

Draw It or Lose It

# **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 | 08/03/2025 | John Rosario | Updated Evaluations |

**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 expanding its game *Draw It or Lose It* from a single-platform Android app to a cross-platform web application. This document outlines a software design plan that supports multiple teams, ensures unique naming, and enforces a single instance of each game in memory. The proposed design applies object-oriented programming and design patterns such as Singleton and Iterator to ensure scalability, maintainability, and clarity in the application’s architecture. The solution prioritizes streamlined development while meeting the client’s software requirements.

## Requirements

The following requirements must be met:

* A game must support one or more teams.
* Each team must have multiple players.
* Game and team names must be unique to prevent naming conflicts.
* Only one instance of a game can exist in memory at any given time.
* The system must validate names before adding new entities.

## [Design Constraints](#_2et92p0)

* The system must function in a web-based distributed environment, which introduces challenges such as concurrent data access, session management, and scalability.
* A single instance of each game must exist at runtime, requiring the use of the **Singlton pattern** for the GameService class.
* Management of entities like teams and players should use the **Iterator pattern** to allow efficient traversal.
* All names must be unique, necessitating validation checks during creation. These constraints shape the application’s architecture and guide how objects are created and managed in memory.

## [System Architecture View](#_ilbxbyevv6b6)

The system follows a three-tier architecture:

1. **Presentation Layer:** the future web UI that will allow users to create games, teams, and players.
2. **Application Layer**: Java classes including Game, Team Player, and GameService, handling business logic.
3. **Data Layer** (not implemented): Will eventually handle persistence.

## [Domain Model](#_8h2ehzxfam4o)

The Domain model consists of several interconnected classes that use object-oriented principles to satisfy the software requirements of the Gaming Room application.

At the core is the Entity class, which provides shared fields (id and name) and methods (getIdI(), getName(), and toString()). TheGame, Team and Player classes all extend Entity, inheriting these common features to promote code reuse and maintain consistency.

The GameService class is designed as a Singleton. This ensures that only one instance of the service exists in memory at any given time, meeting the client’s requirements for a single running instance. The GameService class manages a list of Game objects and handles ID generation for games, players, and teams using internal counters (nextGameId, nextTeamId, and nextPlayerId).

The Game class contains a list of Team objects, and each Team contains a list of Player objects. These associations fulfill the requirement of supporting multiple teams per game and multiple players per team. The addition of teams and players is handled through addTeam() and addPlayer() methods and traversal of collections is done using iterators and support modular design and future scalability.

This model demonstrates the principles of **inheritance, encapsulation,** and  **abstraction**, and uses **design patterns** like **Singleton** and **Iterator** to enforce software constraints. The design promotes clean architecture and ensures that the game system is scalable and adaptable for future enhancements.

**"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 has a Unix-based architecture and supports many development tools. It is less common in production environments compared to Linux and may require additional licensing or hardware investments. macOS servers are rare in production; Apple-only licen | Linux is the most widely used OS for server-side web application hosting due to its stability, scalability, open-source nature. It supports secure and efficient deployment with tools like Apache, Nginx, and Docker. | Windows servers offer good integration with Microsoft technologies like .NET and SQL Server. However, they are typically more expensive to license and may have less community support for open-source stacks. | Mobile devises are not designed to host server-side web applications, but rather to consume them. Hosting on a mobile device is impractical due to hardware, battery, and network limitations. However, mobile compatability must be considered when designing the front-end of the application. |
| **Client Side** | macOS offers excellent development environments like Xcode and IntelliJ, and strong browser compatibility for testing web clients. However, developers must ensure the web app behaves consistently across macOS and other operating systems, especially Windows. | Linux is widely used for development and testing, with IDEs like Eclipse and tools like VS Code. It is less common among everyday end users, so testing should still be done on more mainstream OSs. It offers flexibility and open-source advantages for developers. | Windows is the most commonly used OS by end users, so it is critical to ensure compatibility. It supports a wide range of browsers and development tools. Developers must ensure the application functions correctly in Edge, Chrome, and Firefox on Windows. | Web-applications must be responsive and mobile-friendly. Development must consider screen sizes, touch input, and performance constraints. Testing should be performed on both iOS and Android devices using emulators or real devices. |
| **Development Tools** | macOS supports tools like IntelliJ IDEA, Eclipse, Xcode and VS Code. Java is fully supported making it suitable for this project. Terminal and Homebrew also aid in managing development environments. | Linux supports a wide range of development tools like Eclipse, IntelliJ, and VS Code. Java and build tools like Maven or Gradle are easy to set up. Linux is favored for server development and scripting. | Tools like Eclipse, IntelliJ IDEA and VS Code are fully supported on Windows. It is beginner-friendly, and many developers use Windows for initial development. Java and the JDK are easy to install and maintain. | Mobile Devices are not used to build software but rather test it. Development tools like Android Studio or Xcode (for iOS) are installed on desktops but allow developers to simulate or deploy apps on mobile. |

## 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****,*** a **Linux-based operating platform** is recommended. Linux is widely supported in cloud environments, offers strong performance and security, and is open-source, reducing hosting costs. Its compatibility with Java-based applications makes it ideal for deploying the game server across platforms such as AWS, Azure, or Google Cloud.
2. **Operating Systems Architectures**: The Linux operating system uses a **modular and layered architecture**, which allows developers to customize components for performance and security. It follows a monolithic kernel structure, giving direct access to hardware-level operations and efficient resource handling. This makes it suitable for scalable, high-performance distributed applications like *Draw It or Lose It*.
3. **Storage Management**: Linux offers **robust storage management tools** such as Logical Volume Manager (LVM), ext4 file systems, and cloud-compatible storage integrations. These tools support scalable, secure data handling for session data, game states, and user profiles. Persistent storage (e.g., using PostgreSQL or MongoDB) should be implemented to ensure player and game data is saved between sessions.
4. **Memory Management**: Linux uses **virtual memory management**, which provides efficient use of RAM through paging and swapping. This helps the application scale by allocating memory dynamically based on usage. For *Draw It or Lose It*, this ensures multiple games, teams, and players can be active in memory while unused data is swapped to disk when necessary, keeping performance optimal.
5. **Distributed Systems and Networks**: To ensure seamless multi-platform support, *Draw It or Lose It* should be deployed using **distributed architecture** principles – such as running game logic on a central server while clients (web/mobile) interact through APIs. Technologies like RESTful services or WebSockets can facilitate real-time communication. Load balancers and failover strategies can ensure resilience against outages, while each client platform (mobile, browser) connects independently via secure, HTTPS protocols.
6. **Security**: Security is essential for user data protection. Linux supports **built-in firewalls (iptables/ufw)**, file permission systems, and encryption libraries. User Authentication should be enforced with hashed passwords (e.g., using bcrypt) and optional multi-factor authentication. All client-server communication should be encrytped with TLS/SSl. Regular patching and server hardening will help protect against threats.