

<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 | 07/19/2025 | Sheila Bonilla | First submission |
| 2.0 | 07/31/2025 | Sheila Bonilla | Added Evaluation section (Project Two) |
| 3.0 | 08/14/2025 | Sheila Bonilla | Completed Recommendations section and final review |

**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 planning to expand its current Android-based game, *Draw It or Lose It*, into a web-based application that can run on multiple platforms. The primary objective is to create a scalable and maintainable game environment that supports various teams and players while ensuring optimal performance and stability. This plan utilizes object-oriented programming and design patterns, such as the Singleton and Iterator patterns, to maintain code organization and ensure reusability. It also makes sure that game, team, and player names are unique to avoid duplicates. The system is built in parts, allowing it to handle many users simultaneously without slowing down.

The game consists of four rounds, each lasting one minute. During each round, drawings are gradually rendered from an extensive library of stock images, with the whole image completed at the 30-second mark. Players must guess the puzzle, whether it is a phrase, title, or object, before time expires. If the main team does not guess the answer within the time limit, the remaining teams get one chance each to solve the puzzle within a 15-second timeframe. This structure ensures engaging, fast-paced gameplay while giving all teams the opportunity to score. This setup helps the client grow the game and keep players interested.

## Requirements

*<* Please note: While this section is not being assessed, it will support your outline of the design constraints below. *In your summary, identify each of the client’s business and technical requirements in a clear and concise manner.>*

## [Design Constraints](#_2et92p0)

* Web-Based Distributed Environment: The application must function across various browsers and devices without requiring installation. This requires careful attention to responsiveness, compatibility, and security in a networked environment.
* Single Game Instance in Memory: The application architecture must ensure that only one instance of the GameService class exists in memory at any given time. This constraint is fulfilled using the singleton design pattern, which helps avoid concurrency issues and maintain a consistent game state.
* Unique Names: The system should enforce the uniqueness of game, team, and player names to

prevent naming conflicts and provide an optimal user experience when creating and joining

games.

* Timeline Constraints: The web-based version must be ready within a fixed timeline. This limits flexibility in experimentation and enforces agile, iterative development cycles.

## [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 GameService class oversees the setup and management of all games within the application. It uses the Singleton design pattern to ensure that only one instance is active in memory at any time. This approach helps prevent inconsistencies and redundant access to the service. To enforce this, the constructor is declared private, and a public getInstance() method is used to access the single instance when necessary. When creating a new game via the addGame() method, the application checks the current list of games to confirm that the proposed name hasn't already been used. To avoid duplicate entries, the program goes through each item in the list one by one, checking whether the name already exists. This same approach is used when adding a new team or player; if the name hasn't been taken, it gets added to the list. The way the application is built follows key object-oriented programming practices, keeping everything organized and consistent.

• Inheritance: Inheritance: The Game, Team, and Player classes all inherit from a common superclass named Entity, which contains a unique identifier and a name. This structure reduces redundant code and promotes a uniform design across all classes.

• Encapsulation: Class attributes are kept private or protected. Data can only be accessed or modified using specific methods, which maintain control and protect internal values.

• Polymorphism: Constructors and functions are implemented to adapt their behavior based on the context in which they’re used. This flexibility allows the program to address various situations while using shared logic.

• Abstraction: Methods such as addTeam() and addPlayer() enable users to work with high-level actions without needing to understand the underlying implementation details.

This model ensures that the system remains organized, scalable, and easy to maintain. It also meets the client’s key requirements: guaranteeing that names are unique, providing only one instance of GameService runs in memory, and supporting the application in a distributed, 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** | macOS can host a web server using Apache or Nginx, but it is rarely used in large-scale deployments due to limited enterprise support and dependency on Apple hardware. Licensing costs are included with Apple hardware, but macOS cannot be freely installed on non-Apple servers. | Linux is the preferred choice for web servers. It is free (open-source), reliable, resource-efficient, and supports Apache, Nginx, cloud hosting, and containerization. It scales well for thousands of concurrent players. | Windows Server works well with IIS and .NET applications. It has strong enterprise support and is user-friendly but requires paid licensing and consumes more resources than Linux for large-scale hosting. | Mobile devices cannot function as servers due to limited memory, CPU power, and bandwidth. They are strictly clients for accessing the hosted application. |
| **Client Side** | macOS supports the game via web browsers like Safari or Chrome. Native apps require Xcode and Apple hardware. Development cost is higher due to hardware and OS requirements. | Linux desktops access the game through modern browsers (Chrome, Firefox, Edge). Supporting multiple distributions can require additional technical expertise, but it has no licensing costs. | Windows has the largest desktop market share, allowing the game to reach many users via Edge, Chrome, or Firefox. Development is faster with abundant documentation and talent. | Mobile clients include iOS and Android. The web app must be responsive and tested on multiple screen sizes. Native apps or wrappers require Android Studio and Xcode, and testing is time-intensive due to device diversity. |
| **Development Tools** | macOS supports Xcode, IntelliJ, VS Code, and Terminal. Developers can use Swift for native apps or HTML, CSS, and JavaScript for web-based UI. | Linux supports Eclipse, NetBeans, VS Code, and CLI tools. Common languages include Java, Python, PHP, Ruby, and JavaScript. Highly customizable but requires advanced technical skills. | Windows developers commonly use Visual Studio, Eclipse, and PyCharm. Supported languages include C#, C++, Java, Python, and JavaScript. Licensing is required for some IDEs, such as Visual Studio Professional. | Mobile app development uses Android Studio (Kotlin/Java) and Xcode (Swift/Objective-C). Cross-platform frameworks like Flutter or React Native can reduce effort. Browser testing ensures the responsive site works seamlessly. |

