

<Name-of-Software-Application>

# **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 | 04/16/2025 | Harsh Patel | Recommendations |

**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 Room has requested the development of a web-based game application, "Draw It or Lose It," where teams compete to guess puzzles based on drawings rendered in real-time. The game has four rounds, each lasting one minute, with drawings becoming fully visible at the 30-second mark. To meet the client’s requirements, the application must allow one or more teams, each with multiple players, while ensuring that game, team, and player names are unique. The game must also ensure that only one instance exists in memory at any given time, allowing efficient memory usage and smooth operation.

Our proposed solution involves designing a scalable, distributed, web-based architecture that ensures uniqueness through identifiers and centralized control to manage instances. By using modern web technologies like cloud services for scalability, a relational database for tracking team/player names, and session management for handling game instances, we can develop a robust and efficient game system. The goal is to streamline the development process and ensure seamless gameplay, preparing the foundation for future hardware decisions.

## Requirements

**Business Requirements:**

1. **Multi-team Gameplay**: The game should support one or more teams, with each team consisting of multiple players, to enable competitive gameplay.
2. **Unique Names**: Game, team, and player names must be unique to prevent confusion and ensure smooth user experience.
3. **Single Game Instance**: Only one instance of the game should exist in memory at any given time to avoid conflicts and resource overload.
4. **Round-based Gameplay**: The game must consist of four rounds, with each round lasting one minute. Drawings must be progressively revealed, being fully visible by the 30-second mark.
5. **Turn-based Guessing**: If a team fails to guess the puzzle, other teams should have a 15-second opportunity to submit one guess each to solve the puzzle.

**Technical Requirements:**

1. **Web-based Distributed System**: The game must be accessible via a web application, allowing multiple users to participate simultaneously over the internet.
2. **Real-time Updates**: Drawings must be rendered in real-time with steady progression, requiring low-latency communication to ensure all participants see the clues at the same time.
3. **Unique Identifier Management**: A system for generating and validating unique identifiers for games, teams, and players must be implemented to ensure uniqueness.
4. **Scalable Infrastructure**: The system must scale to accommodate varying numbers of players and teams, requiring efficient resource management and server-side scaling capabilities.
5. **Memory Management**: Efficient memory handling is crucial to ensure that only one game instance is active at any given time, preventing memory leaks and unnecessary resource usage.

By addressing these requirements, the game can be developed to meet both functional needs and performance expectations.

## [Design Constraints](#_2et92p0)

1. **Web-Based Distributed Environment**
   * The game must be accessible to users over the web and support multiple concurrent players and teams. This requires the development of a distributed architecture that can scale with user demand.
   * **Implication**: Distributed systems introduce complexities in synchronization and data consistency. We must ensure that game states are reliably updated across all users in real-time. WebSocket technology or similar protocols will be needed for live, low-latency updates during gameplay.
2. **Uniqueness of Names (Game, Team, Player)**
   * Each game, team, and player must have unique identifiers to avoid conflicts. This requires checking a centralized database or in-memory data structure to verify uniqueness in real time.
   * **Implication**: Using a relational database or NoSQL solution to manage unique identifiers will be critical. Ensuring that name uniqueness checks are efficient is key to preventing delays during team or player creation.
3. **Single Instance in Memory**
   * Only one instance of a game can exist in memory at a time, preventing multiple concurrent games from running and potentially causing confusion or resource overload.
   * **Implication**: Implementing a singleton design pattern at the game instance level ensures that only one instance is active at any given time. This will require careful session and memory management to avoid resource leaks and maintain optimal performance.
4. **Real-Time Rendering**
   * Drawings will be rendered progressively during each round, with the full image available at the 30-second mark. This requires precise timing and real-time rendering capabilities.
   * **Implication**: To maintain real-time performance, we may need to use efficient image-rendering libraries or services that ensure smooth and timely drawing transitions, possibly with caching strategies to handle network delays.
5. **Scalability and Performance**
   * The application must scale to accommodate potentially large numbers of teams and players simultaneously.
   * **Implication**: We will need to design the system to leverage cloud services or load balancers to dynamically allocate resources based on player load. This will ensure that the game remains responsive even with high user traffic.

These constraints require careful consideration of web technologies, database management, and real-time synchronization to deliver a smooth, scalable experience for "Draw It or Lose It."

