

Draw It or Lose It Web App

# **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 | 07/19/25 | Augustus Wesseh | Initial draft completed with key design content for Project One. |

**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 is transitioning their popular Android-only game, Draw It or Lose It, to a web-based, cross-platform system to increase accessibility across desktop and mobile platforms. To ensure scalability, performance, and maintainability, the design will follow software architecture best practices including implementation of the Singleton pattern to maintain one instance of game logic in memory, Iterator pattern to avoid name duplication, and a client-server model with RESTful APIs for platform-independent communication. This system will allow thousands of concurrent players while supporting modular upgrades and efficient development.

## Requirements

Business Requirements:  
- Convert the existing Android-only game into a web-based, cross-platform application.  
- Ensure the system supports multiple teams with multiple players.  
- Team and game names must be unique to prevent conflicts.  
  
Technical Requirements:  
- Only one game instance can exist in memory (Singleton pattern).  
- Unique identifiers must be generated for each game, team, and player.  
- Application should be accessible across major operating systems and browsers.

## [Design Constraints](#_2et92p0)

1. Platform Independence: The application must run across all major operating systems, so development will prioritize web standards and avoid platform-specific code. This encourages usage of technologies like HTML5, CSS, and JavaScript frameworks (e.g., React).

2. Singleton Enforcement: Only one game instance should exist in memory to avoid conflicts and maintain session integrity. This will be enforced using the Singleton design pattern, ensuring centralized state management.

3. Unique Name Validation: To prevent duplicate names for games and teams, an Iterator pattern is implemented in methods like addGame() and addTeam() to check for name uniqueness before object creation.

4. Web Distribution Model: The application must perform well in a browser-based environment. To reduce bandwidth and latency, assets (such as drawings) will be compressed (e.g., WebP format) and served via a Content Delivery Network (CDN).

## [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 Domain Model is structured using object-oriented principles. The Entity superclass holds shared attributes (id, name) inherited by Game, Team, and Player. The GameService class enforces centralized game control and uses the Singleton pattern to maintain one in-memory game instance. Relationships follow composition, where a Game contains multiple Teams, and each Team contains multiple Players.  
  
Key OOP principles applied:  
- Encapsulation: Internal collections (like game lists) are private and only modifiable via controlled methods.  
- Polymorphism: Classes override toString() methods to provide custom output.  
- Inheritance: Common fields (like id and name) are shared through the Entity superclass, promoting reuse and reducing redundancy.  
  
These principles collectively contribute to better maintainability, scalability, and readability of the code.

**"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 host web-based applications using server tools like Apache or Nginx. It’s Unix-based, which makes it secure and stable. However, it’s not typically used in large-scale web hosting due to high hardware costs and licensing limitations. While MacOS can support development and limited deployment for testing or internal environments, it’s not an ideal platform for enterprise-level hosting. | Linux is the most popular server platform for hosting web applications. It’s open-source, highly customizable, and supported by robust server distributions like Ubuntu Server, CentOS, and Red Hat. Its performance, scalability, and low cost make it ideal for hosting thousands of concurrent players. Security is strong, and most cloud services (AWS, Azure, GCP) support Linux environments natively. | Windows Server provides a stable environment for hosting ASP.NET-based applications and offers GUI tools that are helpful for those unfamiliar with command-line interfaces. It does, however, come with higher licensing fees and resource consumption. Windows is suitable for organizations already tied into the Microsoft ecosystem but may be less cost-effective than Linux for large-scale web deployments. | Mobile devices are not typically used to host web-based applications due to limited hardware, battery life, and inconsistent connectivity. Their role in server-side deployment is minimal to none. However, they can interact efficiently with back-end servers via REST APIs. Apps are designed to run as clients rather than hosts in a distributed system. |
| **Client Side** | Developing for Mac requires access to Apple’s tools and environments like Xcode. It’s essential to test the game interface on Safari and ensure compatibility with MacOS hardware. Developers familiar with Unix systems will find MacOS easier to integrate with CI/CD pipelines. However, development costs are higher due to Apple hardware requirements. | Linux has fewer standard browsers and a fragmented desktop environment, which requires thorough testing across different distributions and browsers like Firefox and Chromium. Developers benefit from extensive command-line tooling and package management. However, supporting Linux requires deeper technical knowledge and can be time-consuming due to ecosystem fragmentation. | Windows is the most widely used desktop OS, so it’s critical to ensure seamless functionality in Edge, Chrome, and Firefox. Development tools are mature, and testing can be done across various screen sizes and hardware setups. Developers should consider accessibility features and update compatibility across Windows versions. | Supporting mobile devices means ensuring the application is fully responsive, touch-optimized, and performant on varying screen sizes. Developers need to test across both iOS and Android environments using tools like BrowserStack or physical devices. Time and cost can increase depending on the number of supported devices, but reach is maximized. |
| **Development Tools** | For Mac, Xcode is the primary IDE, and tools like Homebrew, Terminal, and Safari Web Inspector are useful. Languages like Swift, Objective-C, and JavaScript are commonly used. Some cross-platform frameworks like React Native and Flutter can be developed on MacOS. Licenses for Apple developer programs are required. | Linux development is often done using open-source tools like Eclipse, VS Code, and NetBeans. Languages such as Java, Python, PHP, and JavaScript are widely used. Package managers like APT and YUM simplify dependency handling. Most tools are free, but the learning curve may be steeper for beginners. | Windows developers typically use Visual Studio or IntelliJ IDEA. Common languages include C#, JavaScript, and Java. Microsoft offers powerful debugging and GUI design tools. Licensing may apply for enterprise versions of IDEs and the Windows OS itself. Windows is developer-friendly with strong community support. | Mobile app development uses tools like Android Studio for Android and Xcode for iOS. Cross-platform development can be done using Flutter or React Native. Testing frameworks like Appium or Firebase Test Lab are helpful. Licensing costs apply for app store publishing, and development requires familiarity with native and hybrid design principles. |

