

# **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 | 07/08/2025 | Zoe Render | Code and written changes listed below. (Zip file separate) |

**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)

1.0  
06/08/25  
Zoe  
Initial draft filled with project-specific content.

## Requirements

The Gaming Room seeks to expand their Android-only game “Draw It or Lose It” to a web-based, multi-platform solution. This project presents a software design that leverages object-oriented principles and industry-standard design patterns to meet their core requirements. A Singleton pattern ensures only one instance of the game exists at a time, while the Iterator pattern supports unique naming and effective team/player management. By designing for scalability and reusability, this solution lays a strong foundation for future enhancements and deployment across diverse platforms.

## [Design Constraints](#_2et92p0)

- The game must support multiple teams and players.  
- Each game, team, and player must have a unique identifier and name.  
- Only one instance of the game should exist at a time.  
- Teams should be able to check for name availability.  
- The solution must be scalable to support future multi-platform deployment.

## [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 includes four main classes: Entity, Game, Team, and Player, as well as the service manager GameService.  
- Entity serves as the base class, holding shared attributes id and name.  
- Game, Team, and Player extend Entity, showing inheritance.  
- GameService controls the creation and storage of instances using static collections and methods, and follows the Singleton pattern to ensure only one instance exists.  
- The use of Encapsulation is clear, with private fields accessed via getters/setters.  
- The Iterator pattern is employed to traverse collections to prevent name duplication and to locate specific entities efficiently.  
- These principles and patterns provide a clear, scalable design supporting code reusability, maintainability, and constraint enforcement.

**"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** | Evaluate Mac for its characteristics, advantages, and weaknesses for hosting a web-based software application. | Evaluate Linux for its characteristics, advantages, and weaknesses for hosting a web-based software application. | Evaluate Windows for its characteristics, advantages, and weaknesses for hosting a web-based software application. | Evaluate Mobile Devices for their characteristics, advantages, and weaknesses for hosting a web-based software application. |
| **Client Side** | Determine the software development considerations (cost, time, expertise) that are necessary for supporting multiple types of clients as they pertain to Mac. | Determine the software development considerations (cost, time, expertise) that are necessary for supporting multiple types of clients as they pertain to Linux. | Determine the software development considerations (cost, time, expertise) that are necessary for supporting multiple types of clients as they pertain to Windows. | Determine the software development considerations (cost, time, expertise) that are necessary for supporting multiple types of clients as they pertain to Mobile Devices. |
| **Development Tools** | Identify the relevant programming languages and tools (IDEs and other tools) that are used to build this type of software for deploying on Mac. | Identify the relevant programming languages and tools (IDEs and other tools) that are used to build this type of software for deploying on Linux. | Identify the relevant programming languages and tools (IDEs and other tools) that are used to build this type of software for deploying on Windows. | Identify the relevant programming languages and tools (IDEs and other tools) that are used to build this type of software for deploying on Mobile Devices. |

## Recommendations

Operating Platform: I recommend using a Linux-based server environment for hosting Draw It or Lose It. Linux offers stability, scalability, security, and cost-efficiency—key advantages for a web-based, multi-platform application. It is widely supported by major cloud providers, making it suitable for future scalability needs.  
  
Operating Systems Architectures: The recommended Linux platform supports a multi-tiered architecture, enabling separation of the presentation layer (front end), application logic (game engine), and data storage (back end). This architecture facilitates modular development, streamlined debugging, and future integration with mobile clients and APIs. The architecture can be deployed using containers (e.g., Docker) and orchestrated using Kubernetes to enhance portability and scalability across environments.  
  
Storage Management: For efficient and scalable storage, a cloud-based relational database such as MySQL or PostgreSQL is ideal. These systems allow for strong data consistency, query flexibility, and ease of management. For high availability, the database can be configured with replication and backup strategies. Additionally, using cloud-native storage like AWS RDS or Google Cloud SQL enhances performance and provides automatic failover and security updates.  
  
Memory Management: Linux uses virtual memory and supports memory paging and caching efficiently, allowing the game server to allocate memory dynamically based on player load. The GameService Singleton design pattern minimizes unnecessary memory duplication by enforcing a single instance for game control. The OS can also employ memory swapping when system RAM is insufficient, ensuring smooth operation during peak usage.  
  
Distributed Systems and Networks: To enable cross-platform communication, the application will use RESTful APIs over HTTPS. These lightweight, stateless protocols allow multiple front ends (web, mobile) to interact with the server consistently. Load balancers and horizontal scaling will help mitigate network outages and high traffic loads. Using JSON for data exchange ensures compatibility across devices. Dependency monitoring tools (e.g., Prometheus and Grafana) will assist in identifying bottlenecks and failures in real-time.  
  
Security: To protect user data and game integrity:  
- Implement HTTPS with TLS encryption for all data in transit.  
- Use secure authentication and role-based authorization.  
- Sanitize user inputs to prevent injection attacks.  
- Store passwords using strong hashing algorithms (e.g., bcrypt).  
- Enable platform-specific security features (e.g., firewall rules, app sandboxing on mobile).  
These practices help ensure secure communication and data protection across all supported platforms.

1. **Operating Platform**: I recommend using a Linux-based server environment for hosting Draw It or Lose It. Linux offers stability, scalability, security, and cost-efficiency—key advantages for a web-based, multi-platform application. It is widely supported by major cloud providers, making it suitable for future scalability needs.
2. **Operating Systems Architectures**: The recommended Linux platform supports a multi-tiered architecture, enabling separation of the presentation layer (front end), application logic (game engine), and data storage (back end). This architecture facilitates modular development, streamlined debugging, and future integration with mobile clients and APIs.
3. **Storage Management**: For efficient and scalable storage, a cloud-based relational database such as MySQL or PostgreSQL is ideal. These systems allow for strong data consistency, query flexibility, and ease of management. For high availability, the database can be configured with replication and backup strategies.
4. **Memory Management**: Linux uses virtual memory and supports memory paging and caching efficiently, allowing the game server to allocate memory dynamically based on player load. Singleton design ensures minimal memory usage by allowing only one instance of the GameService class in memory.
5. **Distributed Systems and Networks**: To enable cross-platform communication, the application will use RESTful APIs over HTTPS. These lightweight, stateless protocols allow multiple front ends (web, mobile) to interact with the server consistently. Load balancers and horizontal scaling will help mitigate network outages and high traffic loads. Using JSON for data exchange ensures compatibility across devices.
6. **Security**: To protect user data and game integrity:

* Implement HTTPS with TLS encryption for all data in transit.
* Use secure authentication and role-based authorization.
* Sanitize user inputs to prevent injection attacks.
* Store passwords using strong hashing algorithms (e.g., bcrypt).
* Enable platform-specific security features (e.g., firewall rules, app sandboxing on mobile).  
  These practices help ensure secure communication and data protection across all supported platforms.