## [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 UML class diagram for "The Gaming Room" illustrates the structure and relationships between the core entities of the game application. Here's a breakdown of the key classes and their relationships, along with the object-oriented programming principles they demonstrate:

**1. Entity Class**

* The Entity class is a base class that provides common attributes (id and name) and behaviors (getId(), getName(), toString()) for other classes to inherit.
* **Object-Oriented Principle**: **Inheritance** is used here, as the Game, Team, and Player classes extend the Entity class, which allows these classes to reuse common properties and methods without duplicating code. This helps fulfill the requirement for managing unique identifiers and names across games, teams, and players.

**2. Game Class**

* The Game class represents the overall game. It has a list of teams (List<Team>), meaning a game can have multiple teams.
* It includes methods to add teams (addTeam()) and convert the game object to a string (toString()).
* **Object-Oriented Principle**: **Composition** is demonstrated as a game "has-a" list of teams, indicating that a game is composed of one or more teams. This structure helps fulfill the requirement of having multiple teams in a game.

**3. Team Class**

* The Team class represents a team in the game. It has a list of players (List<Player>), indicating that a team can have multiple players.
* It includes methods to add players (addPlayer()) and convert the team object to a string (toString()).
* **Object-Oriented Principle**: Similar to the Game class, **composition** is also present in the Team class, as a team "has-a" list of players. This aligns with the requirement that teams consist of multiple players.

**4. Player Class**

* The Player class represents a player in the game. It inherits from the Entity class, meaning each player has an id and a name.
* It includes a toString() method for converting player objects to strings.

**5. GameService Class**

* The GameService class is responsible for managing the overall game state, including a list of games (List<Game>), as well as tracking unique identifiers for games, teams, and players through nextGameId, nextPlayerId, and nextTeamId.
* It includes methods to add a game (addGame()), get a game by ID (getGame()), retrieve counts of games (getGameCount()), and manage identifiers (getNextPlayerId(), getNextTeamId()).
* The class follows the **Singleton** design pattern, as indicated by the getInstance() method, ensuring that only one instance of GameService exists at any given time. This aligns with the requirement that only one instance of the game can exist in memory.

**6. ProgramDriver and SingletonTester**

* The ProgramDriver class contains the main() method, serving as the entry point for the application.
* The SingletonTester class tests the Singleton behavior of the GameService class to ensure only one instance is active during runtime.

**Object-Oriented Principles Highlighted:**

* **Inheritance**: The Game, Team, and Player classes inherit from the Entity class, promoting code reusability by sharing common attributes (id and name).
* **Composition**: Both Game and Team classes are composed of other objects (Team and Player), reflecting real-world relationships and fulfilling the requirement of managing multiple teams and players.
* **Singleton Design Pattern**: The GameService class follows the Singleton pattern, ensuring that only one instance exists, which is essential to managing the game and fulfilling the requirement that only one game instance is in memory.

