

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 | 09/21/2025 | Valari Alley | Project One part |
| 1.1 | 10/03/2025 | Valari Alley | Project Two part |
| 1.2 | 10/18/2025 | Valari Alley | Project Three part |

**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 aims to expand its current Android-only application, "Draw It or Lose It," into a web-based game that is accessible across multiple platforms. The primary challenge is designing a scalable and maintainable software architecture that supports this expansion while meeting key business requirements. This document outlines a comprehensive software design solution that addresses these needs.

The proposed solution is built on a robust, object-oriented design that ensures the application is both flexible and efficient. To meet the specific requirement that only one instance of the game can exist in memory, the design incorporates the Singleton pattern. This ensures a single, globally accessible game service manages all game sessions, preventing data conflicts and simplifying state management. Furthermore, the design uses inheritance to establish a clear and reusable structure for all primary game components (games, teams, and players), reducing code redundancy. By following the plan laid out in this document, The Gaming Room can confidently proceed with developing a high-quality, cross-platform gaming experience.

## 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)

Developing the "Draw It or Lose It" application in a web-based, distributed environment introduces several key design constraints that have significant implications for development.

1. **Distributed Environment:** The application must operate over the internet, which necessitates a client-server architecture. This means the game logic and data must reside on a central server, while users interact with the game through a client application running in their web browser. The primary implication is the need for constant and reliable network communication. Development must account for potential issues like network latency and data synchronization to ensure a smooth real-time gameplay experience for all users.
2. **Multi-Platform Support:** The client has specified that the game must be accessible on various platforms (e.g., Windows, macOS, mobile devices). To avoid the high cost and effort of building separate native applications for each platform, the application must be built using platform-agnostic web technologies like HTML, CSS, and JavaScript. The user interface must be designed to be responsive, automatically adapting to different screen sizes and resolutions to provide a consistent user experience on any device.
3. **Unique Naming:** A core requirement is that all game and team names must be unique. This constraint requires implementing a validation system that checks any proposed name against all existing names before it is approved. This directly impacts the design of the data management system, which must be optimized for efficient searching and retrieval to prevent delays during game setup. The Iterator pattern will be used to traverse the lists of existing games and teams to enforce this rule.
4. **Single Game Service Instance:** The application logic must ensure only one instance of the core game service can be running in memory at any time. This is a critical constraint for maintaining a consistent state across all games. The

Singleton design patterns must be implemented for the GameService class. This pattern restricts the instantiation of the class to a single object, providing a centralized point of control for managing game sessions and player data.

## [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 provided UML class diagram illustrates the architecture for the "Draw It or Lose It" application. It consists of seven classes that define the structure, relationships, and behaviors of the system.

**Class Relationships and Structure:**

The diagram is logically divided into two main parts: the application drivers and the core game entities.

* ProgramDriver and SingletonTester: These classes are responsible for running and testing the application. The

ProgramDriver contains the main() method which starts the program, and it uses the SingletonTester to verify that the GameService is correctly implemented as a single instance.

* Entity Class: This class serves as an abstract base for the main components of the game. It contains common attributes like an

id and name.

* Game, Team, and Player: These classes represent the core objects in the game. They are connected through

**inheritance**, as each one extends the Entity class. This "is-a" relationship (e.g., a

Game is an Entity) is a fundamental object-oriented principle that promotes code reuse.

* GameService: This class acts as the central hub for managing the entire game.

The classes are also linked through composition, where one object is composed of others. The

GameService holds a list of Game objects, each Game holds a list of Team objects, and each Team holds a list of Player objects. This models the natural hierarchy of the game.

**Object-Oriented Principles:**

The diagram effectively demonstrates several key object-oriented programming (OOP) principles to meet the software requirements:

1. **Inheritance:** The Game, Team, and Player classes inherit from the Entity base class. This is highly efficient as it avoids code duplication by placing the common

id and name attributes in a single parent class, making the system easier to maintain.

