

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 | 02/06/2025 | Matt Kostandin | Initial version of the software design template for Draw It or Lose It |

**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)

Draw It or Lose It, currently an Android-only game, needs to be expanded into a web-based format that supports multiple platforms. To meet this goal, the proposed design introduces scalable and secure architecture and leverages software design patterns. This makes sure a seamless user experiences across diverse computing environments. The client must understand that implementing a single-instance game service, unique naming constraints, and cross-platform support will require careful planning and adherence to best practices. By following this proposal, The Gaming Room can streamline development and integrate additional features as their user base grows.

## Requirements

The Gaming Room needs a web-based version of Draw It or Lose It that is accessible to users on different platforms. A single-instance game service must handle multiple concurrent teams and make sure unique names for each game and team. Architecture should support scalability. This will make it simple to add new features in the future. The client also requires that any design decisions made are mindful of eventual hardware requirements and potential deployments to various operating systems (Windows, Mac, Linux) and mobile devices.

## [Design Constraints](#_2et92p0)

1. Only one instance of the main game service can be active in memory at any given time.
2. Game and team names must remain unique to avoid user confusion and duplication of in-game resources.
3. The system must adapt to a web-based, distributed environment which entails managing concurrency, load balancing, and potential network latency.
4. The design must allow for future scalability so that new features or server expansions can be introduced without significant refactoring.

## [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)

Draw It or Lose It uses a set of classes that illustrate standard object-oriented principles. The Entity class serves as a base, holding essential attributes like an identifier and a name. The Game class, inheriting from Entity, manages multiple teams. Each Team, also inheriting from Entity, organizes one or more players. The Player class inherits these same attributes from Entity. This structure demonstrates inheritance for shared attributes, maintains encapsulation of data within each class, and allows for polymorphic behaviors if future changes introduce unique team or player actions. These relationships efficiently meet requirements like preventing duplicate names and ensuring a single 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** | **Server-Based Hosting**: macOS can host web applications, but it is less common in enterprise-scale production due to limited commercial hosting providers.  **Licensing**: Apple hardware is typically required, which can be more expensive than commodity servers.  **Advantages**: macOS is Unix-based, offering stability, strong security, and a familiar POSIX environment similar to Linux.  **Weaknesses**: Higher hardware costs, smaller hosting market, fewer preconfigured server packages. | **Server-Based Hosting**: By far the most common and cost-effective choice. Linux distributions (e.g., Ubuntu, CentOS) dominate the server market.  **Licensing**: Generally free/open-source, reducing ongoing costs for The Gaming Room.  **Advantages**: Highly stable, extensive community support, large variety of hosting providers, seamless scalability.  **Weaknesses**: Requires expertise in server administration and command-line operations, though well-documented. | **Server-Based Hosting**: Windows Server is readily available from many hosting providers; integrates well with Microsoft stack (ASP.NET, MSSQL).  **Licensing**: Windows Server licenses can be significant, which raises ongoing costs.  **Advantages**: Excellent support for .NET-based applications, easy integration with Azure cloud services.  **Weaknesses**: Higher cost, sometimes more resource-intensive, can require patching overhead. | **Server-Based Hosting**: Typically not used for hosting. Mobile devices (iOS/Android) function almost exclusively as clients.  **Licensing**: App marketplace registration fees exist (e.g., Apple Developer Program, Google Play).  **Advantages**: Not applicable from the server perspective.  **Weaknesses**: Infeasible as a primary server due to hardware limits, connectivity, battery, and reliability concerns. |
| **Client Side** | **Desktop Web Browsers**: Safari, Chrome, Firefox. A responsive HTML5 interface ensures the game works similarly to other platforms. **Application Requirements**: Minimal extra cost for supporting macOS-specific features if the solution is entirely browser-based. If a native macOS app were needed, the development cost and expertise requirements would rise.  **Advantages**: Apple user base is loyal; user experience (UX) can be polished on macOS.  **Weaknesses**: A smaller market share than Windows. | **Desktop Web Browsers**: Chrome, Firefox, other Linux-compatible browsers. A responsive HTML5 interface requires minimal overhead for Linux desktop users.  **Application Requirements**: Because the front end is browser-based, minimal additional cost is needed for Linux desktop support.  **Advantages**: Highly flexible, open-source environment, easy to bundle supporting libraries and frameworks.  **Weaknesses**: Desktop gaming is less common on Linux, though it has a devoted community. | **Desktop Web Browsers**: Edge, Chrome, Firefox. Windows remains the dominant desktop OS, so thorough QA testing is critical for compatibility.  **Application Requirements**: Ensuring support across multiple Windows versions (10, 11, etc.) can increase testing overhead.  **Advantages**: Largest user base for desktop; supporting Windows is essential for broad reach.  **Weaknesses**: Must handle OS version fragmentation and potentially more complex security setup. | **Mobile Devices**: iOS and Android. Ensuring a responsive web interface will cover mobile web browsers, though many companies also provide native apps for performance or UX reasons.  **Application Requirements**: If going native, separate codebases (Swift/Objective-C for iOS and Java/Kotlin for Android) or cross-platform frameworks (Flutter, React Native) are needed.  **Advantages**: The largest and fastest-growing user base is on mobile.  **Weaknesses**: Additional cost, time, and expertise for ongoing maintenance, plus store approval processes. |
| **Development Tools** | **Languages/Tools**: Xcode can be used for native macOS/iOS development; IntelliJ, VS Code, or other IDEs for cross-platform web or backend.  **Licensing Costs**: Xcode is free, but Apple Developer Program membership is required for official app distribution.  **Team Impact**: Knowledge of macOS-specific build systems, or Docker-based workflows if standardizing across the team. | **Languages/Tools**: Commonly use Java, Python, Node.js, Go, etc., with IDEs like Eclipse, IntelliJ, VS Code. Command-line utilities are standard, and Linux-based CI/CD pipelines are popular.  **Licensing Costs**: Typically none; tools and OS are open-source or free-tier.  **Team Impact**: Large open-source community support; possible need for administrators versed in Linux if scaling. | **Languages/Tools**: Visual Studio for .NET, C#, or C++; IntelliJ, Eclipse, and others for Java.  **Licensing Costs**: Windows licenses for servers; Visual Studio has free (Community) editions, but enterprise features may cost more.  **Team Impact**: May require specialized .NET experts or knowledge of Windows-based services, plus potential reliance on Microsoft ecosystem. | **Languages/Tools**: Xcode for iOS, Android Studio for Android, or cross-platform frameworks like Flutter, React Native, and Unity for 2D/3D.  **Licensing Costs**: Apple Developer Program ($99/year), Google Play registration (one-time $25).  **Team Impact**: Might require multiple dev teams unless using a cross-platform approach. Need continuous updates for OS changes. |

