

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 | 05/25/2025 | Ian Brown | Initial draft for the “Draw It or Lose It” software design document. |

**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 industry is evolving rapidly, demanding applications that are both visually compelling and functionally robust across diverse platforms. The design challenge for "Draw It or Lose It" is to create an engaging, web‑based game application that combines real‑time interactive gameplay with a scalable distributed architecture. Our proposed solution leverages modern web technologies, client‑server communication protocols (such as REST APIs and WebSockets), and a modular design that guarantees performance, scalability, and security without sacrificing the creative gameplay dynamics that define the experience.

For the client, critical considerations include ensuring that the design supports dynamic user interactions, accommodates rapid scaling during peak usage times, and maintains strict security controls for user data. This solution is built upon a solid object‑oriented foundation that enables maintainability and iterative enhancements, ensuring that the application meets both current demands and future expansion needs.

## Requirements

The business requirements stipulate that the application must be engaging, competitive, and socially shareable, appealing to both casual gamers and enthusiasts. On the technical front, requirements include real‑time data transmission, cross‑platform compatibility, and high availability in a distributed web environment. Additionally, the application must integrate secure user authentication, enable seamless updates, and support a modular codebase that facilitates future feature enhancements.

## [Design Constraints](#_2et92p0)

Developing "Draw It or Lose It" in a web‑based distributed environment introduces several design constraints that directly influence application development:

Scalability and Responsiveness: The application must handle a high volume of concurrent users. This constraint necessitates architecture that supports load balancing and server clustering. Real‑time gameplay further limits acceptable network latency and demands quick server response times.

Cross‑Platform Compatibility: Given that modern gamers use a variety of devices (desktops, tablets, smartphones), the user interface and backend services must be compatible with major browsers and operating systems. This constraint influences the choice of responsive design frameworks and client‑side technologies.

Security and Data Privacy: Protecting user data is paramount. The design must include secure authentication protocols, data encryption for transmission and storage, and adherence to privacy regulations. These constraints impact the selection of technologies and require ongoing security assessments.

Distributed Environment Implications: Running the application in a distributed environment means that data consistency, synchronization, and fault tolerance become critical. The architecture must be designed to gracefully handle network outages and ensure the integrity of shared state across servers.

Overall, these constraints demand the use of robust, industry‑proven frameworks and a detailed plan for managing distributed state, which ultimately leads to an architecture that is both resilient and flexible.

## [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 provided UML class diagram forms the backbone of our object‑oriented design by capturing key entities and their relationships in "Draw It or Lose It." At its core is the abstract Entity class, which centralizes common attributes (such as unique identifiers, timestamps, and status flags) and behaviors that are shared across multiple domain objects.

Inheritance: Notice how Player (and potentially Team or Game) inherits from Entity, ensuring that each domain object carries common attributes such as an ID or timestamp. This reduces redundancy and promotes a standardized approach to data management.

Encapsulation: Each class manages its own state and behavior, exposing only the necessary methods to interact with other parts of the system. This protects the internal workings of each class, maintaining data integrity and easing maintenance.

Aggregation/Composition: The relationship between Team and Player (and similarly between Game and Team/Player) illustrates aggregation. The Team class acts as a container for multiple Player objects, mirroring real-world groupings and promoting a clear hierarchical structure within the application.

Abstraction: By separating the game’s core logic into the GameService class, the system abstracts complex behaviors into well-defined methods and interfaces. This separation helps in managing change over time and allows for potential scalability or modification of game rules without altering the core domain model.

By adhering to these object‑oriented programming principles, the design not only supports modular development and ease of maintenance but also ensures that the evolving requirements can be integrated with minimal disruption to the overall system.

**"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** | Mac systems are built on a Unix-based foundation that offers robust security features and a stable environment. They integrate well with the Apple ecosystem and come with solid development facilities for testing web-based applications. However, their relatively limited market share in high-volume server deployments and higher cost of hardware make them less popular for large-scale production environments.   |  | | --- | |  | | Linux is the de facto standard for server-side deployments thanks to its open-source nature, extensive community support, and superior scalability. Its flexibility, low cost and customization options make it ideal for handling high volumes of web traffic and distributed workloads. The trade-off is that running Linux servers may require more technical expertise to manage and optimize, particularly in complex setups. | Windows Server offers ease of integration with enterprise tools, particularly within the .NET ecosystem, and is known for its strong administrative interfaces. It provides a familiar environment for many businesses but can be challenged by higher licensing fees and comparatively heavier resource requirements. These factors may increase operational costs, especially in large-scale, resource-intensive applications. | |  | | --- | |  |   Mobil devices are primarily designed as client endpoints rather than server hosts; their processing power, storage, and network capabilities are limited for traditional server roles. While they shine in providing responsive client interfaces, using them as the primary server platform is fraught with challenges related to connectivity, scalability, and maintenance, making them impractical for hosting core server functions. |
| **Client Side** | Mac environments benefit from high‑quality graphics and a streamlined development experience leveraging native tools such as Xcode. The design considerations focus on delivering a responsive and visually attractive interface, while development may involve higher costs when targeting a market of users accustomed to the premium feel of Apple devices. The ecosystem is well‑defined, but expertise in Mac‑oriented design principles is essential to maximize user engagement. | Linux clients can benefit from low‑cost, open‑source frameworks that are both flexible and adaptable. The diversity of Linux distributions requires careful planning to ensure consistent user experience, but the overall cost of deployment is minimized by the absence of licensing fees. However, the variety of desktop environments and tools available might increase development time, as compatibility across distributions should be factored into planning and testing. | Windows offers a large user base and strong support for legacy as well as modern applications, with robust development tools like Visual Studio providing significant advantages. The client‑side development process is supported by mature frameworks and a wealth of libraries, though the need to cater to both old and new technologies can complicate the development process. Licensing fees and the diversity of Windows versions also add to the time and expertise required for smooth deployment. | Mobile client development demands attention to unique user interfaces that are optimized for touch and small screen resolutions. Using native development tools like Xcode for iOS and Android Studio for Android, or cross‑platform solutions (such as React Native or Flutter), requires balancing cost, time, and specialist expertise. While the effort can result in a highly engaging user experience, ensuring performance across multiple device configurations can increase both the development time and support complexity. |
| **Development Tools** | For macOS, developers often rely on Xcode and related tools, using languages such as Swift and Objective‑C for native application development. Tools like Electron may also be used for cross‑platform desktop applications. These tools integrate seamlessly within the macOS environment, allowing rapid development cycles and efficient debugging, but they require familiarity with Apple’s development ecosystems and sometimes demand additional investment in premium hardware. | On Linux, the toolchain is rich with open‑source options. IDEs such as Eclipse, IntelliJ IDEA, or Visual Studio Code are widely used alongside programming languages like Python, Java, or JavaScript (Node.js). The extensive library of free tools and customizability make Linux a favorite for many developers, although setting up the ideal development environment may require deeper technical knowhow and system configuration. | Windows developers typically take advantage of Microsoft Visual Studio and the .NET framework, which offer robust features for debugging, designing, and deploying applications. The ecosystem supports languages such as C# and F#, making it a powerful environment for building enterprise applications. However, the costs associated with licensing and sometimes higher system requirements can be considered when allocating resources for development. | Mobile development has a segmented tool landscape. For iOS, Xcode and Swift or Objective‑C are the primary tools, while Android development hinges on Android Studio paired with Kotlin or Java. Cross‑platform frameworks like Flutter, React Native, and Xamarin reduce duplicate efforts, though they may introduce challenges in achieving platform‑specific optimizations. The complexity of managing multiple codebases or ensuring uniform performance across platforms tends to increase both the cost and the expertise required for successful delivery. |

