

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 | 01/26/2025 | Jimmy Itela | Executive summary and Design Constraints |
| 3.0 | 02/23/2025 | Jimmy Itela | Final Project |

**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)

This software design document outlines the proposed solution for creating a web-based game application, "Draw It or Lose It," developed for our client, The Gaming Room. Inspired by the classic TV game show Win, Lose, or Draw, the game challenges teams to compete in guessing what is being drawn. The solution incorporates a library of stock images as visual prompts, enabling multiple teams to engage in up to four rounds of gameplay. Transitioning the game to a web-based platform ensures accessibility across various devices and enhances the overall gaming experience. This design aims to meet the client's requirements while delivering an intuitive and engaging application for users.

## Requirements

1. Web-Based Accessibility: The game must be accessible via a web-based platform, ensuring compatibility with multiple devices and operating systems, including desktops, tablets, and smartphones.

2. Multi-Team Support: Each game must support one or more teams, with the ability to assign multiple players to each team.

3. Unique Identification: Game, team, and player names must be unique to prevent conflicts and enhance user experience during game setup and gameplay.

4. Single Instance Enforcement: The application must allow only one active instance of a game in memory at any given time, implemented through unique identifiers for games, teams, and players.

5. Time-Limited Gameplay: Game rounds must adhere to specific time limits, such as one minute per round. Drawings should progressively reveal details, with the final drawing completed at the 30-second mark.

6. Second-Chance Mechanism: If a team fails to solve the puzzle within the allotted time, other teams must each have the opportunity to guess the solution within a 15-second window.

7. Real-Time Interaction: The application must provide real-time interaction for players, ensuring smooth and seamless communication between teams, players, and the game.

8. User-Friendly Interface: The system should feature an intuitive and easy-to-navigate interface, allowing players of varying technical skills to participate without difficulty.

9. Cross-Browser Support: The game must be compatible with all major web browsers (e.g., Chrome, Firefox, Safari, Edge) to ensure universal access.

## [Design Constraints](#_2et92p0)

1. Web-Based Distributed Environment:

The game must operate on a web-based platform, addressing challenges related to network communication, security, and compatibility across multiple web browsers and devices.

2. Cross-Browser and Device Compatibility:

The application must ensure seamless functionality on all major web browsers (e.g., Chrome, Firefox, Safari, Edge) and various devices, including desktops, tablets, and smartphones.

3. Unique Identifiers:

The system must enforce the uniqueness of game, team, and player names to avoid naming conflicts and provide a smooth user experience during the creation and joining of games.

4. Single Instance Limitation:

To ensure efficient resource management and functionality, the system must restrict the game service to only one active instance in memory at any given time.

5. Performance and Scalability:

The system must be optimized for high performance and designed to handle increasing user demands, including simultaneous gameplay sessions without performance degradation.

## [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 below illustrates the classes within the "Draw It or Lose It" game application. It employs object-oriented design principles to represent the relationships and functionalities of the system.

Entity Class:

The Entity class serves as the base superclass for all entities in the application. It defines shared attributes such as id and name, which are inherited by all subclasses. This ensures that every entity in the system has a unique identifier and a name. The Game, Team, and Player classes are subclasses of Entity, inheriting these attributes and extending them with additional functionality.

GameService Class:

The GameService class manages the lifecycle of the application, including creating and handling game instances. It uses a composition relationship with the Game class, maintaining references to multiple game objects. Similarly, the Game class holds references to multiple Team objects, and the Team class holds references to multiple Player objects. This layered composition ensures a clear hierarchy of ownership and responsibility.

ProgramDriver Class:

The ProgramDriver class acts as the entry point for the application. Within it, the singleton instance of GameService is created, ensuring that only one instance of GameService exists throughout the application’s runtime. This class is also responsible for adding games, teams, and players by interacting with the GameService instance.

SingletonTester Class:

The SingletonTester class tests the implementation of the singleton pattern for the GameService class. This ensures that only one GameService instance is active, providing consistent behavior during application execution.

Object-Oriented Programming (OOP) Principles:

This UML diagram highlights the use of several OOP principles:

1. Inheritance:

The Entity superclass allows subclasses (Game, Team, Player) to inherit shared attributes and methods, reducing redundancy and ensuring consistent functionality across entities.

2. Encapsulation:

The GameService class encapsulates its attributes, such as the list of active games, and exposes only the necessary methods for interaction. This ensures data integrity and abstraction.

