

The Gaming Room

# **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 |
| --- | --- | --- | --- |
| 2.0 | 08/02/2025 | Andriana Djurdjevic | Updated development requirements in the Evaluation Table for different Operating Systems |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room wants to bring their game Draw It or Lose It from Android to a web-based version that works across platforms. This project focuses on creating a solid software design that supports that goal.

To make sure everything runs smoothly, we’ll use the Singleton pattern to keep only one instance of the main game service running. We'll also use the Iterator pattern to make sure game, team, and player names stay unique. This setup will help keep things organized, efficient, and ready for future development.

## Requirements

Business Requirements:

* The game needs to work as a web app, not just on Android.
* It should support more than one team, with multiple players on each team.
* Game and team names must be unique so there’s no confusion.
* The system should be easy to update and grow over time.

Technical Requirements:

* Only one game should run in memory at a time.
* Each game, team, and player needs a unique ID.
* The game should run smoothly on different devices and platforms.
* The code should be efficient and manage memory well.

## [Design Constraints](#_2et92p0)

Since the game will be web-based and run on different platforms, we need to design it so it works across various systems and browsers. This means we can’t rely on platform-specific tools and should keep the system lightweight and flexible. Another constraint is only running one instance of the game at a time. To handle that, we will use the Singleton pattern to make sure only one GameServer is created in memory.

## [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 diagram below is used to provide a visual design of the game system. It outlines how the program will be developed. In the diagram, the Entity class creates a relationship between the Game, Team, and Player classes. The arrows show that each of these classes inherits shared attributes from the Entity superclass. We can also see the classes, variables, and methods that will be used during development. The ProgramDriver class in the top left corner points to the SingletonTester, showing that ProgramDriver will use SingletonTester to test the Singleton pattern. This is important because one of the key requirements is making sure only one instance of the game can exist in memory. The GameService class holds the main logic and methods that make up the core structure of the game and its functionality.

Another requirement is that each game, team, and player must have a unique name, which is why each of these classes is clearly defined and connected. The lines connecting the classes represent associations between them, and the numbers on those lines indicate how many objects can be associated. For example, GameService can manage zero or more Gameobjects, and each Game can have multiple Team objects, and so on. This diagram will help guide the development process and can be adjusted if needed along the way.

**"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** | Mac is known for its strong security and reliable performance but is not typically used as a server-side platform and is not widely used for large-scale hosting. Mac servers are also less customizable and tend to be more expensive. | Linux is the most common and cost-effective server platform. It’s open-source, highly scalable, supports Apache/NGINX, and has no licensing costs. Ideal for web-based hosting of high-traffic applications | Windows servers support IIS and integrate well with Microsoft tools. Licensing costs are higher, and performance/scalability can lag behind Linux. Still viable in enterprise settings. | Not applicable – mobile devices are not used for server hosting and do not support server-side infrastructure. |
| **Client Side** | Requires Apple hardware and knowledge of macOS. Apps are accessed through Safari, so testing is needed for Safari quirks. Modern browsers support responsive web apps using HTML/CSS/JS. | Linux clients will access the app through Firefox or Chrome. Requires responsive design and browser testing. Generally cost-effective but needs broader compatibility testing. | Windows clients will use browsers like Chrome, Edge, or Firefox. Browser support for responsive web apps is strong. Development is straightforward. | Mobile clients will access the app through Chrome or Safari. Must use responsive web design (e.g., media queries, flexible layout). Extra testing needed for touch controls and screen sizes. |
| **Development Tools** | Requires Xcode for iOS-specific testing, only available on macOS. Other tools: VS Code, IntelliJ. Languages: JavaScript, HTML/CSS, Java. Apple Developer Program needed for native iOS deployment. | Tools like VS Code, Eclipse, and IntelliJ are available. Languages: JavaScript, Python, Java, Bash. No licensing costs. Works well for backend and full-stack development. | Visual Studio, VS Code, and Eclipse are popular. Languages: JavaScript, Java, C#. Some tools like Visual Studio Pro may require licenses. Strong developer documentation. | For browser-based development, native tools (Android Studio, Xcode) are not needed. Cross-platform frameworks like React or Vue work well. Still requires mobile testing via emulators or real 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**: I would recommend using a Linus-based platform over other environments, as it is cost-effective, stable, highly scalable, and widely used for hosting web applications. It also works well with many cloud services and that makes it a great choice for building a web -based game that is accessed on multiple platforms.
2. **Operating Systems Architectures**: Linux uses a monolithic kernel architecture, which means that the core parts of the system can communicate directly and helps the system run smoother and more efficiently. It also supports multiple users and can run many tasks at once.
3. **Storage Management**: For storage, a good option to use is a cloud-based database like MySQL or PostgreSQL running on a Linux server. These systems are reliable and are easy to scale as the game grows.
4. **Memory Management**: Linux handles memory very efficiently by automatically allocating memory where it’s needed and reclaims unused memory to keep the system running smoothly.
5. **Distributed Systems and Networks**: To support communication across platforms, the game must use reliable cloud hosting and content delivery networks which can handle outages by rerouting site traffic.
6. **Security**: For security and to keep users safe, the game should use HTTPS encryption, user authentication, and secure data storage with encryption. Linux also has strong security tools which help against attacks and unauthorized access.