

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.2

## 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 | 09/20/2025 | Jacob May-Eschenbach | Initial Software Design Doc created to meet The Game Room’s requirements. |
| 1.1 | 10/01/2025 | Jacob May-Eschenbach | Provide an Architectual pattern and explain the characteristics, advantages, and weaknesses of each platform (Linux, Mac, windows, and mobile platforms). |
| 1.2 | 10/14/2025 | Jacob May-Eschenbach | Completed the Recommendation section. |

**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)

<Write a summary to introduce the software design problem and present a solution. Be sure to provide the client with any critical information they must know in order to proceed with the process you are proposing.>

The Gaming Room is looking to expand its game *Draw It or Lose It* into a web-based version that can be accessed across multiple platforms and devices. This solution will broaden the customer base, enhance accessibility, and generate new revenue opportunities. To meet these needs, the application will be designed using industry best practices in software development and optimized to run seamlessly on all modern browsers.

Requirements

* A game will have the ability to have one or more teams involved.
* Each team will have multiple players assigned to it.
* Game and team names must be unique to allow users to check whether a name is in use when choosing a team name.
* Only one instance of the game can exist in memory at any given time. This can be accomplished by creating unique identifiers for each instance of a game, team, or player.
* The web application must be compatible with modern browsers.
* Must follow industry best practice (software design and security)

## [Design Constraints](#_2et92p0)

|  |  |
| --- | --- |
| **Constraint** | **Implication for Development** |
| **Web-Only Platform** | Development must rely on web standards (HTML5, CSS, JavaScript/TypeScript). Native mobile code cannot be reused directly, requiring refactoring of the Android version into a browser-based solution. |
| **Multi-User Concurrency** | The application must support many users interacting simultaneously. Developers must handle race conditions, concurrent updates, and ensure consistent game state across all connected clients. |
| **Team and Player Management** | Teams must allow multiple players, and each entity (game, team, player) must have unique identifiers. Requires strict database validation and UI checks for name availability. |
| **Time-Sensitive Gameplay** | Game logic is dependent on strict timers (rounds, overtime). Server-side clock control is required to ensure fairness and prevent manipulation from client-side devices. |
| **Single Instance of Game Sessions** | Only one active instance of a given game can exist. Developers must implement singleton patterns in memory. |
| **Cross-Device Accessibility** | Must run smoothly on desktops, tablets, and smartphones. Responsive design and adaptive layouts are essential for usability on varying screen sizes. |
| **Performance Limitations** | Latency must be minimized to keep gameplay real-time. Requires optimized networking, efficient rendering, and possible caching strategies. |
| **Security Requirements** | Must protect against unauthorized access and data leaks. Development must implement secure authentication, encrypted communication, and validation of all inputs. |
| **Budget and Time Constraints** | Limited resources require prioritization of core features first. Extra features (leaderboards, advanced graphics) may be postponed to later phases. |

## [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)

<Describe the UML class diagram provided below. Explain how the classes relate to each other. Identify any object-oriented programming principles that are demonstrated in the diagram and how they are used to fulfill the software requirements efficiently.>

## Class Descriptions and Relationships:

* **ProgramDriver and SingletonTester:**  
   ProgramDriver is the main class with the main() method. This is where the program starts running. It “uses” the SingletonTester class, which means it relies on SingletonTester to check that the singleton pattern works correctly.
* **Entity:**  
   Entity is the parent class for Game, Team, and Player. It holds shared attributes like id: long and name: String. All of the subclasses automatically get those attributes as well.
  + **Game, Team, and Player:**

These three classes inherit from the Entity class, which means they get the id and name attributes from it. They also have their own specific attributes and methods. The 0..\* notation shows their relationships. For example, a Game can have zero or more Team objects, and a Team can have zero or more Player objects.

* **GameService:**  
   GameService is where everything is managed. This is also where the singleton pattern takes place as there is only one instance that has been created. It keeps track of lists of games and teams, handles adding or retrieving games, and manages unique IDs for games, teams, and players. The 0..\* on its associations just means it can manage as many game or team objects as needed starting at zero.

