

# 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 4**](#_Toc115077324)

[**Evaluation 4**](#_Toc115077325)

[**Recommendations 8**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.3 | 06/19/2025 | Sheryl Michalski | Project Three |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room seeks to develop a web-based version of Draw It or Lose It, a game where teams compete to guess puzzles based on rendered images. The goal is to create a scalable and efficient application that ensures seamless multiplayer functionality while maintaining unique identifiers for game instances, teams, and players. By leveraging a structured software development approach, this document outlines a framework that meets the client’s requirements while addressing the technical challenges of implementation.

## Requirements

The Draw It or Lose It game application must support multiple teams, each composed of various players, ensuring a smooth multiplayer experience. Game and team names must be unique to prevent duplication and facilitate seamless player onboarding. The game consists of four rounds, each lasting one minute, with images rendered progressively as clues for guessing. If a team fails to solve the puzzle, others can submit a single guess within a 15-second limit. The application must be web-based and capable of handling multiple concurrent users to ensure an engaging and scalable platform. Technically, only one game instance can exist in memory at any given time, requiring robust management of unique identifiers for games, teams, and players. A database system should store team and player data efficiently while enforcing uniqueness checks. Additionally, performance must be optimized for minimal latency, ensuring real-time gameplay and smooth image rendering. Security is also paramount, requiring protection of user data and enforcement of access controls.

## [Design Constraints](#_2et92p0)

Developing a web-based game application comes with several constraints:

1. **Multiplayer Support & Team Management:** The system must allow multiple teams with multiple players while ensuring a smooth user experience and real-time interactions.
2. **Unique Identifiers:** Game, team, and player names must be unique, requiring robust database management and validation logic.
3. **Single Instance Memory Management:** Only one instance of the game can exist at any given time, necessitating careful resource allocation and synchronization techniques to prevent conflicts.
4. **Web-Based Distribution:** The game must function in a distributed environment, requiring compatibility with various devices, browsers, and network conditions.
5. **Scalability & Performance:** The application must handle concurrent users efficiently while ensuring responsiveness and minimal latency.
6. **Security & Data Integrity:** Measures must be taken to prevent unauthorized access, ensure fairness, and protect user data.

## [System Architecture View](#_ilbxbyevv6b6)

Please note: Provided at a later date

.

## [Domain Model](#_8h2ehzxfam4o)

The ProgramDriver Class contains the main method. ProgramDriver uses Directed Association with SingletonTester to test if there is already an instance of GameService. The entity class is the parent class of the Game, Team, and Player classes. Game, Team, and Player all inherit the Entity’s required attributes. A Player cannot have a Team, but a Team can have a Player. A Team cannot have a Game, but a Game can have a Team. A Game cannot have a GameService, but a GameService can have a game. Game Service must only have one instance of each game running at any time. Each Game can only have one unique Team at any time. Each team can only have one player at a time.

