

Game Management System

**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 6**](#_Toc115077326)

**[Document Revision History](#_grjogdjh5fi8)**

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 01/26/25 | Jason Hney | Initial version of the software design document |

**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 tasked us with designing a scalable and efficient solution for "Draw It or Lose It." This project involves transitioning the game from a single-platform environment to a web-based distributed environment to allow multi-platform access. Our solution leverages object-oriented principles, the Singleton design pattern for GameService, and the UML class diagram structure to ensure maintainability, scalability, and optimal resource management. This document outlines the technical requirements, design constraints, and recommendations to achieve this transition seamlessly.

**Requirements**

The client’s business and technical requirements include:

1. Transitioning "Draw It or Lose It" to a web-based distributed environment to support multiple platforms.
2. Ensuring scalability and extensibility of the application to handle growth in users, teams, and games.
3. Incorporating security measures to protect user data and maintain data integrity.
4. Employing object-oriented design principles for code reuse and maintainability.
5. Using modern software development tools and platforms that align with the client’s infrastructure and goals.

[**Design Constraints**](#_2et92p0)

Design constraints for this project include:

1. **Web-Based Distributed Environment**: The application must function across different operating systems and devices, requiring robust client-server communication and consistent user experience across platforms.
2. **Singleton Design Pattern**: The GameService class utilizes the Singleton pattern, ensuring only one instance exists for managing games, teams, and players, which simplifies resource management but necessitates careful implementation to avoid synchronization issues.
3. **Security**: With sensitive user data being transmitted across networks, robust encryption protocols and secure storage mechanisms must be implemented.
4. **Platform Compatibility**: The application must be compatible with Windows, Linux, macOS, and mobile devices, each of which has unique development and deployment challenges.

[**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 provided illustrates the relationships between various classes:

1. **Entity Class**: Serves as a base class for Game, Team, and Player, encapsulating common properties such as ID and name. This adheres to the object-oriented principle of inheritance, promoting code reuse.
2. **GameService Class**: Implements the Singleton design pattern to manage the creation and retrieval of games, teams, and players. It also ensures unique IDs are generated for each entity.
3. **Game Class**: Represents a game and maintains a list of associated teams, demonstrating composition.
4. **Team Class**: Represents a team and maintains a list of players, further extending the composition principle.
5. **Player Class**: Represents individual players within a team.

These classes collaboratively fulfill the requirements of scalability, modularity, and reusability.

**"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** | macOS supports server-based deployment through macOS Server, but it is not widely used for large-scale web applications. It may not be the best choice for hosting due to limited support for enterprise web services. Licensing costs can be higher, and server maintenance tools are less developed compared to Linux. | Linux is the most common choice for web server deployment due to its open-source nature, high security, and extensive support for web hosting. It supports Apache, Nginx, and cloud-based hosting solutions. No licensing costs make it cost-effective. | Windows Server is a strong option for web hosting, especially with ASP.NET applications. However, it requires licensing fees and is typically more resource-intensive. It supports IIS (Internet Information Services) but may have security vulnerabilities. | Mobile devices are not designed to host server applications. However, they can interact with cloud-based servers and APIs to deliver content efficiently. |
| **Client Side** | macOS supports modern web browsers like Safari, Chrome, and Firefox. Web applications must be optimized for macOS screen resolutions and system preferences. Development for macOS requires expertise in Swift, Objective-C, or cross-platform frameworks like React or Flutter. | Linux users primarily access web applications via browsers like Firefox and Chrome. Linux desktop adoption is lower, but web applications remain accessible. Web-based development must ensure compatibility across distributions. | Windows is widely used, and applications should be tested on Edge, Chrome, and Firefox for browser compatibility. Windows supports web technologies, and the majority of users are familiar with the platform. | Mobile devices require responsive web design and compatibility with Android and iOS browsers. The application should be touch-optimized and lightweight for smooth performance. |
| **Development Tools** | Xcode is the primary development tool for macOS applications. Web applications can be built using standard front-end technologies like HTML, CSS, and JavaScript, supported by tools such as VS Code and WebStorm. | Linux offers powerful open-source development tools like Eclipse, IntelliJ IDEA, and VS Code. Web applications can be developed using PHP, Python, Node.js, and other technologies. | Windows supports development environments like Visual Studio, IntelliJ IDEA, and VS Code. ASP.NET, Node.js, and JavaScript frameworks are common for web development. | Mobile applications require tools like Android Studio (for Android), Xcode (for iOS), or cross-platform solutions like React Native, Flutter, or Xamarin. |