## 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 recommended for the server environment, offering reliability, scalability, and cost-effectiveness. Linux distributions typically provide better stability for high-demand services, while their modular design lets you include only the components you need. This reduces overhead and simplifies system maintenance, ensuring that Draw It or Lose It remains responsive, secure, and easily scalable to meet growing player demands.
2. **Operating Systems Architectures**: Linux provides a multi-user, multitasking architecture well suited for concurrent processes, enabling each service (such as database, application logic, and web server) to run with dedicated resources. Containerization platforms (Docker or Kubernetes) leverage this design to isolate application components in lightweight, portable containers. This approach eases deployment, streamlines updates, and supports rolling releases, reducing downtime while distributing workloads efficiently across multiple nodes.
3. **Storage Management**: A cloud-based or local relational database system (MySQL or PostgreSQL) is appropriate due to its compatibility with Linux and strong community support. These databases excel in handling transactional data, such as player information and game states. Cloud-based storage solutions offer scalability, automated backups, and failover options, enhancing reliability. Local deployments may be preferable if there are strict latency requirements or compliance considerations, but both approaches allow Draw It or Lose It to store and access critical data seamlessly.
4. **Memory Management**: Linux’s virtual memory system efficiently manages processes by allocating resources dynamically, preventing memory conflicts between different services. When combined with a garbage-collected language like Java, the risk of memory leaks is further minimized since the system actively reclaims unused objects. Developers can also fine-tune memory usage with configuration parameters to optimize performance and address the specific needs of Draw It or Lose It’s real-time gameplay.
5. **Distributed Systems and Networks**: Draw It or Lose It can communicate across various platforms using REST APIs or WebSockets, ensuring low-latency, bidirectional communication for features like live drawing or instant user feedback. Load balancing and container orchestration enhance system reliability, automatically redistributing traffic when a node is busy or offline. Synchronizing game state across distributed instances is simplified by using a reliable database, supplemented with in-memory data stores such as Redis to reduce latency and handle high-volume requests.
6. **Security**: Encryption over HTTPS safeguards data in transit by preventing unauthorized access during gameplay or account management. Secure authentication methods (token-based or session-based) further protect user sessions, reducing the chances of credential theft. By enforcing strict compliance with OS patching, monitoring firewall rules, and following secure data handling practices (hashed passwords, minimal data exposure), The Gaming Room can mitigate risks. Additionally, regular vulnerability assessments, intrusion detection systems, and role-based access control (RBAC) help ensure the game remains secure against emerging threats across all platforms.