## Recommendations

1. **Operating Platform**: For The Gaming Room to successfully expand *Draw It or Lose It* to multiple computing environments, I recommend deploying the game on a Linux-based platform for the server-side and supporting client-side access via responsive web technologies. Linux offers a stable, scalable, and cost-effective environment for hosting web applications. It’s also widely supported across all major cloud providers, making it an ideal choice for a game expected to handle traffic from various client platforms like desktop (Windows, Mac, Linux) and mobile (iOS, Android).
2. **Operating Systems Architectures**: The Linux platform is built on a modular and secure architecture, which makes it highly adaptable to a wide range of deployment scenarios. The OS separates user space from kernel space, ensuring a clear boundary between application processes and core system functions. This separation enhances security and stability. Linux also supports a variety of system architectures (x86, ARM, etc.), enabling flexibility if the client decides to scale horizontally or host across different devices or cloud regions.
3. **Storage Management**: A Linux-based environment can utilize logical volume managers (LVM) and file systems like ext4 or XFS to manage storage efficiently. For *Draw It or Lose It*, cloud-based object storage (such as AWS S3 or Google Cloud Storage) can be integrated for scalable storage of media assets like stock drawings and user-generated content. Local caching mechanisms can also be employed to reduce latency. The storage system will be capable of dynamic scaling to match demand, with robust backup and disaster recovery options.
4. **Memory Management**: Linux uses a virtual memory system that includes paging, segmentation, and swapping to manage memory efficiently. It also implements caching and shared memory to optimize performance. The kernel’s out-of-memory (OOM) killer ensures stability in low-memory conditions. For the game, this means server processes can be managed effectively, with minimal risk of crashes due to memory leaks or spikes. Technologies like containers (e.g., Docker) can further isolate memory usage per component, making it easier to scale and maintain.
5. **Distributed Systems and Networks**: To support cross-platform gameplay and communication, *Draw It or Lose It* should use RESTful APIs over HTTPS as the core communication method between client devices and the server. Data serialization formats like JSON ensure compatibility across devices. Services can be deployed using microservices architecture, allowing each component (game logic, authentication, leaderboard, etc.) to scale independently. Tools like Kubernetes can be used to manage distributed services. Additionally, load balancers, health checks, and retries will help mitigate the effects of network outages or latency. Message queues (e.g., RabbitMQ or Kafka) can also be used for asynchronous tasks like notifications or chat.
6. **Security**: Protecting user data and platform integrity is essential. The system will implement HTTPS for all communications, ensuring end-to-end encryption. Token-based authentication (JWT) and role-based access control (RBAC) will limit user access to only what’s necessary. On the server, Linux’s built-in tools like SELinux or AppArmor can enforce process isolation and access policies. Data at rest will be protected using disk encryption (e.g., LUKS), while user credentials and sensitive information will be securely hashed and stored. Additionally, firewall rules and intrusion detection systems (IDS) will provide an extra layer of defense against external threats.