**Object-oriented Programming Principles:**

The classes demonstrate a few object-oriented principles. Inheritance is shown with the Entity class and its subclasses (Game, Team, and Player). The subclasses share common attributes like id and name as well as methods such as getId() and getName(). This helps avoid repeating code. Another principle shown is encapsulation. This is used to keep data safe inside the classes. The attributes are made private so they can’t be accessed directly. Methods are made public (addGame() or getGame()) and are used to interact with the data. This protects the program and its data. Polymorphism is another principle used. The subclasses (Game, Team, and Player) are all types of Entity. This means they can be treated as Entity objects as needed. This makes methods more flexible since a single method could operate on any of the subclasses. Finally, a singleton pattern is shown with the GameService class. This ensures that only one instance of the GameService is running. It uses a private constructor to prevent new copies from being made, a private static attribute to store the instance, and a public getInstance() method to access it.

"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** | Can host web apps using Apache, Nginx, or Node.js but mainly used for small setups or testing.  Works well with Apple’s ecosystem for iOS/macOS builds.  Secure and stable UNIX-based OS that supports modern web browsers.  Not ideal for large-scale hosting or handling thousands of players.  Limited cloud support and higher hardware costs make it less scalable.  Yes, supports server-based deployment but not common for production use.  No software license fees, but Apple hardware and hosting are expensive. | Can host web apps using Apache, Nginx, Node.js, Python, Java, or .NET Core.  Open-source, stable, secure, and widely used for large-scale hosting.  Supports containers (Docker) and orchestration (Kubernetes) for scalability.  Can handle thousands of players and multiple game instances efficiently.  Strong cloud support with many providers offering Linux-based servers.  Yes, fully supports server-based deployment for production environments.  No OS licensing fees; mostly free to use, only pay for cloud hosting or optional enterprise support. | Can host web apps using IIS, .NET, Node.js, or other web technologies.  Well-suited for Microsoft-based development and integration with Azure cloud services.  User-friendly management tools and GUI make setup easier for admins.  Can support multiple game instances, teams, and thousands of players, but heavier resource use than Linux.  Scales well in enterprise environments but less common for container-based deployments.  Yes, fully supports server-based deployment for production.  Licensing costs can be high — Windows Server requires per-server licenses and possibly CALs; cloud pay-as-you-go options are available. | Mobile devices (iOS, Android) cannot host web applications; they act as clients connecting to a server.  Useful for testing or development with emulators, but not practical for production hosting.  Advantages: players can access the game anywhere via browsers or apps.  Weaknesses: limited processing power, battery-dependent, no persistent network for hosting.  Server-based deployment: **not applicable**; must connect to a backend server.  Licensing costs: none for hosting, but distribution requires app-store fees (Apple: $99/year, Google Play: $25 one-time). |
| **Client Side** | Supports modern browsers (Safari, Chrome, Firefox) for responsive web apps.  Can be used for native macOS or iOS apps if desired.  Development must ensure cross-browser compatibility and responsive design for multiple screen sizes.  Cost and time increase if building native apps; web-based client is faster and simpler.  Skills needed: HTML, CSS, JavaScript (React, Vue), optional Swift for iOS.  Apple Developer Program subscription ($99/year) required for distributing apps. | Users access the game via browsers (Chrome, Firefox, Edge) on Linux desktops.  Typically, no native Linux client is needed; a responsive web app is sufficient.  Development must ensure consistent UI and functionality across browsers.  Development is lower cost and faster if focusing on responsive web app only.  Skills needed: HTML, CSS, JavaScript, front-end frameworks (React, Vue).  No OS licensing costs. | Supports modern browsers (Edge, Chrome, Firefox) for responsive web apps.  Optional native Windows apps increase development time and cost.  Development must guarantee browser and OS compatibility and responsive UI.  Skills needed: HTML, CSS, JavaScript (for web), C#/.NET for native apps.  Licensing costs may apply for paid development tools (Visual Studio). | Players access the game via mobile browsers (Safari on iOS, Chrome on Android) or native apps.  Web apps must be fully responsive and touch friendly.  Native development requires separate Android and iOS codebases unless using cross-platform frameworks (Flutter, React Native).  Skills needed: responsive design, mobile UI/UX, optional Flutter or React Native.  App-store fees: Apple ($99/year), Google Play ($25 one-time). |
| Development Tools | Languages/Tools: Swift/Objective-C for native apps, HTML/CSS/JavaScript for web apps, Xcode IDE for iOS/macOS, Visual Studio Code or IntelliJ for web development.  Team Impact: Development team may need iOS/macOS specialists for native apps, plus web developers for responsive front-end.  Licensing Costs: Xcode is free; Apple Developer Program $99/year for app distribution; Mac hardware required. | Languages/Tools: HTML/CSS/JavaScript (React, Vue) for web front-end; Node.js, Python, Java, or .NET Core for backend; IDEs like VS Code, IntelliJ, Eclipse; Docker/Kubernetes for deployment.  Team Impact: Web developers and backend engineers needed; DevOps team recommended for containerization and cloud deployment.  Licensing Costs: Mostly open-source tools, so minimal cost; optional enterprise support if desired. | Languages/Tools: C#/.NET for native apps, HTML/CSS/JavaScript for web apps; Visual Studio IDE (Community, Professional, or Enterprise), VS Code, or JetBrains Rider; IIS for hosting.  Team Impact: Backend and front-end developers needed; additional specialists for native Windows apps if chosen.  Licensing Costs: Visual Studio Professional/Enterprise requires paid licenses; Community Edition is free; Windows Server licensing may apply. | Languages/Tools: Swift (iOS) and Java/Kotlin (Android) for native apps; HTML/CSS/JavaScript for responsive web apps; cross-platform frameworks like Flutter or React Native for single codebase. Emulators and physical devices for testing.  Team Impact: Mobile developers or cross-platform specialists required; testing team needed for multiple devices and screen sizes.  Licensing Costs: Android Studio is free; Apple Developer Program $99/year; Google Play registration $25 one-time; device hardware costs |