**"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 can host web-based applications through environments like Apache or Nginx, but they are less commonly used for server hosting due to higher cost and compatibility issues with server-based software. While MacOS provides a Unix-like base, similar to Linux, it isn't as customizable for server needs. Macs also tend to be more expensive to scale in a server environment. | Linux is the most popular OS for hosting web-based applications due to its flexibility, scalability, and open-source nature. It supports a wide range of server software like Apache, Nginx, and various databases (MySQL, PostgreSQL). Linux distributions like Ubuntu and CentOS are often used in server environments because of their security, performance, and customizability, along with the ability to run on almost any hardware, making it ideal for cloud services and web hosting. | Windows Server is a robust option for hosting web applications, especially if the software relies on Microsoft technologies like ASP.NET or SQL Server. Windows offers great support for enterprise environments and comes with easy-to-use GUI-based tools. However, it can be more expensive due to licensing costs, and it lacks the same flexibility and open-source benefits as Linux. Windows servers are also generally more resource-intensive | Mobile devices generally aren't used to host web-based applications due to hardware limitations and battery constraints. However, they can be used to interface with cloud-based services for server-side logic, especially through mobile apps. For small-scale testing or edge cases, mobile devices can act as lightweight servers but are not suited for large-scale deployment or continuous hosting. |
| **Client Side** | Mac development for client-side applications requires a focus on the macOS ecosystem and Apple's design guidelines. Mac systems are favored for creative industries and software that requires high-performance graphics, but developing for multiple platforms (Windows, Linux) can be time-consuming and expensive due to platform-specific requirements. Developers may need expertise in Swift and Objective-C for native Mac applications. | Linux offers a highly customizable development environment for client-side applications, though it may not be ideal for consumer-oriented software that needs a polished graphical interface like on Mac or Windows. Linux has strong support for cross-platform development through tools like GTK and Qt, but market share for Linux desktops is small. Development time and cost may be higher when considering compatibility with other OSs. | Windows dominates the desktop market, making it essential for client-side application development. The platform offers a wide range of tools (like Visual Studio) and frameworks (like .NET), making it easier and faster to develop. However, ensuring compatibility across different versions of Windows can increase development complexity. Expertise in C#, .NET, and other Windows-specific technologies is typically required. | Developing for mobile devices requires cross-platform frameworks like React Native or Flutter, as well as platform-specific languages (Swift for iOS, Kotlin/Java for Android). The main challenges include optimizing for various screen sizes and managing app store requirements. Development time can vary depending on whether a native or cross-platform approach is used, and costs can be high due to the need for different expertise and tools for iOS and Android. |
| **Development Tools** | For Mac, developers use Xcode, Apple's IDE for macOS and iOS development, which supports languages like Swift and Objective-C. Mac is also compatible with tools like Sublime Text, JetBrains, and Docker, offering versatility in development. However, Xcode is required for Apple platform deployment, which can be limiting if developers are working on multiple platforms. | Linux supports a wide range of programming languages (C, C++, Python, JavaScript) and development tools (Vim, Emacs, Eclipse). It is favored by developers who want full control over their environment and flexibility in terms of scripting and automation. Linux also integrates easily with open-source CI/CD tools and Docker for containerized development and deployment. | Windows supports a wide variety of development tools, including Visual Studio, a powerful IDE for .NET and C# development. It also works well with other tools like Eclipse, Atom, and Docker. Windows is known for ease of integration with Microsoft-based technologies and enterprise tools, though developers need to manage its higher resource usage. | Mobile development tools vary by platform. iOS development requires Xcode and Swift, while Android development uses Android Studio with Java or Kotlin. Cross-platform tools like Flutter and React Native enable developers to build apps for both platforms from a single codebase, saving time and reducing maintenance costs. These tools are essential for ensuring apps run smoothly on both iOS and Android. |

## 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**: For The Gaming Room's "Draw It or Lose It" expansion, Linux is the most suitable operating platform. Its scalability, low cost, and support for a wide variety of open-source tools make it the ideal choice for server-side hosting. Linux also integrates well with cloud platforms and distributed systems, which is important as the game expands to multiple environments.
2. **Operating Systems Architectures**: Linux uses a monolithic kernel architecture, which provides high performance and low latency. It is known for its stability and scalability, which is crucial for hosting multiplayer games with heavy network traffic. Linux also allows for easy customization, enabling The Gaming Room to optimize its server environment for gaming performance.
3. **Storage Management**: For Linux, a flexible and reliable storage management system such as ZFS or EXT4 would be appropriate. These file systems provide robust data integrity and support for large-scale storage needs, including snapshotting and compression. These features ensure that game data, player information, and other critical assets are safely managed and can be scaled as needed.
4. **Memory Management**: Linux employs a sophisticated memory management system that uses both virtual memory and physical RAM efficiently. The OS ensures that memory is allocated dynamically as needed, with support for swapping and paging. This is crucial for a game like "Draw It or Lose It," which may require large amounts of memory for handling multiple users simultaneously. Linux’s ability to prioritize processes and allocate resources dynamically ensures smooth operation even under heavy load.
5. **Distributed Systems and Networks**: Linux excels in distributed systems due to its compatibility with containerization technologies like Docker and Kubernetes, which make it easy to scale services across multiple platforms. To enable "Draw It or Lose It" to communicate between platforms, The Gaming Room could utilize REST APIs and WebSockets to maintain real-time connectivity. Linux's strong support for network protocols and fault tolerance ensures reliable communication even in cases of outages or connectivity issues.
6. **Security**: Linux is well-regarded for its robust security features, including built-in firewalls (such as iptables) and SELinux for access control. With regular updates and a large community of developers, Linux remains secure against vulnerabilities. The Gaming Room can implement encryption (TLS/SSL) to protect user data both at rest and in transit. Additionally, role-based access control and multi-factor authentication can be enforced to safeguard sensitive information across platforms.