

Draw It of 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 | 07/20/25 | David Morelli | Initial version of the software design document for *Draw It or Lose It*, including executive summary, requirements, constraints, domain model, evaluation, and recommendations. |

**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 has requested assistance in transforming their current Android-only game, *Draw It or Lose It*, into a web-based application that can run across multiple platforms. The game allows multiple teams to compete in guessing stock image-based drawings under time constraints. To meet the client’s goals, this project proposes a scalable and maintainable software solution using object-oriented principles and design patterns. The application will ensure that each game, team, and player is uniquely identified and managed in memory. Key patterns used in this design include the **Singleton** pattern for managing a single instance of the game service and the **Iterator** pattern to check for name uniqueness. This software design will provide the flexibility needed to expand to other platforms while maintaining performance, security, and usability.

## Requirements

**Business Requirements:**

* Expand Draw It or Lose It beyond Android to support web-based access on multiple platforms (Windows, macOS, Linux, and mobile).
* Maintain a consistent game experience regardless of platform.
* Ensure unique names for games, teams, and players to prevent conflicts.
* Allow multiple teams to compete per game, with multiple players per team.
* Deliver a scalable and maintainable solution that supports future growth.

**Technical Requirements:**

* Ensure only one instance of the game runs in memory (Singleton pattern).
* Use an iterator-based approach to prevent duplicate names across games and teams.
* Design the system using object-oriented programming principles.
* Host the application in a web-based, distributed environment.
* Securely manage user sessions and game data across platforms.
* Support real-time interaction between clients over a network.

## [Design Constraints](#_2et92p0)

The *Draw It or Lose It* game must be developed for a web-based distributed environment, which introduces several design constraints. These include ensuring consistent game state across different platforms, managing concurrent access, and maintaining a single instance of the game service. The application must operate securely and efficiently over the internet, which limits certain local system dependencies and requires careful management of user sessions and data synchronization.

The system must be designed using object-oriented principles and implemented with support for the Singleton and Iterator design patterns. These constraints affect how instances of the game, teams, and players are created, stored, and accessed. In addition, the system must be deployable across multiple operating systems, so platform-independent tools and languages (such as Java and web technologies) should be used to ensure compatibility and maintainability.

## [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 for *Draw It or Lose It* illustrates the object-oriented structure of the system. At the core of the model is the abstract Entity class, which provides shared attributes such as id and name and is extended by Game, Team, and Player classes. This demonstrates **inheritance**, promoting code reuse and consistency.

The GameService class is implemented as a **Singleton**, ensuring that only one instance of the game service is active in memory. This pattern enforces centralized control over game creation, team registration, and player management, which is crucial in a distributed environment where concurrent sessions must interact with the same instance.

The **Iterator** pattern is applied through the addGame() and getGame() methods, allowing the system to loop through existing game instances to check for name uniqueness. This maintains data integrity by preventing duplicate games or teams from being created.

The model uses **encapsulation** by keeping fields private and exposing functionality through methods, and it supports **polymorphism** through the shared interface of the Entity subclasses. These principles work together to create a flexible, scalable, and maintainable application structure that supports the client’s goals.

**"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 be used for server hosting but is not ideal due to high costs, limited cloud support, and licensing constraints. Not typically recommended for scalable web apps. | Linux remains the top choice for server hosting. It supports open-source stacks (e.g., Apache, Nginx, MySQL) and is ideal for cloud environments like AWS and Azure. Cost-effective and secure. | Windows Server offers a good GUI-based environment and is common in enterprises, but licensing costs are higher, and it’s less ideal for open-source tools. | Mobile devices are not used for hosting. Instead, the server must be optimized for mobile network requests, minimal latency, and load balancing. |
| **Client Side** | macOS clients require Safari testing and integration with system features. Web apps work well but some graphical/animation-heavy features may require optimization. | Linux users typically access apps through browsers like Chrome or Firefox. Development is inexpensive, though desktop usage is lower compared to Mac/Windows. | Windows has the largest desktop market share. Web-based UI should support Edge and Chrome, and performance must be tuned for lower-end hardware in some cases. | Must support both Android and iOS with responsive UI. Native development requires platform-specific skills; cross-platform frameworks reduce effort. |
| **Development Tools** | macOS supports IntelliJ, Xcode, VS Code, and JavaScript frameworks. Best for iOS app development. Licensing required for publishing to Apple App Store. | Linux supports open-source tools like Eclipse, VS Code, and terminal-based Git. Strong community support makes it ideal for cost-effective full-stack development. | Visual Studio, Eclipse, and proprietary tools are commonly used. Good support for Java, C#, and web development but higher licensing costs. | Android Studio (Java/Kotlin), Xcode (Swift), Flutter, and React Native are common tools. Cross-platform tools can reduce duplication of effort across iOS/Android. |