## 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**: A Linux-based operating platform is the best foundation for expanding Draw It or Lose It into new computing environments. Linux is known for its reliability, performance, and scalability, which are essential for supporting a growing player base. Its compatibility with modern cloud infrastructures enables flexible deployment and consistent performance across desktop, web, and mobile platforms. Because it is open-source, Linux also reduces licensing costs while offering strong community and enterprise support. This platform provides a secure, efficient, and future-ready environment for continued development and expansion.
2. **Operating Systems Architectures**: Linux’s is layered and modular, a kernel managing essential tasks like memory, processes, and hardware access, while a separate user space runs applications and libraries. This separation ensures stability, security, and efficient resource use, which is critical for a real-time game. By leveraging containerization (Docker) and orchestration (Kubernetes), Linux allows services to run in isolated environments, scale dynamically, and deploy consistently across multiple computing environments, providing a robust foundation for growth and cross-platform compatibility.
3. **Storage Management**: A layered storage strategy on Linux ensures both performance and reliability. PostgreSQL manages persistent data such as user accounts, scores, and leaderboards, providing strong consistency and durability. Redis handles transient game state and session data, enabling low-latency updates for real-time gameplay. Static assets, including images and media, are stored in object storage and delivered via a CDN, reducing load times and server strain. Coupled with containerized services and persistent volumes, this approach allows the system to scale efficiently, maintain high availability, and support seamless cross-platform access as the game grows.
4. **Memory Management**: Linux handles memory through virtual memory, paging, and process isolation. This provides a stable foundation for Draw It or Lose It. In containerized deployments, cgroups enforce strict memory limits per service, preventing any single process from monopolizing resources and ensuring consistent performance. Redis stores ephemeral game state in memory, enabling fast, real-time updates while reducing database load. This combination of OS-level management and containerized controls allows the system to scale dynamically, maintain responsiveness under heavy player load, and deliver a seamless experience across multiple platforms.
5. **Distributed Systems and Networks**: To make Draw It or Lose It work across different platforms, the game uses a distributed system. The main parts (like login, matchmaking, real-time drawing, and leaderboards) run on Linux servers with Docker and Kubernetes. This ensures that each part can scale and stay reliable. Services communicate through APIs or a message bus, and WebSockets keep gameplay updates happening in real time for all players. Temporary game data is stored in Redis for fast access, and permanent data is saved in PostgreSQL. The system also includes load balancing, multi-region servers, and failover features, while clients can reconnect automatically if their connection drops. This setup ensures that the game is stable, responsive, and enjoyable for everyone, even when lots of people are playing at the same time.
6. **Security**: Keeping player information safe is a key priority. On Linux servers, sensitive data like user accounts, passwords, and game scores are protected with encryption at rest and in transit using HTTPS/SSL. Along with this, the system should also use strong authentication methods, including hashed passwords, multi-factor authentication (MFA), and role-based access controls, so only authorized users can access critical information. Firewalls, regular updates, and monitoring tools help prevent attacks and vulnerabilities. For real-time gameplay, WebSocket connections are encrypted, and temporary game data in Redis is secured to avoid exposing sensitive information. By combining these measures with Linux’s built-in security features, the game can provide a safe, reliable, and secure experience for all players, no matter the device or platform.