

# **Draw it or Lose it Web App** **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 | 07/22/2025 | Brandon Patrick | Initial Draft |
| 1.1 | 09/20/2925 | Brandon Patrick | Updated Recommendations section 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)

Creative Technology Solutions is working with The Gaming Room to modernize their game, *Draw It or Lose It*, by developing a web-based version that can support multiple platforms. Currently, the game is only available on Android, and the goal is to expand to other systems using a scalable and maintainable software design. The solution will use object-oriented principles and software design patterns to meet requirements such as unique game/team names, multiple teams and players, and a single instance of the game service. This design document outlines the software architecture, domain model, evaluation of platforms, and platform recommendations that will guide development.

## Requirements

The Gaming Room requires:

* The ability to support one or more teams per game.
* Each team must support multiple players.
* Game and team names must be unique.
* Only one instance of the game should exist in memory.
* The application must be accessible on multiple platforms and allow team creation, player tracking, and game state management.

## [Design Constraints](#_2et92p0)

The primary constraint is that the application must run in a web-based distributed environment, which introduces challenges such as concurrency, session state management, and synchronization of game data across platforms. To address this, the design will use the Singleton pattern to ensure only one instance of the GameService exists, and the Iterator pattern to efficiently search for existing names. Unique identifiers will be generated for games, teams, and players to maintain consistency. Additionally, design decisions must support scalability, platform-independence, and efficient performance over the web.

## [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 shows a clear hierarchy using object-oriented design principles. The Entity class acts as a base class containing shared fields id and name, which are inherited by the Game, Team, and Player classes. The GameService class handles all game logic and data management, using lists to store games and track the next available IDs. This design promotes **inheritance**, **encapsulation**, and **code reuse**, reducing duplication and improving maintainability.

Each class manages its own collection: Game holds a list of Team objects, and each Team holds a list of Player objects. The use of the Singleton pattern in GameService ensures that only one game service instance is created, meeting the client’s constraint. The Iterator pattern is implied in methods like addGame() and getGame() to loop through lists and check for uniqueness. Overall, the design is modular and scalable.

**"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** | Can host via Apache/Nginx but uncommon in production. Better for development environments. | Most popular choice for web hosting. Stable, secure, low cost, and highly supported. | Supports web hosting but has higher licensing costs and more overhead. Reliable, but not as cost-effective. | Not used for servers. Primarily client-side. |
| **Client Side** | Requires Safari compatibility and Apple hardware for testing. | Less common as desktop OS, but still requires testing for browser compatibility. | Most common desktop OS. Testing and support are essential. | Must support both Android and iOS. Requires responsive design and device-specific testing. |
| **Development Tools** | Xcode, IntelliJ IDEA, Java, HTML/CSS/JS. | Eclipse, IntelliJ, Java, VS Code, Git. Strong open-source ecosystem. | Visual Studio, IntelliJ, Eclipse, Java, web frameworks. | Android Studio for Android, Xcode for iOS, Flutter or React Native for cross-platform. |

## Recommendations

**Operating Platform**  
 Linux is the recommended platform for hosting the game backend due to its cost-effectiveness, modular architecture, and strong support in web-based distributed environments. It is widely used for scalable web applications and integrates seamlessly with cloud platforms.

**Operating Systems Architectures**  
 Linux uses a **monolithic kernel architecture** with modular support. The kernel handles memory management, device drivers, and process scheduling in one core, allowing efficient performance. Its modularity ensures only required services are loaded, improving security and stability. This design is well-suited to hosting a distributed, web-based application like *Draw It or Lose It*.

**Storage Management**  
 The application requires storage for roughly 200 images at 8 MB each (~1.6 GB), plus additional space for metadata, player data, and logs. A **cloud object storage system** such as Amazon S3 is recommended, offering high availability and redundancy. On the server, journaling file systems such as **ext4** or **XFS** ensure durability and integrity. Player data should be handled by a managed database (MySQL/PostgreSQL) or cloud-native solution like AWS RDS for scalability.

**Memory Management**  
 Linux employs **virtual memory, paging, caching, and demand loading** to maximize efficiency. Frequently accessed image assets can be cached in RAM for smoother gameplay. Java’s **garbage collection** will handle clearing unused objects, while Linux process isolation prevents memory leaks from affecting other processes. This combination ensures responsiveness while maintaining stability under load.

**Distributed Systems and Networks**  
 The game should be delivered as a **distributed system** using RESTful APIs or WebSockets for communication between clients and the server. Load balancers will distribute requests across server instances. Redundancy and failover mechanisms should be implemented to handle outages. Containerization with **Docker** and orchestration with **Kubernetes** will allow scaling and resilience across nodes in the distributed environment.

**Security**  
 User security will be prioritized through **TLS/HTTPS encryption**, **input validation**, and **role-based access control (RBAC)**. Passwords will be hashed, and sensitive data encrypted. Firewalls and intrusion detection systems will help defend against external threats. For enhanced user authentication, multi-factor authentication (MFA) or OAuth2 can be integrated. Logs will be maintained for monitoring and audits to detect suspicious behavior.

### **References**

Amazon Web Services. (n.d.). *Amazon S3*. <https://aws.amazon.com/s3/>

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating system concepts* (10th ed.). Wiley.

Tanenbaum, A. S., & van Steen, M. (2016). *Distributed systems: Principles and paradigms* (3rd ed.). Pearson.