1. **Encapsulation:** All class attributes (e.g., -id:long, -name:String) are marked as private (-), meaning they can only be accessed through public methods (+), such as getId() and getName(). This protects the internal state of the objects from unintended outside modification and ensures data integrity.
2. **Singleton Pattern:** The GameService class is a prime example of a design pattern. It has a private constructor and a public static

getInstance() method. This design guarantees that there will only ever be one

GameService object, fulfilling the client's critical requirement for a single, centralized point of control for all game logic and data.

**"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 is built on a stable, UNIX-like foundation, offering good security. However, it is not a conventional choice for hosting scalable web applications. The primary weakness lies in the mandatory use of expensive Apple hardware and a smaller server administration community, making it a less cost-effective and flexible option compared to other platforms. | Linux is the industry standard for web servers. Its primary advantages are that it is open source (no licensing costs), exceptionally stable, and highly secure. With a vast ecosystem of tools (Apache, Nginx) and comprehensive support from all major cloud providers, it offers unparalleled flexibility and scalability. The main weakness can be a steeper learning curve for administrators unfamiliar with a command-line interface. | Windows Server offers a robust and user-friendly hosting environment with a familiar graphical user interface (GUI) and excellent support from Microsoft. Its main advantage is seamless integration with the .NET framework and other Microsoft products. However, the significant licensing costs and higher resource consumption make it a more expensive and potentially less performant choice for typical web hosting compared to Linux. | Mobile devices are fundamentally unsuitable for hosting a web-based application server. They are designed as clients and lack the processing power, memory, reliable 24/7 connectivity, and server-grade operating system features required to handle multiple concurrent users. Attempting to use a mobile device as a server would result in poor performance, instability, and security vulnerabilities. |
| **Client Side** | For a web-based game, the client on a Mac is simply a web browser like Safari or Chrome. Therefore, development considerations are minimal and focused on ensuring cross-browser compatibility. No specific Mac development expertise is needed, making the cost and time equivalent to supporting any desktop browser. | Like macOS, supporting Linux clients involves ensuring the web application works correctly in browsers like Firefox and Chrome. The development effort is focused on web standards and responsive design rather than any Linux-specific considerations, resulting in no additional cost or time. | As the dominant desktop operating system, Windows is a primary target for client-side testing. The development considerations are centered on ensuring the web application is fully compatible with popular browsers like Edge, Chrome, and Firefox. The cost, time, and expertise required are baseline for standard web development. | Mobile devices are a critical client platform. Development must follow a "mobile-first" responsive design strategy to ensure usability on smaller touchscreens. This increases development time and cost, as it requires specialized expertise in creating fluid layouts, optimizing performance for lower-powered hardware, and ensuring a seamless touch-based user experience. |
| **Development Tools** | The core back-end language, Java, is platform-independent. Developers on Mac can use standard, powerful IDEs like IntelliJ IDEA and Eclipse. For the front end, web development tools like Visual Studio Code are readily available. There are no platform-specific tool constraints. | Linux offers excellent support for Java development with access to the same leading IDEs, such as Eclipse and IntelliJ IDEA. The open-source nature of the platform means a vast array of free and powerful development and testing tools are available, making it a highly efficient environment for developers. | Windows provides a robust environment for Java development with full support for all major IDEs. The front-end development stack (HTML, CSS, JavaScript) and associated tools are platform-agnostic, meaning there is no difference in the toolchain required compared to Mac or Linux. | The client application for mobile devices will be built using standard web technologies (HTML, CSS, JavaScript). Development will utilize frameworks like React or Angular to create a responsive user interface. Testing is performed using browser-based mobile simulators and on physical devices to ensure compatibility and performance. |

## 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**: Based on the evaluation of server-side platforms, the clear recommendation is to host the "Draw It or Lose It" web application on a Linux-based server (e.g., Ubuntu Server or CentOS). The analysis in the "Evaluation" section highlights that Linux is the industry standard for web servers, offering unparalleled flexibility, stability, and security. Crucially, it is open source, which eliminates all software licensing costs, a significant advantage over the expensive Windows Server alternative. This platform provides a mature and high-performance environment for running the Java-based backend.

