustrates the relationships among the core classes:

* **Entity Class (Base Class):** Holds common attributes like id and name. This promotes code reuse and simplifies maintenance.
* **Game Class:** Inherits from Entity. Represents a game instance and maintains a list of teams.
* **Team Class:** Inherits from Entity. Holds a collection of players.
* **Player Class:** Inherits from Entity. Represents individual participants.
* **Inheritance:** Shared attributes are centralized in the Entity class to avoid redundancy.
* **Encapsulation:** Class fields are private, and public getters/setters control access.
* **Polymorphism:** All classes can be handled through the base Entity type when needed.
* **Abstraction:** The design hides implementation details from users while exposing only necessary functionality.

These principles ensure efficiency, maintainability, and scalability in fulfilling client requirements.

**"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 can technically host web applications, but it's not commonly used for large-scale deployments. It’s better suited for development or small-scale testing. While secure and developer-friendly, the hardware cost is high and enterprise support is limited. | Linux is the top choice for server hosting. It’s stable, scalable, open-source, and widely used in the cloud (e.g., AWS, Azure). Perfect for RESTful API deployments with strong community and tool support. | Windows Server is a reliable hosting platform with strong support for .NET and IIS. However, it comes with high licensing costs and is less favored for open-source environments. | Mobile devices are not used to host the application. Instead, they will act as clients communicating with the central server. |
| **Client Side** | Requires ensuring the app runs smoothly in Safari and matches macOS users' expectations for visual quality. A responsive, polished UI is important here. | Most Linux users will access the game through modern browsers like Firefox or Chrome. As long as standard web practices are followed, compatibility is not an issue. | Must ensure compatibility with Edge, Chrome, and Firefox. A responsive web design approach ensures functionality across desktop resolutions. | The design must be responsive and optimized for smaller screens and touch input. Use media queries, lightweight assets, and test thoroughly on Android and iOS. |
| **Development Tools** | Tools like Xcode (for iOS), IntelliJ, and VS Code are common. Languages include Java, JavaScript, HTML, and CSS. macOS is required for iOS testing. | Tools like Eclipse, VS Code, and IntelliJ work well. Languages include Java, Python, JavaScript, HTML, and CSS. Ideal for building full-stack web apps. | Visual Studio is a top choice on Windows, especially for C# and .NET development. JavaScript, HTML, and CSS are also supported. | Use Android Studio for Android, Xcode for iOS, or cross-platform tools like Flutter and React Native. JavaScript frameworks are ideal for responsive browser apps. |

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 the server side, Linux is the best choice. It’s reliable, scalable, and doesn’t come with heavy licensing costs. It’s the standard for many web and distributed applications, and it works smoothly with cloud providers like AWS, Azure, and Google Cloud. If the game suddenly grows in popularity, Linux can scale without expensive or complicated upgrades. Plus, its open-source community provides plenty of resources and support for troubleshooting and improvements.
2. **Operating Systems Architectures**: A 64-bit Linux setup will give us the performance and stability we need. It takes advantage of modern multi-core processors, handles large amounts of data efficiently, and is compatible with most frameworks we might use now or in the future. It’s a solid choice that won’t hold us back as the game evolves.
3. **Storage Management**: We’ll use a cloud-based relational database, like PostgreSQL or MySQL, to store everything from player accounts to game histories. These databases can grow with demand, automatically back up data, and offer quick search capabilities to keep the game running smoothly. Hosting in the cloud also makes it easier to keep everything available and secure.
4. **Memory Management**: Linux does a great job of managing memory through techniques like virtual memory, caching, and swapping. This means it can keep the game responsive even when a lot of players are active at once. It also helps prevent performance drops by making sure resources are used efficiently and not wasted.
5. **Distributed Systems and Networks:** Draw It or Lose It will use a client-server model, so players can connect from any device through their browser. All communication will go through secure RESTful APIs over HTTPS. Load balancers will spread traffic evenly across servers, and backup systems will kick in if something goes wrong, helping keep the game online without interruptions.
6. Security:

Security is a core consideration for protecting user data and system integrity. The following measures will be implemented:

* **Encryption: All data moving between players and the server will be encrypted with SSL/TLS.**
* **Authentication: We’ll use secure hashing (like bcrypt) and token-based methods to keep accounts safe.**
* **Data Protection: Sensitive information will be encrypted in the database.**
* **Network Safety: Firewalls and monitoring systems will block threats and track suspicious activity.**
* **Ongoing Monitoring: Continuous logging will help us quickly spot and respond to problems.**