## Recommendations

**-Operating Platform**Linux is the most appropriate operating system for the server platform due to its proven stability, security, scalability, and cost-effectiveness. Linux offers strong support for open-source technologies, containerization (e.g., Docker), and cloud environments (AWS, Azure, GCP), which are essential for a distributed game application. For the client side, a responsive web-based interface should be developed using HTML5 and JavaScript frameworks. This ensures compatibility across major browsers on Windows, macOS, Linux, and mobile devices. For native mobile applications, cross-platform tools such as Flutter or React Native can be leveraged to minimize development effort and maintain a consistent user experience across Android and iOS.

**-Operating System Architectures**A client-server architecture is recommended. The backend server will run on Linux, where the application logic and database are hosted. This centralized approach simplifies updates, ensures consistent game states, and enables secure communication. The server will handle real-time game management, user authentication, and persistent storage. Clients (desktop or mobile) will communicate with the server through REST APIs for general functions and WebSockets for live gameplay updates. This layered architecture separates concerns between presentation, logic, and storage, improving maintainability and scalability.

**-Storage Management**A relational database management system (RDBMS) such as PostgreSQL or MySQL should be used to store structured game data, including user accounts, teams, and game history. These databases provide indexing, referential integrity, and transactional reliability. For high availability, database replication and automated backups will be implemented. File storage, such as game assets or user uploads, should be managed using a distributed file system or cloud storage service with redundancy and versioning to prevent data loss. Caching systems (like Redis) will be incorporated to reduce database load and improve performance for frequently accessed data.

**-Memory Management**The Java Virtual Machine (JVM) provides automatic memory management through garbage collection, which reclaims unused objects to free memory. This ensures efficiency when handling multiple concurrent sessions of the game. The Singleton design pattern will be used for the GameService to prevent redundant instances in memory, maintaining a single authoritative game state. Additionally, caching strategies will be applied to minimize repetitive computation and database queries. Heap memory allocation will be tuned for optimal performance, and monitoring tools will track memory usage to prevent leaks or performance degradation during peak loads.

**-Distributed Systems and Networks**To support real-time multiplayer interaction across platforms, distributed software and networking technologies will be employed. WebSockets will provide low-latency, bidirectional communication for gameplay updates such as drawing and guessing. REST APIs will handle non-real-time operations like user registration or retrieving game history. Load balancers will distribute requests evenly, while auto-scaling ensures responsiveness during spikes in user activity. Failover strategies, such as replicated servers and health monitoring, will mitigate outages. Data consistency across distributed nodes will be maintained using synchronization mechanisms and eventual consistency models where applicable.

**-Security**Security is critical to protecting user data and ensuring fair gameplay. All communication will be encrypted using HTTPS with TLS. Authentication will use secure token-based methods such as OAuth 2.0 or JWT, protecting sessions from hijacking. Sensitive user information will be stored in encrypted form, and inputs will be validated to prevent SQL injection, cross-site scripting (XSS), and other vulnerabilities. Role-based access control (RBAC) will restrict administrative functions to authorized users. On the server side, Linux security tools such as firewalls, SELinux policies, and intrusion detection systems will provide an additional layer of protection.

**-Development Considerations:**

* **Cost:** Open-source tools (Linux, Java, Eclipse, React) keep costs low. Apple's development and publishing tools are the most costly.
* **Time:** Web and cross-platform mobile frameworks speed up deployment. Native app development is slower but sometimes necessary for performance.
* **Expertise:** Teams need knowledge of web backends, front-end frameworks, and mobile development. Consider training or hiring additional staff with cross-platform development experience.