

**Draw It or Lose It**

**CS 230 Project Software Design Template**  
Version 2.0

## 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 | 12/15/24 | Gabriela Pastor | Recommendations were edited. |

## [**Executive Summary**](#_sbfa50wo7nsh)

The Gaming Room wants to take its game, **Draw It or Lose It**, to the next level by making it web-based and accessible across multiple platforms. This means transitioning from the current Android-only app to something more flexible and scalable. Our goal is to create a seamless gaming experience for users while meeting their technical requirements, like ensuring unique team and game names, securing user data, and supporting distributed systems.

To achieve this, we’ll implement design patterns like the **Singleton Pattern** to keep only one instance of the game running in memory, and the **Iterator Pattern** to check for unique names efficiently. This approach will ensure scalability and simplicity while minimizing errors. By considering the client’s needs for performance, security, and cross-platform support, we’ll deliver a solution that sets the stage for the game’s success in a distributed environment.

## 

## **Requirements**

Here’s what the client needs:

1. **One Game Instance**: Only one game can be active in memory at a time.
2. **Unique Identifiers**: Game, team, and player names must be unique.
3. **Multi-Platform Support**: The game must work on web, mobile, and desktop platforms.
4. **Secure Data Handling**: User information must be protected during storage and transmission.
5. **Performance and Scalability**: The game must perform efficiently, even in a distributed system.

## [**Design Constraints**](#_2et92p0)

Here’s what we need to consider while designing this solution:

1. **Concurrency**: In a distributed environment, multiple users may try to access or modify the same data simultaneously. To manage this, we’ll use the Singleton Pattern to control access to the game instance and ensure consistency.
2. **Cross-Platform Compatibility**: We’ll need to use technologies like Java, HTML, and CSS that can work across different devices and operating systems.
3. **Security**: Protecting user data is critical. This means encrypting sensitive information during transmission (e.g., SSL/TLS) and using secure authentication methods like OAuth 2.0.
4. **Resource Management**: The game must handle memory and storage efficiently, especially when running on devices with limited resources.

These constraints guide how we approach the project, ensuring we meet the client’s requirements without compromising functionality or security.

## [System Architecture View](#_ilbxbyevv6b6)

For this project, we’re focusing on the software design. While there’s no specific architecture diagram required, the game’s system will rely on:

* A distributed server setup for hosting the game data.
* APIs to connect the client-side applications (web and mobile) to the server.
* Secure communication channels between components.

This architecture ensures scalability and smooth operation across platforms.

## [**Domain Model**](#_8h2ehzxfam4o)

**"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.**

The UML diagram gives us a clear picture of how the system is organized. Here’s a breakdown:

1. **Entity Class**:
   * Acts as the foundation for Game, Team, and Player. It holds common attributes like id and name.
2. **Game Class**:
   * Represents a single game instance and manages multiple teams.
3. **Team Class**:
   * Holds a list of players and provides methods to add or manage them.
4. **Player Class**:
   * Represents individual participants in the game.
5. **GameService Class**:
   * Manages all the games, teams, and players.
   * Uses the **Singleton Pattern** to ensure only one instance exists.
   * Uses the **Iterator Pattern** to maintain unique names for games and teams.

This design is efficient, easy to maintain, and fulfills the client’s requirements for scalability and data integrity.