**"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)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | A Unix-based system with strong security and efficiency. Stable performance, built-in development tools, and good security. Limited support for enterprise hosting, fewer cloud native tools | Open source, highly customizable, widely used for servers. High scalability, robust security, a large support community, and flexible configurations. Steeper learning curve, required advanced knowledge for optimization. | User-friendly, widely used in enterprise environments. Strong support for .NET applications, compatibility with Microsoft services, and integrated GUI-based tools. Less flexible than Linux, potential licensing costs, and security concerns if not managed properly. | Typically hosted via cloud-based services rather than on a device. Allows access from anywhere, supports mobile-friendly apps. Limited processing power, dependent on network connectivity, not ideal for hosting. |
| **Client Side** | Developing macOS applications typically requires access to Apple hardware and development tools, such as Xcode, which is free, but macOS devices can be expensive. Apple’s ecosystem is well-documented, but developing and optimizing for macOS may take time due to strict App Store guidelines. Developers need familiarity with Swift or Objective-C for native applications. Cross-platform frameworks like Electron or Flutter can help build macOS-compatible apps alongside other platforms. | Linux is open source, so there are no licensing fees, making development cost-effective. However, additional costs may arise from configuring development environments and maintaining compatibility. Development time depends on the distribution used; certain Linux distributions require more setup. Open-source libraries and extensive documentation help streamline development. Developers should be proficient in languages like Python, C++, or Java and familiar with Linux-based frameworks. Cross-platform compatibility is manageable with web-based applications or containerization solutions like Docker. | Windows development may involve licensing fees for specific software tools, but Microsoft's development ecosystem (Visual Studio, . .NET) offers free and paid tiers. Developing Windows can be efficient, thanks to extensive support and integrated development environments like Visual Studio. Windows applications typically use C#, .NET, or C++ for native development. Web-based solutions can be developed using JavaScript frameworks or cross-platform tools. | Developing mobile applications requires different costs depending on the approach: native development requires platform-specific tools, while cross-platform solutions (Flutter, React Native) can streamline costs. Native app development can take longer, especially if developing separately for iOS and Android. Cross-platform frameworks can reduce development time. Developers need skills in Swift (iOS) and Kotlin/Java (Android) for native development. Cross-platform frameworks (React Native, Flutter) allow for simultaneous development across both platforms. |
| **Development Tools** | **Programming Languages:** Swift (for macOS apps), Objective-C, JavaScript (for web applications), Python, C++.  **IDEs & Tools:** Xcode (Apple’s primary development environment), Visual Studio Code, Electron (for cross-platform desktop apps), Docker (for containerized deployment), Node.js (for backend development).  **Frameworks:** SwiftUI, React, Angular (for web front-end | **Programming Languages:** Python, Java, JavaScript, C++, Rust, PHP  **IDEs & Tools:** Visual Studio Code, Eclipse, JetBrains IntelliJ IDEA, Vim, GNU Emacs, Docker (for containerized development), Apache/Nginx (for hosting)  **Frameworks:** Django, Flask, Node.js, React, Angular, Spring (for enterprise applications) | **Programming Languages:** C#, JavaScript, Python, C++, Java  **IDEs & Tools:** Visual Studio, Visual Studio Code, JetBrains IntelliJ IDEA, Unity (for game development), Docker, Node.js, IIS (for hosting)  **Frameworks:** .NET, ASP.NET, React, Angular, Electron (for desktop applications) | **Programming Languages:** Swift (iOS), Kotlin (Android), Java, Dart (for Flutter), JavaScript (for hybrid apps)  **IDEs & Tools:** Xcode (for iOS development), Android Studio (for Android), React Native, Flutter (for cross-platform apps), Firebase (for backend services)  **Frameworks:** SwiftUI, Jetpack Compose (Android), React Native, Flutter |

## Recommendations

Operating Platform

A cloud-based platform such as Amazon Web Services (AWS), Microsoft Azure, or Google Cloud Platform (GCP) are recommended to support expansion across multiple computing environments. Cloud platforms provide scalability, cross-platform compatibility, and reliable hosting for web applications while allowing access from various devices, including desktop and mobile.

Operating Systems Architectures

The recommended platform should support Linux-based hosting, offering high security, stability, and scalability for web applications. Linux is widely used for server environments, ensuring smooth integration with containerization tools like Docker and Kubernetes. The game’s frontend can be developed using React or Angular, while the backend can leverage Node.js, Django, or Flask to support smooth functionality.

Storage Management

A distributed storage system using cloud-based databases such as Amazon RDS, Firebase Firestore, or MongoDB Atlas would be ideal. These systems ensure data integrity, real-time synchronization, and efficient handling of user accounts, game sessions, and team structures. Object storage solutions like AWS S3 can manage game-related media assets and ensure accessibility.

Memory Management

To optimize real-time gameplay, the cloud environment should use virtual memory management and caching strategies, such as Redis or Memcached. These tools help reduce latency in image rendering while maintaining smooth performance. The game engine should also use garbage collection techniques to free up memory when objects are no longer needed.

Distributed Systems and Networks

A microservices architecture is recommended for seamless communication between platforms. Each service (user management, game logic, team tracking) can function independently, reducing dependencies and improving reliability. Load balancers ensure even traffic distribution across servers, while CDNs (Content Delivery Networks) minimize latency issues for players worldwide. APIs should facilitate real-time interactions and cross-platform connectivity.

Security

End-to-end encryption should be implemented using HTTPS and TLS protocols to protect user information. Authentication and authorization mechanisms such as OAuth 2.0, JWT (JSON Web Tokens), and Multi-Factor Authentication (MFA) will safeguard accounts. Additionally, firewalls, intrusion detection systems (IDS), and automated security scans should be in place to prevent cyber threats.