

# Draw It or Lose It

# **CS 230 Project Software Design**

Version 2.0

## Table of Contents

[**CS 230 Project Software Design** 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 |
| --- | --- | --- | --- |
| 2.0 | 11/28/2024 | Liu, Kevin | The Draw It or Lose It project involves developing a web-based, multi-platform game |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room requires the expansion of “Draw It or Lose it” into a web-based, multi-platform version. This upgrade is intended to reach a broader audience by allowing players to participate from various devices while preserving the interactive team-based gameplay that has contributed to the game’s popularity. The new web-based version will provide simultaneous access to players across different platforms, allowing them to engage in real-time gameplay from any device with internet access.

## Requirements

1. **Game Instances:** Only one game instance can exist in memory at a time.
2. **Teams and Players:** Multiple teams are allowed, and each team may consist of multiple players.
3. **Unique Names:** Each game, team, and player name must be unique to avoid conflicts.
4. **Real-time Gameplay:** The game must support synchronized interactions across platforms for smooth gameplay.
5. **Cross-platform Accessibility:** The application should work seamlessly across web, mobile, and desktop environments.

## [Design Constraints](#_2et92p0)

Multi-platform Compatibility: The game must run consistently across web, mobile, and desktop platforms. This requirement influences the choice of web frameworks, languages, and design principles, necessitating a responsive design and a backend that supports multiple device formats.

Unique Identifiers for Game Components: To meet the requirement of unique names for games, teams, and players, the application must use a robust system of unique identifiers. This system will need to be lightweight and efficient to maintain speed and scalability, especially as the user base grows.

Real-time Synchronization: With multiple users accessing the game in real-time, the application must handle concurrency and ensure that all players receive live updates. This requirement will impact the design of the backend, necessitating either a stateful server or an efficient, stateless communication protocol to handle live game data.

## [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 diagram illustrates the core classes of the Draw It or Lose It game application. At the top, the Entity superclass holds common attributes, such as id and name, which all entities in the application share. This means that every entity whether a Game, Team, or Player inherits these identifiers. The main entities are structured as follows. A game comprises multiple Teams, and each Team includes multiple Players.

The GameService class, associated with Game though composition, manages the game’s lifecycle and holds references to its instances. Similarly, Game is composed of multiple Teams, while each Team contains multiple Players. The ProgramDriver class, which houses the main function, serves as the application’s entry point, creating a singleton instance of GameService. This setup ensures there is only one GameService instance, which the ProgramDriver uses to add games, teams, and players. Additionally, ProgramDriver depends on SingletonTester, as noted by the <<uses>> relationship.

The UML diagram illustrates key object-oriented principles: inheritance, encapsulation, and abstraction. Inheritance allows Game, Team, and Player to extend Entity and inherit its attributes, reducing redundancy. Encapsulation is demonstrated by GameService, which restricts direct access to its attributes, allowing interaction only through specific methods, ensuring data privacy and the singleton pattern. Abstraction is seen in the separation of roles among classes, keeping interactions focused on essential tasks while hiding unnecessary details.

**"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)

Linux is a powerful open-source operating system known for stability, security, and scalability, especially in server environments. Its compatibility with open-source development tools like Node.js and Docker makes it a versatile choice for web applications, providing cost-effective and high-performance options for backend hosting. However, it requires technical expertise to manage and maintain, which can be challenging for teams unfamiliar with its command-line interface.

MacOS offers a secure, UNIX-based environment with an intuitive interface, widely favored among developers and creative professionals. It’s especially useful for development teams working within the Apple ecosystem, as it integrates seamlessly with iOS development, allowing efficient testing and deployment of Apple-compatible versions. While macOS provides excellent performance, its high hardware costs and less flexible server capabilities make it less ideal for back-end hosting compared to Linux. Nevertheless, for development and design, macOS offers robust tools and a reliable environment, making it a valuable platform for creating and testing the game.