## [**Evaluation**](#_2o15spng8stw)

| **Aspect** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server-Side** | Offers server-based deployment with reliable performance. Requires Apple hardware, which increases cost. | Highly scalable, cost-effective, and customizable. Excellent for server-based hosting with tools like Apache and NGINX. | Widely used, easy to configure. Licensing costs and resource usage are higher compared to Linux. | Not suitable for hosting a web server due to resource constraints. |
| **Cost for Hosting** | Apple hardware adds upfront cost; macOS licensing is free once hardware is purchased. | Free and open-source with optional costs for enterprise support (e.g., Red Hat). | Requires Windows Server licenses; additional Client Access Licenses (CALs) may be needed. | Mobile devices are not designed to act as hosts for web applications. |
| **Client-Side** | Smooth development using Xcode; strong integration with Apple ecosystem but limited to macOS users. | Supports modern web technologies (HTML5, CSS, JavaScript) but requires more expertise to configure and maintain. | Compatible with popular tools (Visual Studio, .NET); supports responsive web design. | Development requires frameworks (React Native, Flutter) for cross-platform compatibility. |
| **Development Tools** | Xcode and IntelliJ IDEA are effective but may require licensing for advanced features. | Free tools like Eclipse and IntelliJ IDEA are robust and widely supported. | Paid tools like Visual Studio are robust, with free community editions available. | Android Studio (free) and Xcode (requires Apple hardware) are key for mobile app development. |
| **Scalability** | Scalable but requires investment in high-performance Apple servers. | Excellent scalability with containerization tools like Docker and orchestration via Kubernetes. | Scales well but demands higher costs for hardware and additional software. | Relies on back-end servers for scaling; mobile devices are limited in independent scalability. |
| **Cross-Platform Development** | Limited flexibility due to proprietary tools. | Highly flexible with support for modern technologies and programming languages. | Familiar ecosystem for many developers; good support for modern web standards. | Requires responsive design and specific SDKs for each platform (Android and iOS). |
| **Security** | Strong security features like built-in firewall and support for SSL/TLS encryption. | Highly secure with proper configuration; offers robust support for SSL/TLS and SSH. | Good security features but requires regular updates and patching to maintain integrity. | Security is dependent on app configuration and integration with back-end services. |

To help The Gaming Room make the best decisions for expanding "Draw It or Lose It," here’s a clear breakdown of how different platforms stack up for hosting and developing your game.

**1. Server-Side: Hosting the Game**

Each platform has its pros and cons when it comes to hosting your game on a server:

* **Mac:** Reliable and stable, but the catch is you’ll need Apple hardware, which can get expensive. While macOS is free to use once you have the hardware, scalability isn’t as flexible compared to other platforms.
* **Linux:** This is the MVP for hosting. It’s cost-effective (often free), scalable, and incredibly customizable. With tools like Apache, NGINX, Docker, and Kubernetes, Linux is ideal for handling thousands of players in a distributed setup.
* **Windows:** Familiar and easy to use, but there’s a price tag. Windows Server requires licenses and can be more resource intensive. That said, it’s great for teams already comfortable in the Windows ecosystem.
* **Mobile Devices:** Unfortunately, mobile devices aren’t built to host a game server. Instead, they’ll rely on the back-end servers you choose for scalability and performance.

**Recommendation:** Go with Linux for hosting. It’s budget-friendly, highly scalable, and perfect for the level of performance you need.

**2. Client-Side: Supporting Players Across Devices**

The goal here is to make the game accessible on desktops (Mac, Linux, Windows) and mobile devices (iOS, Android). Here’s what’s involved:

* **Mac:** Developing for Mac is smooth if you have a Mac! Tools like Xcode are great, but you’re locked into Apple hardware for development.
* **Linux:** Flexible and powerful, but it can take some extra effort to set up and optimize for web technologies. It’s a favorite among developers who enjoy customization.
* **Windows:** Windows is a solid all-rounder. It supports a variety of development tools, and it’s widely used, so most developers will feel right at home.
* **Mobile Devices:** You’ll need frameworks like React Native or Flutter to ensure compatibility across iOS and Android. Responsive design will also ensure the app works beautifully on devices of all shapes and sizes.

**What this means for you:** A responsive web application is the way to go for desktops, and using cross-platform mobile frameworks will save time and effort.

**3. Development Tools: What It Takes to Build This**

Here’s the toolkit you’ll need to bring "Draw It or Lose It" to life across platforms:

* **Languages:** JavaScript (React or Angular) for the web; Dart (Flutter) for mobile; Java/Kotlin for Android; and Swift for iOS.
* **IDEs:** Visual Studio Code and IntelliJ IDEA are versatile options. Android Studio is a must for mobile, and Xcode is great for iOS but requires a Mac.
* **Extras:** Docker for consistent environments, and CI/CD tools like Jenkins or GitHub Actions to automate testing and deployment.
* **Cost:** Many tools are free or open source, like Eclipse and Android Studio. However, premium tools like IntelliJ IDEA (Ultimate) or Apple hardware for Xcode will add to your budget.

