

# The Gaming Room 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 | 07/18/2025 | Rehan Arshad | Initial submission with Executive Summary and design components. |

**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 is transitioning from a mobile-only game to a web-based platform to support broader accessibility and multi-platform functionality. The current game, Draw It or Lose It, is only available on Android, and the client now seeks to deploy the game across multiple systems, including web, desktop, and mobile environments.

To support this goal, a software solution has been designed using object-oriented programming principles and design patterns that ensure scalability, maintainability, and ease of development. The solution introduces a central Game Service that enforces the Singleton pattern, ensuring that only one game is managed at any given time. It also leverages the Iterator pattern to manage game, team, and player objects while preventing duplicate names.

This design will allow the application to efficiently manage teams and players, validate name uniqueness, and ensure consistent game state across all supported platforms.

## Requirements

1. **Team and Player Management**

* A game must support one or more teams.
* Each team must contain multiple players.

1. **Unique Identification**

* Game names and team names must be unique.
* Each game, team, and player must have a unique identifier.

1. **Game Instance Control**

* Only one instance of the game service should exist in memory at a time.
* A singleton design pattern must be used to enforce this constraint.

1. **Code Reusability and Maintainability**

* The solution should use object-oriented principles such as inheritance to reduce redundancy (i.e., common attributes like id and name should be handled in a base Entity class).

1. **Extensibility for Multi-Platform Support**

* The solution must be scalable to support future integration with different platforms (e.g., web, Android, iOS).

## [Design Constraints](#_2et92p0)

1. **Single Game Instance (Singleton Pattern)**  
   Only one instance of the game service can exist in memory at any given time. This constraint is necessary to maintain consistent state and control across all parts of the application. The Singleton design pattern is implemented to enforce this rule and support centralized management of game objects.
2. **Unique Identification**  
   Each game, team, and player must have a unique identifier (ID) and a unique name. This constraint ensures data integrity, avoids duplication, and allows users to validate names before assigning them to new teams or players.
3. **Platform Independence**  
   The application must be designed in a way that supports multiple platforms (web, desktop, mobile). This requires use of language features and technologies that are portable and maintain consistent behavior across environments.
4. **Memory Management**  
   Since the game runs in a potentially distributed environment with multiple concurrent users, efficient memory usage and object reuse is crucial. Singleton pattern and object pooling strategies are used to help manage memory footprint.
5. **Security and Scalability**  
   The solution must be scalable to accommodate additional players, teams, and game instances in future updates while maintaining proper access controls and secure object instantiation.

## [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)

**Classes and Relationships**

* **Entity (Base Class)**:
  + Contains common attributes id and name.
  + Acts as the superclass for Game, Team, and Player to avoid redundancy and promote code reuse (inheritance).
* **GameService (Singleton Class)**:
  + Manages the entire life cycle of games, including adding and retrieving games.
  + Uses the Singleton design pattern to ensure only one instance of this class exists in memory, which supports centralized game management.
* **Game (Extends Entity)**:
  + Represents a game session.
  + Contains a list of Team objects.
  + Prevents duplicate game names using the Iterator pattern during creation.
* **Team (Extends Entity)**:
  + Represents a team in the game.
  + Contains a list of Player objects.
  + Enforces unique team names using the Iterator pattern.
* **Player (Extends Entity)**:
  + Represents a participant assigned to a team.

**Object-Oriented Principles Demonstrated**

* **Inheritance**:
  + Game, Team, and Player all extend the Entity base class to share common fields and behaviors like id and name.
* **Encapsulation**:
  + Access to fields is controlled through getter methods (e.g., getName(), getId()), protecting direct access to internal state.
* **Singleton Pattern**:
  + Applied to GameService to ensure a single instance handles all game-related operations.
* **Iterator Pattern**:
  + Used in methods like addGame() and addTeam() to search existing collections and ensure name uniqueness.

This domain model aligns with the client's software requirements by promoting maintainability, reusability, and scalability.