**Operating Systems Architectures**: The recommended Linux operating system uses a modular monolithic kernel architecture. This design is layered:

* **The Kernel:** This is the core of the OS. It directly manages the server's hardware, system processes, memory, and the file system. It provides the foundational services that the game server will rely on for every operation.
* **The Shell (e.g., Bash):** This provides a command-line interface for administrators to manage the server, install software, and configure security settings.
* **User Space / Applications:** This is where the "Draw It or Lose It" application will run. The Java Virtual Machine (JVM) will run as a process in the user space, executing the game logic. A web server (like Apache or Nginx) will also run here, handling incoming network requests from users and serving the client-side (HTML, CSS, JavaScript) files.

**Storage Management**: The recommended operating platform provides robust storage management through two primary components:

1. **File System:** The server should use a journaling file system like ext4. This is the modern standard for Linux and provides high performance and reliability. Journaling ensures data integrity by logging changes before they are committed, which helps prevent data corruption in the event of an unexpected server shutdown.
2. **Database System:** To manage application data such as user accounts, player lists, and the unique game and team names a Relational Database Management System (RDBMS**)** is required. An open-source solution like PostgreSQL or MySQL is recommended. These systems integrate seamlessly with Linux and Java, are highly scalable, and are optimized for the efficient data retrieval and validation (like checking for unique names) that the application requires.

**Memory Management**: The Linux platform uses advanced memory management techniques, but it's important to understand how they interact with the Java application:

1. **OS-Level (Linux):** Linux uses virtual memory and paging. This means each process, including the Java Virtual Machine (JVM) running the game, gets its own isolated virtual address space. The OS "pages" data from the hard drive (from a "swap space") into physical RAM as needed. For a real-time game, it is critical to have sufficient physical RAM to avoid swapping, which would introduce severe performance lag.
2. **Application-Level (JVM):** The "Draw It or Lose It" application code runs *inside* the JVM. The JVM requests a large block of memory (the Java heap) from the Linux OS. The JVM then manages this heap itself, allocating memory for all game objects like Game, Team, and Player.
   * The application's Singleton design for the GameService ensures that this single, critical object is created once and persists in the heap, managing all game sessions.
   * The JVM's Garbage Collector will automatically reclaim memory from completed game sessions and objects that are no longer referenced, preventing memory leaks and ensuring the server remains stable.

**Distributed Systems and Networks**: The application will operate on a client-server distributed model, which is necessary to support clients on multiple platforms.

* **Server:** The recommended Linux server will act as the central authority. It will host the Java backend (including the GameService), which contains all game logic and manages the "single source of truth" for game state. It also hosts the database.
* **Clients:** The clients are the web browsers running on users' devices (Mac, Windows, mobile). They will run the HTML/CSS/JavaScript front end.
* **Communication:** The client and server will communicate over the internet via standard web protocols. A RESTful API should be used for actions like creating a game or adding a player. For the real-time drawing component, WebSockets are recommended for a persistent, low-latency, two-way connection.

**Security**: Security is essential and must be handled in layers:

1. **Data in Transit:** All communication between the client's browser and the Linux server must be encrypted using HTTPS (SSL/TLS). This prevents attackers from "sniffing" network traffic to steal user data or hijack game sessions.
2. **Data at Rest:** All sensitive user information (like passwords or personal details) stored in the database must be secured. Passwords must be hashed and salted (using an algorithm like bcrypt) to make them unreadable even if the database is compromised.
3. **Platform & Network Security:** The recommended Linux OS provides powerful, built-in security features:
   * **Firewall:** A firewall (like ufw or iptables) must be configured to block all incoming network ports except those necessary for the application (e.g., port 443 for HTTPS) and administration (port 22 for SSH).
   * **User Permissions:** The Java game server must *not* run as the "root" (administrator) user. It should run as its own unprivileged user, which severely limits an attacker's ability to harm the system if they find an exploit in the game's code.
   * **Regular Patching:** The server's operating system and all its software must be updated regularly to protect against known vulnerabilities.