

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 |
| --- | --- | --- | --- |
| 3.0 | 10/14/25 | Matthew Garcia Nieves | Added Project Three recommendations covering system architecture, storage, memory management, distributed systems and security. |

**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 wants to take their existing Android-only game, *Draw It or Lose It*, and make it available as a web-based application that works across different platforms. To do this, we need a design that can support games, teams, and players in a way that keeps everything unique and organized.

In my solution, I used object-oriented programming principles and some design patterns to make the code efficient and easier to maintain. For example, a base class called Entity holds common data like the ID and name, which all other classes (Game, Team, Player) extend. This reduces duplicate code. I also used the singleton pattern in the GameService class to make sure there’s only one instance running that controls games and IDs. On top of that, the iterator pattern is used whenever we add or search for games, teams, or players, so names stay unique and no duplicates sneak in.

Overall, the design keeps the system simple but expandable, meeting the client’s need for a solid starting point to bring their game to the web.

## Requirements

From the clients request and rubric these are the key requirements:

* Only one instance of the GameService exists in memory (singleton pattern).
* System must allow one or more teams per game.
* Each team needs to allow multiple players.
* Game and team names must be unique.
* IDs must be unique across games, teams, and players.
* Shared attributes (id and name) should be reused across classes (inheritance).

## [Design Constraints](#_2et92p0)

Because this will be a web-based distributed environment there are some design limits we need to consider. The system must:

* Handle multiple users at once without creating duplicate data. This is why we implemented the singleton pattern for GameService.
* Make sure IDs and names stay unique, we handle this with iterator checks.
* Be able to run in different environments (Windows, Mac, Linux, and mobile browsers).
* Be easy to maintain and expand in the future as needed.

## [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 diagram shows how the main classes are related. Entity is the base class with shared fields like id and name, and Game, Team, and Player, all extend it so they inherit those fields instead of repeating them. A Game contains a list of teams, and each Team contains a list of players, while the GameService singleton manages all these objects and provides the methods to add or get them. This setup uses object-oriented principles like inheritance (Game, Team, and Player extending Entity), encapsulation (fields kept private with getters), abstraction (the base Entity class avoiding duplicate code), and polymorphism (the toString method adjusting output by class type). Overall this setup make Ids and names stay unique, avoids duplicate data and keeps the code easier to work with going forward.

**"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 run web servers like Apache or Nginx, but it’s rarely used for production hosting. Most hosting providers don’t offer macOS as a server option, which makes it less practical. Apple’s licensing also makes it expensive for large-scale deployments. It’s reliable but better suited for local testing rather than live web hosting. | Linux is by far the most common server OS for web applications. It’s open-source, stable, and supports all major web technologies (Java, Python, Node.js, etc.). It scales easily for distributed systems, and there are no licensing fees, which makes it cost-effective for The Gaming Room. Linux servers also handle concurrent user loads efficiently and offer built-in security tools. There are no OS licensing cost since Linux is open source. | Windows servers can host a web app through IIS and integrates well with Microsoft ecosystems, which is helpful if future versions use .NET or C#. However, Windows Server licensing can get expensive, and it typically requires more resources to run smoothly compared to Linux. | Mobile devices don’t serve as hosts, but they connect as clients. For testing purposes, small local servers can be run through emulators, but in production, mobile devices will always rely on the main web server. Licensing cost won’t apply here since mobile devices aren’t used as servers. |
| **Client Side** | Macs aren’t a huge chunk of the desktop market, but Safari support is still essential for compatibility. Development and testing on macOS can be pricey since you need Apple hardware and Xcode. Still, making sure Draw It or Lose It runs well on Safari helps reach Apple users. | Linux users can access the game though any modern browser like Firefox or Chrome. The development effort for Linux clients are minimal since most web apps already work across Linux browsers without additional coding. | Windows will likely be the most common client environment. The game needs to be tested across browsers like Chrome, Edge and Firefox to ensure performance and responsiveness. Supporting Windows is critical because it covers the widest audience. | Mobile platforms (Android and iOS) are key for this project. The app must scale properly to fit smaller screens and handle touch input smoothly. Using responsive web design (HTML5, CSS3, JavaScript) ensures the same experience across devices. Cross-platform frameworks like Flutter or React Native can also be used for hybrid development if a native mobile app is added later. |
| **Development Tools** | macOS developers typically use Xcode, IntelliJ IDEA, or Eclipse. Common languages include Swift for local apps and JavaScript for browser-based work. Apple’s ecosystem is polished but tightly controlled, so developer licensing adds cost. | Linux supports a wide range of free development tools like VS Code, Eclipse, and IntelliJ IDEA. It works with Java, Python, and JavaScript, languages that the Draw It or Lose It project already uses. It’s ideal for both server and client development with minimal cost. | Windows developers rely on Visual Studio, Eclipse, or IntelliJ. Common languages are Java, C#, and JavaScript. Windows has strong debugging tools but comes with higher licensing and system maintenance costs. | Mobile development uses Android Studio (Java/Kotlin) and Xcode (Swift) or frameworks like Flutter or React Native for hybrid apps. These tools allow developers to test across multiple screen sizes and OS versions. Apple’s App Store requires a paid developer account, while Android is more flexible and open. |