**What’s the impact?** A project like this might need specialized developers for each platform. Using cross-platform frameworks (like Flutter) can minimize the need for separate teams and save time.

**Key Takeaways**

* **Server Platform:** Linux is the best choice for hosting, balancing cost, performance, and scalability.
* **Development Strategy:** Responsive web design for desktops and cross-platform frameworks (Flutter or React Native) for mobile will ensure the game is accessible to everyone.
* **Security:** Protect user data with SSL/TLS encryption and use OAuth 2.0 for authentication. Regularly update your system to stay ahead of vulnerabilities.
* **Scalability:** Use tools like Docker and Kubernetes to handle the growing number of players efficiently.

**Why This Matters**

By choosing the right platforms and tools, we can make "Draw It or Lose It" a seamless experience for players across all devices. The Linux-based server will ensure performance under heavy loads, and modern development frameworks will save time while delivering a polished product. With the right strategy, your game will be ready to scale and succeed.

## **Recommendations**

**1. Operating Platform**

I recommend Linux as the hosting platform for *Draw It or Lose It*. Here’s why:

* It’s cost-effective: Linux is free and open source, so you won’t have to worry about licensing fees.
* It’s highly scalable: Tools like Docker and Kubernetes work seamlessly with Linux, making it perfect for handling a growing number of players.
* It’s reliable under pressure: Linux is known for its stability and efficiency, which is essential for an online multiplayer game.

**Choosing Linux means you get performance, flexibility, and cost savings—all in one package.**

**2. Operating System Architectures**

Linux’s architecture is a perfect match for *Draw It or Lose It* because of its modular and flexible design:

* Modular Design: You can customize the operating system to include only what’s needed, which keeps things lightweight and efficient.
* Process Isolation: Linux keeps processes separated to ensure one task doesn’t interfere with another—great for a game environment with multiple players.
* Virtualization Support: Tools like Docker allow you to create isolated, stable environments, making deployment and scaling a breeze.

**This kind of setup ensures the game runs smoothly, whether you have 10 players or 10,000.**

**3. Storage Management**

For storing player data, I recommend a cloud-based database like AWS RDS or MongoDB Atlas. These solutions are:

* Scalable: As the game grows, so does the database—without losing performance.
* Reliable: They offer automatic backups and redundancy, so your data is always safe.
* Compatible: They integrate seamlessly with Linux servers and modern development tools.

**This means *Draw It or Lose It* can manage user data efficiently without any hiccups.**

**4. Memory Management**

Linux handles memory like a pro, which is essential for resource-intensive applications like games:

* Virtual Memory: It optimizes memory by moving less-used data to disk, freeing up space for critical processes.
* Garbage Collection: If we pair Linux with Java, it will automatically clear out unused memory to keep things running smoothly.
* Caching: Linux stores frequently used data in cache memory, making it quick and easy to access.

**The result? Fast load times and a seamless gaming experience, even when thousands of players are online.**

**5. Distributed Systems and Networks**

To allow the game to run across different platforms (like web, mobile, and desktop), we’ll use:

* RESTful APIs: These APIs will act as the “bridge” between the game server and players, enabling smooth communication.
* Load Balancers: Tools like NGINX will distribute player traffic evenly, preventing the server from becoming overwhelmed.
* Fallback Systems: In case of outages or network issues, we’ll have backup mechanisms to keep the game running.
* Real-Time Protocols: Using WebSockets or HTTP/2 ensures real-time interactions, so players see updates instantly.

**This setup ensures a fast and reliable experience, no matter what platform the players are using.**

**6. Security**

Security is non-negotiable, especially for user data. Here’s how we’ll protect it:

* **SSL/TLS Encryption:** This will encrypt all communication between players and servers, keeping their information private.
* **OAuth 2.0 Authentication:** This secure login system ensures only authorized users can access their accounts.
* **Regular Updates:** By keeping Linux servers and tools up to date, we’ll stay ahead of potential security threats.
* **Firewalls:** Using tools like UFW or iptables, we’ll block any malicious traffic trying to target the game server.