

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 |
| --- | --- | --- | --- |
| 1.0 | 09/23/2024 | Juan Rodriguez | Initial version of the software design document for The Gaming Room project. |
| 1.1 | 10/7/2024 | Juan Rodriguez | Revised document with additional recommendations for multi-platform deployment. |
| 1.2 | 10/19/2024 | Juan Rodriguez | Expanded recommendations to include detailed analysis of operating system architectures, storage management, memory management, distributed systems, and security for Project Three. |

**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 intends to expand its existing game, "Draw It or Lose It," from an Android-only application to a multi-platform, web-based distributed environment. The challenge is to design a software solution that allows multiple teams to participate in a game, with each team having several players. Furthermore, the solution needs to enforce unique names for games and teams and ensure only one instance of the game service is available at any given time. The proposed solution utilizes Object-Oriented Programming (OOP) principles and design patterns such as Singleton and Iterator to efficiently manage the game, team, and player entities in this distributed setting.

## Requirements

*The client, The Gaming Room, requires the following:*

* *The game application must support multiple teams, each with several players.*
* *Names for games and teams must be unique to prevent conflicts.*
* *Only one instance of the game service must be running to ensure consistency (Singleton Pattern).*
* *The application must be deployed in a web-based, distributed environment, allowing cross-platform accessibility for users.*
* *Security and Scalability: The system should be secure, scalable, and handle concurrent players across different platforms.*

*These requirements are focused on maintaining scalability, consistency, and ease of use for all users of the application.*

## [Design Constraints](#_2et92p0)

* **Web-Based Distributed Environment**: The application must be capable of running on a web-based distributed system, implying that components must be stateless where appropriate and capable of handling multiple requests concurrently. Ensuring synchronization and data consistency in a distributed setting adds complexity.
* **Singleton Implementation**: Since only one instance of GameService can exist in memory at any time, proper implementation of the Singleton pattern is necessary to manage game instances globally.
* **Unique Identifiers**: Unique names for games and teams require validation before adding them to their respective lists, which necessitates efficient searching mechanisms (e.g., the Iterator Pattern).
* **Cross-Platform Compatibility**: The design must ensure that the system is accessible across multiple platforms, including desktop operating systems (Linux, Windows, Mac) and mobile platforms (iOS, Android).

These constraints will impact the architecture of the application, requiring careful consideration of concurrency control, scalability, and system resource management.

## [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)

* **Entity Class**: A base class that contains shared properties like id and name. This follows the principle of **inheritance**, allowing all derived classes (Game, Team, Player) to inherit common fields, reducing code duplication.
* **GameService Class**: Implements the **Singleton Pattern** to manage games, ensuring only one instance exists. The service maintains a list of games (List<Game>) and provides methods for adding games, retrieving games by ID or name, and ensuring names are unique.
* **Game, Team, Player Classes**: These classes extend Entity to inherit common attributes. **Composition** is demonstrated in how a Game contains multiple Teams, and a Team contains multiple Players. This hierarchical structure allows easy navigation through related entities.

The use of inheritance, composition, and design patterns like Singleton and Iterator ensures that the system is maintainable, scalable, and efficient.