**Recommendations**

Analyze the characteristics of and techniques specific to various systems architectures and make a recommendation to The Gaming Room. Specifically, address the following:

**Operating Platform**:

For *Draw It or Lose It*, a **Linux-based server environment** is the most suitable operating platform. Linux offers **high scalability, stability, security, and cost-effectiveness**, making it an ideal choice for web-based distributed applications. With broad support for web servers such as **Apache and Nginx**, Linux ensures smooth deployment and seamless integration with cloud services. Additionally, Linux's open-source nature eliminates licensing costs, reducing overall expenses.

**Operating Systems Architectures**:

Linux follows a **monolithic kernel architecture**, meaning essential services like memory management, process scheduling, and file system handling are integrated into the kernel. This architecture ensures **high performance, efficient resource allocation, and better system calls handling**. In contrast, Windows follows a **hybrid kernel** approach, blending monolithic and microkernel benefits. For *Draw It or Lose It*, the Linux monolithic architecture enhances **reliability, system efficiency, and rapid interprocess communication (IPC)**, which is essential for a multi-platform game.

**Storage Management**:

The recommended storage management system is a **cloud-based distributed storage solution**, such as **Amazon S3, Google Cloud Storage, or Azure Blob Storage**. These solutions provide:

* **Scalability:** Automatically adjusts storage based on user demand.
* **Data Redundancy:** Prevents data loss with built-in backups.
* **Low Latency:** Ensures fast retrieval times, crucial for a real-time gaming experience.  
  Additionally, **MySQL or PostgreSQL** can be used for structured game data storage, while **Redis or Memcached** can handle session storage and caching.

**Memory Management**:

Linux utilizes **paging and virtual memory** techniques to optimize memory usage. The recommended server will allocate memory dynamically using techniques like **demand paging, swap space, and kernel-level memory management**. This ensures:

* Efficient handling of multiple users.
* Optimized RAM utilization with swap space.
* Reduced memory leaks using **garbage collection** in managed languages like Python or Java.

For the **client-side**, memory usage is optimized using **browser caching**, **lazy loading**, and **local storage** to reduce server load and improve responsiveness.

**Distributed Systems and Networks**:

To allow *Draw It or Lose It* to operate across multiple platforms, a **distributed system architecture** is required. This includes:

* **Client-server model:** The game logic resides on the server, while clients (Windows, Mac, Linux, mobile devices) interact via API calls.
* **Load balancing:** Distributes incoming traffic across multiple servers using tools like **NGINX, AWS Elastic Load Balancer, or HAProxy**.
* **Microservices architecture:** Breaks down the game logic into independent services, improving scalability and fault tolerance.
* **Cloud deployment:** Hosting the application on **AWS, Google Cloud, or Azure** ensures global accessibility.
* **WebSockets or RESTful APIs:** Enables real-time data synchronization between devices.

**Security**:

Security is critical for user data protection. The following strategies will be implemented:

* **End-to-end encryption (TLS 1.3):** Protects data during transmission.
* **Role-based access control (RBAC):** Restricts data access based on user roles.
* **OAuth 2.0 authentication:** Secures user login and third-party integrations.
* **Regular security updates and patching:** Prevents vulnerabilities.
* **Data hashing (SHA-256):** Ensures safe password storage.
* **Distributed Denial of Service (DDoS) protection:** Using **Cloudflare or AWS Shield** to mitigate attacks.