## 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**: We recommend a Linux‑based cloud platform for hosting Draw It or Lose It. Linux stands out as the industry standard for high‑performance web applications due to its stability, flexibility, and cost‑effectiveness. A Linux environment is ideally suited for containerized deployments (using Docker) and orchestration frameworks like Kubernetes, which ensure rapid scalability, easy management of microservices, and seamless integration with various computing environments. This platform choice also facilitates expansion to hybrid or multi‑cloud architectures, enabling The Gaming Room to meet growing demand without locking into proprietary ecosystems.
2. **Operating Systems Architectures**: The recommended architecture adopts a microservices model where each functional component of Draw It or Lose It is isolated into discrete, stateless services. By leveraging containerization (e.g., Docker) on Linux, the architecture ensures that each service—be it the game engine, player management, or real‑time communication—runs in its own isolated environment with defined resource quotas. This modular approach enables continuous deployment, independent scaling of services, and fault isolation. The use of a service mesh (such as Istio) can further simplify inter‑service communication, enhance security, and monitor service health, all of which are essential in a distributed operating system architecture.
3. **Storage Management**: For efficient storage management, a hybrid approach is advised that combines a relational database (such as PostgreSQL) with a NoSQL solution (such as MongoDB or Cassandra). The relational database can manage structured data like user profiles, session information, and transactional records, while the NoSQL database is more suited for unstructured, real‑time data like gameplay events and logs. Coupling these with an in‑memory caching layer (using Redis or Memcached) improves read performance and reduces latency. This multi‑tier storage solution not only supports high availability through data replication and clustering but also allows for horizontal scaling as user demand grows.
4. **Memory Management**: On a Linux‑based cloud platform, memory management is handled at several layers. First, the operating system itself uses advanced memory management techniques (such as paging, swapping, and use of cgroups in container environments) to ensure that no single process overconsumes system resources. Container orchestration (via Kubernetes) lets you define memory limits per service, enabling efficient use of system memory and protecting against memory leaks. Additionally, the application layer employs language‑level garbage collection (e.g., Java’s JVM or managed runtimes in Node.js and Python) to reclaim unused memory. Coupled with application‑level caching (using tools like Redis), these techniques ensure that Draw It or Lose It can handle dynamic loads while maintaining system stability and responsive performance.
5. **Distributed Systems and Networks**: To enable seamless communication between various platforms, Draw It or Lose It will leverage a distributed microservices architecture that uses well‑defined APIs and messaging protocols. RESTful APIs facilitate standard communication for asynchronous tasks, while WebSocket connections provide the real‑time interactivity required by the game. Load balancers, reverse proxies, and service discovery tools help in managing network traffic and routing requests efficiently. Furthermore, deploying services across multiple availability zones or regions can mitigate the impact of connectivity issues or outages. Implementing a resilient network infrastructure with failover strategies—such as using circuit breakers and automated health checks—ensures that even if one part of the system faces disruptions, overall functionality is maintained.
6. **Security**: Security is non‑negotiable for The Gaming Room. The recommended Linux‑based platform inherently supports robust security frameworks, such as SELinux or AppArmor, that enforce strict access controls at the OS level. For application‑level security, all data transmissions should be encrypted using TLS/SSL protocols; sensitive data at rest must be protected using strong encryption standards. Secure authentication and authorization mechanisms (e.g., OAuth 2.0 and JWT tokens) are essential to verify and manage user access. Regular vulnerability assessments, automated patch management, and intrusion detection systems will ensure timely mitigation of potential threats. Additionally, network security can be enhanced by implementing firewalls, anti‑DDoS measures, and employing role‑based access control models to limit exposure and protect personal and gameplay data across all connected platforms.