To keep one codebase working across desktop (Linux, macOS, Windows) and mobile (iOS/Android), we’ll ship a responsive, standards-based web app: semantic HTML5, modern CSS (flex/grid), and JavaScript/TypeScript. We’ll test on these major browsers (Chrome, Edge, Safari, Firefox), use a mobile-first layout, larger touch targets, lazy-load assets, and handle variable network conditions (timeouts/retries). Accessibility (keyboard focus, ARIA where needed) keeps the UI reliable across devices. This approach keeps cost/time lower than building separate native apps and avoids needing multiple dev teams.

**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**:

The best choice for hosting *Draw it or Lose It* is still Linux, especially on a cloud-based platform like AWS or Google Cloud. Linux is open source, so there is no licensing cost, and it’s known for being stable and secure. It also supports most modern web technologies developers use today, which makes it easier to build, deploy, and scale as more players join.

Other systems like Windows Server or macOS could work, but they add more cost and don’t offer the same flexibility or scalability. For this project, Linux gives the right balance of performance, security, and affordability while supporting the needs of a web-based multiplayer game.

1. **Operating Systems Architectures**:

Linux uses a modular structure that fits perfectly for a web application setup. The operating system separates the hardware control handled by the kernel from the user processes that run the application. This helps keep the system stable and organized, even with heavy traffic.

For *Draw It or Lose It*, a client server model is the best fit. The server will handle the main logic, game sessions, and data, while clients (players on desktops or mobile browsers) connect to it to play. This setup keeps the game synchronized and allows multiple servers to share the workload through load balancing if needed. It also makes updates easier since the main system is maintained in one place rather than on each device.

1. **Storage Management**.

The best storage setup for this project combines a relational database for structured data (like games, teams, and players) with cloud storage for game assets such as images or media. A database like PostgreSQL or MySQL works well because it handles relationships and constraints, keeping all records unique and organized.

Cloud-based storage (like AWS S3 or Azure Blob) is ideal for larger files and makes it easier to deliver images or content quickly to users anywhere. Using automatic backups and region replication also protects data in case of server failures or outages, ensuring that no player progress or information is lost.

1. **Memory Management**:

Linux handles memory management efficiently by using both physical memory and virtual memory when needed. It allocates memory dynamically, freeing up space as processes end or when data is no longer needed. This helps prevent slowdowns and crashes, especially when many players are active at once.

In a cloud setup, tools like Kubernetes or AWS Elastic Beanstalk can automatically scale memory resources based on demand. That means if the game sees a sudden increase in players, more memory and processing power can be added automatically, then reduced again when traffic slows down. This keeps the game smooth for players while avoiding wasted resources.

1. **Distributed Systems and Networks**:

Because*Draw It or Lose It* will be accessed from different devices and locations, a distributed system is the best approach. The application will run on multiple servers connected through a reliable network. Each server can handle different parts of the workload so that performance stays consistent even when traffic spikes.

All communication between the servers and clients will happen through RESTful APIs over HTTPS, which ensures consistent and secure data exchange. Using load balancers keeps the system from being overwhelmed by too many requests on one server. If one server goes down, traffic can automatically shift to another so that gameplay isn’t interrupted.

This setup also helps reduce lag since users connect to the closest available server, and it keeps the experience stable even during temporary connection issues.

1. **Security**:

Security must be a priority for protecting player information and keeping the system safe. Every connection between players and the game’s servers should use HTTPS with TLS encryption so that data is always secure in transit.

Users will log in with credentials tied to secure tokens (like short-lived session tokens or JWTs), which expire regularly to prevent unauthorized access. Passwords and sensitive data will be encrypted in storage using strong encryption standards such as AES-256.

Access to administrative features should be limited to authorized users only, using role-based access control. The system should also track all administrative actions, and failed login attempts for review. Regular updates and security testing will help protect against new threats. By following OWASP and NIST guidelines, the system stays aligned with modern security standards.

**References**

Amazon Web Services. (2023). *Best practices for deploying web applications on AWS.* Amazon Web Services, Inc. <https://docs.aws.amazon.com/>

Google Cloud. (2024). *Distributed systems design and reliability.* Google LLC.  
<https://cloud.google.com/architecture>

National Institute of Standards and Technology. (2023). *Zero Trust Architecture (NIST Special Publication 800-207).* U.S. Department of Commerce. <https://doi.org/10.6028/NIST.SP.800-207>

Open Web Application Security Project. (2023). *REST security cheat sheet.* OWASP Foundation. <https://owasp.org/www-project-cheat-sheets/>

Red Hat. (2023). *Linux advantages for enterprise web applications.* Red Hat, Inc. <https://www.redhat.com/>

Silberschatz, A., Galvin, P. B., & Gagne, G. (2020). *Operating system concepts* (10th ed.). Wiley.