**"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 is not commonly used as a server-side platform in production environments due to limited enterprise-level support and fewer hosting providers. However, macOS offers a Unix-based system that is stable and secure, with great tools for development. It is best suited for local development or staging environments rather than deployment. | Linux is the most widely used server-side platform for web applications. It offers robust performance, stability, open-source flexibility, and high security. Its command-line interface and broad compatibility with web servers like Apache and Nginx make it the preferred platform for hosting scalable and reliable web-based applications. | Windows servers provide strong compatibility with Microsoft technologies such as .NET, ASP.NET, and IIS. While Windows offers a user-friendly interface and wide enterprise use, it is often more expensive due to licensing costs and may require more system resources compared to Linux. | Mobile devices are not typically used as server hosts. However, they rely on server-side resources to function. When evaluating server-side development, it’s essential to ensure the hosted application supports mobile accessibility and communication through APIs or cloud services. |
| **Client Side** | Developing for Mac clients may require specific design considerations to match Apple’s ecosystem and UI standards. It often involves higher development costs due to the need for macOS devices and expertise with Apple development tools. However, Macs are well-regarded for creative and design-oriented tasks. | Linux client-side development is cost-effective but has a smaller user base. Developers need experience with desktop environments like GNOME or KDE, and software should be compatible across various Linux distributions. It is more commonly used by developers or power users than by mainstream consumers. | Windows dominates the client-side market share, especially in business environments. Development for Windows clients is well-supported, cost-effective, and benefits from a wide user base. However, compatibility with older versions and system diversity can add complexity. | Mobile client-side development requires cross-platform support for both Android and iOS. Development can be time-consuming due to varying screen sizes, operating system constraints, and performance optimization needs. Tools like Flutter or React Native help reduce overhead by enabling code reuse across platforms. |
| **Development Tools** | Developers on Mac can use tools such as Xcode, IntelliJ IDEA, Eclipse, and Visual Studio Code. macOS is required for native iOS development, which makes it essential for teams targeting the Apple ecosystem. Unix compatibility also enables terminal-based tooling similar to Linux. | Linux supports a wide array of development tools such as Eclipse, IntelliJ IDEA, VS Code, and command-line utilities. It is particularly developer-friendly for backend development and scripting. Most server-side applications are built and tested in a Linux environment. | Windows developers use tools like Visual Studio, Eclipse, IntelliJ IDEA, and VS Code. It supports a wide range of programming languages and is especially strong for developing desktop applications. Tools for .NET and C# are exclusive to Windows or best supported on it. | For mobile, Android Studio is used for Android, and Xcode is used for iOS. Cross-platform tools like Flutter, React Native, and Xamarin enable development for both platforms from a single codebase. Testing tools like emulators and cloud testing environments are also commonly used. |

## 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**: Use Linux as the primary server-side platform for The Gaming Room’s game, *Draw It or Lose It*. Linux is widely used in the tech industry because it is cost-effective, highly scalable, stable, and secure. It also supports a wide range of programming languages and web servers, making it ideal for hosting modern web applications like multiplayer games. Using Linux will help the game perform reliably and grow across different platforms.
2. **Operating Systems Architectures**: To support The Gaming Room’s goal of expanding *Draw It or Lose It* to multiple platforms, a cross-platform architecture should be used. This can be achieved through responsive web design and frameworks like Flutter or React Native, which allow one codebase to run on macOS, Windows, Linux, Android, and iOS. This approach ensures that users have consistent experience no matter what device they’re using, and it helps reduce development time and maintenance.
3. **Storage Management**: For *Draw It or Lose It*, I recommend using a cloud-based storage solution like AWS S3 or Google Cloud Storage. These services offer scalable and secure storage that can handle game data, user accounts, and player drawings across all platforms. Cloud storage also ensures that data is accessible in real time, regardless of the device being used, while keeping it protected and backed up.
4. **Memory Management**:To keep *Draw It or Lose It* running smoothly, especially on mobile devices with limited memory, it’s important to use lightweight frameworks and memory-efficient code. This includes using optimized data structures and algorithms that reduce memory usage. Managing memory carefully will help avoid lag, crashes, and slowdowns, which is critical for a game that needs to perform well across different devices.
5. **Distributed Systems and Networks**: Since Draw It or Lose It will be played across multiple platforms, it should be designed as a distributed system using a client-server model. The server will handle game logic and data, while each client (web or mobile app) connects to the server through a network. Communication should happen using RESTful APIs or WebSockets so that data like player scores and drawings stay updated in real time. To handle network issues or outages, the system should support reconnection and data recovery features, ensuring that users don’t lose progress if their internet connection is interrupted.
6. **Security**: Security is essential for protecting user data in *Draw It or Lose It*, especially since the game will be available online across different platforms. To keep user information safe, the system should use secure coding practices like input validation, encrypted communication (HTTPS), and strong user authentication. Role-based access control should also be implemented to make sure users only have access to the parts of the game they’re allowed to see. These steps will help prevent unauthorized access and keep sensitive information secure.