Windows provides high accessibility and compatibility across various software, which is valuable for end-user applications like games. Windows offers a familiar interface and broad support for development tools, making it easy for most users to navigate and understand. Its compatibility with Azure enables smooth cloud integration for scalable deployment, though it does come with higher licensing costs and potential security vulnerabilities compared to Linux.

iOS and Android are leading mobile platforms, ideal for gaming apps due to their wide reach and mobile-first ecosystems. Both platforms support Progressive Web Apps (PWAs), which allow the game to operate as a web-based game with a native-like feel on mobile devices. Android’s open-source flexibility offers broad device compatibility, while iOS’s tightly curated ecosystem provides strong security and seamless user experience. However, Android’s platform fragmentation can pose compatibility challenges, and iOS’s strict App Store guidelines may limit flexibility.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Reliable server capabilities but less common for hosting. Higher hardware costs. | Excellent for hosting web apps; widely used for servers due to scalability, security, and cost-effectiveness | Highly compatible with enterprise systems; Windows Server offers strong hosting capabilities but with significant costs. | Typically connects to backend servers; not directly relevant for hosting. |
| **Client Side** | Smooth user experience on macOS Safari and modern browsers. Supports responsive web designs. | Modern browsers ensure compatibility. Supports responsive design for web browsers and mobile. Ideal for development using open tools. | Modern browsers ensure compatibility. Supports responsive web apps for desktop users. | Must ensure compatibility with iOS and Android browsers. Responsive design for seamless mobile experience. |
| **Development Tools** | Xcode for Apple ecosystem; cross-platform tools like React Native also compatible. | Tools like Eclipse, Visual Studio Code, or IntelliJ IDEA. No licensing costs for open-source development tools. | Visual Studio IDE licensed or open tools like VS Code. Licensing fees for proprietary tools can increase costs. | Android Studio for Android, Xcode for iOS. Development for both platforms may require two teams with distinct skills. |

## Recommendations

1. **Operating Platform**: I recommend developing it as a Progressive Web Application (PWA) using a cross-platform framework like React for the frontend and Node.js for the backend. PWAs enable the application to function across various devices, while providing near-native performance on mobile devices. This approach avoids the need to create separate applications for each operating system, significantly reducing development and maintenance efforts while ensuring consistent user experience across platforms.
2. **Operating Systems Architectures**: For the chosen platform architecture, and Node.js together use a client-server model that supports distributed, web-based applications. Node.js, a JavaScript-based, asynchronous server environment, supports real-time data exchange through RESTful APIs and WebSockets, which are essential for interactive and engaging gameplay.
3. **Storage Management**: NoSQL databases are excellent for managing semi-structured data, which aligns well with the requirements for storing information on players, game instances, and teams. Firebase is optimized for real-time data synchronization, which makes it ideal for interactive games that require instant updates across multiple platforms.
4. **Memory Management**: Node.js handles memory through JavaScript’s built-in garbage collection, freeing memory that is no longer in use to prevent memory leaks. However, developers need to monitor server performance closely, given Node.js’s single-thread event loop. React relies on the browser’s memory management, which depends on the user's device, so using optimization techniques like lazy loading and memory-optimized component updates in React can reduce memory usage, especially for mobile devices with limited resources.
5. **Distributed Systems and Networks**: For real-time communication between devices, WebSockets are recommended to keep the game synchronized across platforms. WebSockets provide continuous, bidirectional communication between clients and server, enabling fast updates essential for interactive multiplayer games. Hosting the application and database on a cloud server, such as AWS or Google Cloud, ensures high availability and reduces latency by scaling to meet user demand automatically.
6. **Security**: Security is essential, particularly to protect user information and game data. I recommend using HTTPS with TLS (Transport Layer Security) to encrypt data during transmission and AES encryption for sensitive stored data. Implementing a robust authentication system, like OAuth 2.0, helps verify user identity, while Firebase Authentication can simplify multi-platform access. Additionally, ensuring that all data inputs are validated and sanitized prevents injection attacks and malicious inputs.