3. Abstraction:

The methods in the Game, Team, and Player classes hide unnecessary implementation details, focusing on interactions that are meaningful to the application.

4. Composition:

Relationships such as Game containing Team objects and Team containing Player objects demonstrate the principle of composition. These relationships ensure the modularity and scalability of the design.

5. Singleton Pattern:

The GameService class follows the singleton design pattern, ensuring that only one instance manages the entire application’s game lifecycle.

---

UML Design Highlights

Composition Relationships:

GameService → Game: A GameService instance manages multiple games.

Game → Team: A game contains multiple teams.

Team → Player: A team is composed of multiple players.

Dependency Relationships:

The ProgramDriver class depends on the SingletonTester class to validate the singleton behavior of the GameService instance.

By showcasing the relationships between these classes, the diagram emphasizes simplicity, reusability, and scalability. Each class focuses on its core responsibilities, avoiding unnecessary complexity while ensuring efficient functionality.

**"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** | Unix-based providing a stable and secure foundation for hosting web applications. Higher hardware cost. | Open-source and highly customizable, offering flexibility and scalability for diverse use cases. Great choice for Server based. | Broad compatibility with a wide range of software and hardware, supported by a strong developer ecosystem. Less stable in comparison to Unix-based alternatives. | Optimized for portability and touchscreen-based interactions. Must account for varied screen sizes and hardware capabilities. |
| **Client Side** | Intuitive and user-friendly interface simplifies the learning curve for end-users. | Free to use and distribute but requires diverse expertise due to steep learning curves in GUI development. Greater flexibility in client customization due to its open-source nature. | Licensing costs may be higher compared to open-source platforms. Familiarity for most users and developers but limited cross-platform compatibility. | Essential considerations include responsive design and native feature integration. Must be optimized for native features. |
| **Development Tools** | Popular tools include Node.js, JavaScript, and IDE’s like XCOde and VSCode. Strong support for web and app development in the Apple ecosystem. | Offers a rich ecosystem with tools like VSCode, Atom, and Sublime Text, alongside advanced CLI tools. Supports a wide range of package managers. | C# and .NET Frameworks dominate, along with IDE’s like Visual Studio and JetBrains. Strong compatibility with enterprise-level tools. | Tools such as Kotlin, Swift, Objective-C, Android Studio, and XCode offer powerful capabilities for mobile. Simulators and emulators play a critical role in testing across multiple devices. |

## 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**: To enable the Gaming Room to expand Draw It or Lose it to other computing environments, adopting a web-based operating platform is highly recommended. This approach allows the game to be accessible and playable on various devices, including desktops, laptops, tablets, and smartphones, via web browsers. By leveraging modern web technologies, such as HTML, CSS, and JavaScript, the game can reach a wider audience and deliver a consistent experience across all supported platforms.
2. **Operating Systems Architectures**: The proposed web-based architecture should adopt a client-server model, emphasizing scalability and efficiency. Use HTML, CSS, and JavaScript to deliver a responsive and interactive user interface, ensuring smooth gameplay client side. Implement a multi-tier architecture, separating the presentation layer, application logic, and data management to enhance scalability and maintainability. Backend frameworks like Node.js, Django, or .NET Core can be used to manage requests efficiently on the Server Side.
3. **Storage Management**: A robust storage management system storage management system combining a relational database management system and cloud-based storage is recommended. Use systems like MySQL, PostgreSQL, or MongoDB to store structured data, including game progress, user profiles, and statistics. Utilize could platforms like AWS to store large media assets.
4. **Memory Management**: Modern web browsers manage memory efficiently through automatic garbage collection, minimizing memory leaks and optimizing performance. The application should take advantage of browser-based memory management techniques to handle dynamic data allocation and deallocation seamlessly.
5. **Distributed Systems and Networks**: To enable seamless communication between players across various platforms, a distributed software architecture should be implemented. Use a central server or cloud-based solution to handle real-time synchronization of game states and player interactions. Address connectivity issues by incorporating mechanisms like message queuing, reconnection protocols, and data synchronization during outages or bandwidth limitations.
6. **Security**: Protecting user data is paramount. Use robust authentication systems to secure user access. Role-based access control can further protect sensitive game data. Employ encryption methods such as HTTPS to secure data in transit, and use AES encryption for data stored in databases. Also implement protocols to prevent common vulnerabilities like SQL injection. Regular security audits should be conducted to identify and address potential threats.