

<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 | 03/23/25 | Phillip Gooden | Initial draft for software design document |

## [Executive Summary](#_sbfa50wo7nsh)

I am working with The Gaming Room to expand Draw It or Lose It from an Android app to a web based game that can run on different platforms. The game must support multiple teams and players, ensure that names are unique, and limit the game service to only one instance in memory. I plan to use the singleton pattern for the main service and an iterator style approach to check for name uniqueness. I will also use a base class named Entity to share common attributes like id and name.

## Requirements

1. I need to handle multiple teams in a single game.
2. Each team can include many players.
3. Game, team, and player names must be unique.
4. There should only be one instance of the game service at any time.
5. Common attributes and behaviors will be placed in the Entity base class.

## [Design Constraints](#_2et92p0)

1. Scalability
   1. I need to design the system so it can handle many users at the same time without slowing down.
2. Reliability
   1. I want the system to keep running even when some parts fail or the network has problems.
3. Security
   1. I must protect player and team data through secure connections and proper data handling.
4. Singleton Constraint
   1. Only one instance of the game service can exist. The singleton pattern will enforce this.
5. Unique Names
   1. The system must confirm that names are not already used before creating new games, teams, or players.

## [System Architecture View](#_ilbxbyevv6b6)

I am not including any specific architecture diagrams here.

## [Domain Model](#_8h2ehzxfam4o)

The UML diagram for The Gaming Room's game application shows how object oriented principles can meet the client's needs in a straightforward way. Entity is a base class that Game, Team, and Player inherit from, which means they share attributes like id and name as well as methods like toString. Game contains Team objects, and Team contains Player objects, creating a clear hierarchy. The Singleton pattern in GameService guarantees only one instance runs, which simplifies game creation and ID generation. Encapsulation protects data by keeping lists private, while polymorphism lets classes customize toString. An iterator style approach checks for unique names for games and teams, which matches the client's requirement for uniqueness. Overall, this design is scalable, supports team player relationships, and helps maintain memory efficiency for a web based environment.

**"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 can host web apps but is less common in production. It is helpful for iOS dev. | Linux is very popular for hosting. It is open source and secure but requires expertise. | Windows Server is widely used in enterprises. It has strong Microsoft support but licenses are needed. | Hosting a full server on a mobile device is not realistic. Devices can be clients or partial hosts. |
| **Client Side** | Supporting Mac clients can be costly due to hardware expenses and possible licensing fees, though cross-platform tools can help lower expenses. Development time may increase if developers are new to Apple’s ecosystem, requiring expertise in macOS frameworks, Swift, or Objective-C. | Linux is often low cost due to free distributions and open-source tools, though enterprise support can add expenses. Setup and configuration demand knowledge of command-line utilities and shell scripting, which can shorten development time once mastered. | Windows may involve licensing fees, but tools like Visual Studio can streamline development for those familiar with the Microsoft ecosystem. Developers need expertise in .NET frameworks and Windows-specific APIs. | Mobile development extra costs include app store fees and specialized hardware, especially for iOS testing. Teams need proficiency in Swift, Kotlin, or cross-platform frameworks, and must account for platform-specific UI and performance constraints. |
| **Development Tools** | Programming languages like Swift and Objective-C for native applications and Java for web based projects, with IDEs such as Xcode, IntelliJ IDEA, or Eclipse. Additionally, tools like Homebrew and Terminal commands help manage dependencies and automate builds for smooth deployment. | Programming languages like Java, Python, or Node.js with IDEs such as IntelliJ IDEA, Eclipse, or Visual Studio Code to build and deploy software. Command line tools, package managers (apt, yum), and build systems like Maven or Gradle further streamline development and deployment. | Windows typically relies on Visual Studio for .NET applications and also use Java based IDEs like IntelliJ IDEA or Eclipse for web based projects. Tools such as PowerShell, NuGet, and MSBuild play key roles in dependency management and automating the build process on this platform. | Mobile typically relies on Android Studio for Android apps and Xcode for iOS apps, employing languages such as Java, Kotlin, Swift, or Objective-C. Cross platform frameworks like Flutter or React Native, along with IDEs like Visual Studio Code, facilitate faster development and smoother deployment across various mobile 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 recommend Linux for its stability, security, and widespread server support.
2. **Operating Systems Architectures**: Linux uses a modular design with the kernel handling processes and memory, while apps run in user space.
3. **Storage Management**: I suggest a database like PostgreSQL or MySQL. Both are reliable and work well on Linux.
4. **Memory Management**: Linux uses virtual memory and the JVM includes garbage collection. This approach reduces memory leaks.
5. **Distributed Systems and Networks**: I can use a web based API to communicate with different clients. If I need to handle more users, I can add more servers and load balancing.
6. **Security**: I should use HTTPS to encrypt data in transit, store sensitive information securely, and apply strong access controls. Linux includes tools like SELinux or AppArmor to help enforce security.