**"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 servers can host web applications effectively, but scalability and compatibility with enterprise software may be limited compared to Linux. | Linux is highly recommended for hosting web-based applications due to its stability, open-source nature, and scalability. It's widely used for server deployments. | Windows servers are reliable, but they require licensing, making them more costly. Windows also has good integration with Microsoft technologies. | Mobile devices are generally clients rather than servers. Hosting server-side applications on mobile devices is impractical due to limited processing power. |
| **Client Side** | Development for Mac clients requires expertise in macOS environments and specialized development tools like Xcode, which can increase costs. | Linux clients are less common but require minimal setup for web applications. Development is typically less expensive due to open-source tools. | Windows is widely used, making client-side development straightforward, though licensing costs must be considered. Expertise in .NET may be required. | Mobile client development requires expertise in native mobile platforms (Android, iOS) and cross-platform tools like React Native, which can impact cost and development time. |
| **Development Time** | Developing for macOS browsers will require additional time to test for Safari-specific behaviors, especially in handling media and performance optimizations for high-resolution displays. | Development for Linux systems generally aligns with standard web development timelines. Additional time may be needed for testing on multiple browsers like Firefox and Chrome. | Windows requires extensive testing across multiple browsers and screen sizes. Compatibility testing for different versions of Windows may extend the development timeline. | Testing for both Android and iOS devices will add to the overall time, particularly for optimizing touch interactivity, screen orientation changes, and performance across varying mobile hardware. |
| **Development Tools** | Tools like Xcode and Java can be used to build and deploy applications on macOS. Java-based web frameworks are also compatible. | Linux supports a variety of open-source tools like Eclipse, IntelliJ IDEA, and web servers like Apache and Nginx, making it versatile for development. | Tools like Visual Studio and .NET are used for development on Windows, providing a wide range of functionality for enterprise-level applications. | For mobile, tools such as Android Studio, Xcode, or cross-platform solutions like Flutter and React Native are commonly used. Java and Swift are relevant languages. |
| **Technical Impact on the Development Team** | Developing on macOS requires developers to be familiar with both Xcode for native apps and JavaScript frameworks for web-based apps. Multiple teams may be needed if native iOS apps are required alongside a web-based solution. | Linux developers typically have experience with open-source tools and are used to deploying scalable web applications. The team can be smaller because the Linux ecosystem supports both server and client development efficiently. | Windows developers might require expertise in **.**NET for enterprise applications, and JavaScript for web apps. Windows environments may require larger teams if the application uses complex server-side technologies. | Developing for both iOS and Android will require expertise in Swift (iOS) and Java/Kotlin (Android) or proficiency in cross-platform tools like **React Native**. Mobile development often necessitates a specialized team, but cross-platform tools can mitigate this. |
| **Licensing Costs** | Xcode is free, but it requires a Mac, which can increase hardware costs. There are no significant licensing costs for JavaScript-based frameworks. | Linux development tools are typically open-source and free, so there are no licensing costs. Enterprise Linux distributions like Red Hat may have support subscription costs. | VisualStudio has licensing fees for enterprise editions, but the CommunityEdition is free for smaller teams. Windows Server also has licensing costs, especially for large-scale deployments. | Android Studio is free, but iOS development requires a Mac and a $99/year developer fee for deploying apps to the App Store. Cross-platform tools like Flutter and React Native are free to use. |

## 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 deploying the game application on a Linux-based environment. Linux offers stability, scalability, and cost-effectiveness due to its open-source nature, making it ideal for hosting a web-based, distributed system. Additionally, Linux’s robust networking capabilities and strong community support make it an excellent choice for scalability and continuous development.
2. **Operating Systems Architectures**: The Linux architecture, with its modular and open-source kernel, is highly efficient for server-side applications. It facilitates smooth integration with various web servers and databases, ensuring the application can scale and maintain performance as user demand increases. Moreover, the architecture supports robust process management and multi-threading, which are essential for running multiple game instances simultaneously without causing delays or crashes.
3. **Storage Management**: For storage management, I recommend using a database management system like MySQL or PostgreSQL. These databases integrate well with Linux and provide reliable data management for storing game, team, and player data. Both options offer advanced features for data integrity, backup and recovery, and scalability, ensuring that as the game scales up, the database can handle more concurrent read and write operations. Considering cloud solutions like AWS RDS or Azure Database for PostgreSQL can also provide managed services, which reduce maintenance overhead.
4. **Memory Management**: Linux employs advanced memory management techniques such as virtual memory, caching, and paging to efficiently utilize system resources. These features are beneficial when handling multiple concurrent users in a distributed environment. Linux’s ability to optimize memory allocation ensures that each game instance receives the required resources without monopolizing the server’s capacity. Utilizing caching mechanisms can also improve performance by reducing the need to fetch images repeatedly from the disk.
5. **Distributed Systems and Networks**: Communication between different platforms can be achieved using a RESTful API, which allows various clients (e.g., web browsers, mobile apps) to interact with the server over HTTP/HTTPS protocols. This approach ensures compatibility and efficient data exchange across different platforms. To handle network issues, implementing load balancing and redundant server configurations can improve reliability, ensuring that game instances remain active even if a server experiences connectivity issues.
6. **Security**: Security is critical for protecting user information and ensuring the integrity of game data. Security measures include SSL encryption for data transmitted between platforms, proper user authentication, and robust authorization mechanisms. On the Linux server, tools like iptables can be used for firewall management, while fail2ban can help prevent unauthorized access. To ensure user data security, all sensitive information, such as passwords and personal data, should be encrypted both in transit and at rest. Regular security audits and updates will also help maintain the system’s integrity over time.