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**: For hosting *Draw It or Lose It*, Ubuntu Server (Linux) is the most suitable choice. It delivers reliable performance, strong built-in security, and the flexibility to scale as demand increases without the cost of commercial licenses. It integrates seamlessly with leading web servers such as Apache and Nginx and includes native support for containerized workloads and cloud deployments. This setup makes it capable of supporting thousands of concurrent players without compromising stability or speed.

It is important to note that this recommendation applies to the server side. On the client side, the game will run in a browser on Windows, macOS, Linux, and mobile devices, ensuring maximum reach and compatibility (Red Hat, 2023).

1. **Operating Systems Architectures**: A modular, distributed architecture is the best approach for this application. Separating the web server, application logic, and database layers improves fault tolerance and simplifies updates. Linux is well-suited to this design because it can handle independent processes without disrupting other services, which is critical for maintaining uninterrupted gameplay.

Its kernel supports multiple inter-process communication methods (such as sockets, pipes, and shared memory) that allow the drawing engine, scoring system, and chat module to work in parallel. At the infrastructure level, CDN, load balancer, stateless container instances, relational database, and object storage pipeline provide flexibility and support rolling updates without downtime (Silberschatz et al., 2022).

1. **Storage Management:** To keep gameplay smooth, the system must retrieve images and game data almost instantly. This is achieved by combining a Content Delivery Network (CDN) with efficient caching strategies so that frequently used files reach players with minimal latency.

On the backend, using NVMe or SSD storage ensures rapid read performance, while Linux Logical Volume Manager (LVM) allows storage to be expanded without downtime. Player data and scores will be stored in a relational database like PostgreSQL, while game assets, roughly 1.6 GB for 200 HD images, will be stored in an object storage service.

Access to these assets is controlled through the backend or CDN, not directly from the client, ensuring better security. Regular nightly backups and replication safeguard against data loss, and snapshot capabilities provide quick recovery if updates fail or data is corrupted (IBM, 2024).

1. **Memory Management**: Linux manages memory efficiently, even under heavy workloads. It uses virtual memory, swap space, and demand paging to ensure active processes always have the resources they need. The page cache speeds up repeated access to frequently used files, and cgroups limit memory usage per service, so one process cannot consume all resources.

In gameplay, the current and next round’s assets are preloaded into memory, while unused data is released after each round. This approach ensures smooth performance even with thousands of active players (Silberschatz et al., 2022).

1. **Distributed Systems and Networks**: The game will operate as a distributed, web-based system, with the server processing API requests from both desktop and mobile clients. Standard operations will use HTTPS/REST, while real-time updates such as countdown timers or score changes will be pushed over WebSockets.

Cloud hosting combined with load balancing ensures high availability. Load balancers will perform health checks and reroute traffic automatically if a node fails. On the client side, retry logic with exponential backoff will prevent unnecessary strain on the server, and idempotent operations will prevent duplicate actions during temporary connection issues (Tanenbaum & Van Steen, 2017).

1. **Security**: All communication will be encrypted using HTTPS/TLS to prevent unauthorized interception. Authentication will follow OAuth 2.0 and short-lived JWTs, while Role-Based Access Control (RBAC) will manage permissions between regular players and administrators.  
   Backend security will include strict input validation, parameterized queries to prevent SQL injection, and the use of Content Security Policy (CSP) and Cross-Origin Resource Sharing (CORS) rules. If cookies are used, Cross-Site Request Forgery (CSRF) protection will be in place.  
   Infrastructure protections will include a Web Application Firewall (WAF) to block malicious traffic, rate limiting to control repeated requests, regular key rotation, and centralized logging with real-time alerts for incident response (OWASP, 2023).

**References**

IBM. (2024). *Linux storage management with LVM*. IBM Documentation. <https://www.ibm.com/docs/en/linux>

OWASP. (2023). *Transport Layer Protection Cheat Sheet*. OWASP Foundation. <https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protection_Cheat_Sheet.html>

Red Hat. (2023). *Why choose Linux for servers*. Red Hat. <https://www.redhat.com/en/topics/linux>

Silberschatz, A., Galvin, P. B., & Gagne, G. (2022). *Operating system concepts* (10th ed.). Wiley. <https://www.os-book.com>

Tanenbaum, A. S., & Van Steen, M. (2017). *Distributed systems: Principles and paradigms* (3rd ed.). Pearson. <https://www.distributed-systems.net>