

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 | 05/23/2025 | Justin Turner | Initial Software Design Blueprint for Draw It or Lose It |

**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)

This UML diagram outlines the core architecture behind Draw it or Lose it, a multiplayer game platform designed to efficiently manage the relationships between games, teams, and players. At the center of the system is the GameService class, implemented as a Singleton to ensure a single, consistent instance is responsible for all game logic, including the creation of new games and the assignment of unique IDs to games, teams, and players. The design follows an inheritance structure where Game, Team, and Player all extend from a common Entity class, standardizing attributes like IDs and names across the system. Execution begins through the ProgramDriver class, which validates the Singleton behavior via SingletonTester. To successfully build on this structure, it is essential to rely on the getInstance() method for accessing GameService, as direct instantiation will compromise the Singleton pattern. Additionally, developers must allow the system to manage all ID assignments to prevent data collisions and ensure that entities are created in the correct order games first, then teams within games, and finally players within teams. The diagram presents a streamlined, scalable foundation that supports clean code, predictable behavior, and future expansion of the game environment.

## Requirements

The vision behind Draw it or Lose it is clear, build a multiplayer gaming system that is as sharp behind the curtain as it is fun on the screen. The client is not just looking for functionality, they need structure, stability, and smart design that can grow with the game. On the business side, which means the system must be easy to manage, efficient to scale, and solid enough to manage expanding player bases without falling apart. Technically, there is no room for shortcuts. The backbone of this project is a Singleton-driven GameService, ensuring that all games, teams, and players are born from one authoritative source. IDs are assigned automatically to keep the system clean and conflict-free. Inheritance ties everything together under a shared Entity structure, setting the stage for fast updates and a clear chain of responsibility. Everything starts with ProgramDriver, which kicks the tires on the Singleton pattern before the system gets up to speed. Bottom line: this is not just code, it is a carefully designed ecosystem built to deliver precision and play at scale.

## [Design Constraints](#_2et92p0)

Designing Draw it or Lose it for a web-based, distributed environment brings both opportunity and responsibility. This is not a local app it is a system that has to perform reliably across different machines, networks, and potentially hundreds of concurrent users. That reality sets the stage for some hard design constraints. The use of a Singleton GameService, while effective in a controlled environment, introduces complexity when scaling across distributed systems, where managing a sole source of truth requires careful synchronization or alternative stateless patterns. Additionally, objects like Game, Team, and Player while well-structured for local instantiation must be designed with serialization, network latency, and data consistency in mind. Security, too, is non-negotiable; user data and in-game states must be protected across requests. These constraints mean the architecture must be lean, modular, and ready to separate logic from presentation, requiring API-driven communication and cloud-friendly infrastructure. In short, this is not just about building a game it is about architecting a responsive, secure, and scalable system that plays well across the web.

## [System Architecture View](#_ilbxbyevv6b6)

N/A

## [Domain Model](#_8h2ehzxfam4o)

The UML class diagram for Draw it or Lose it lays out an object-oriented structure where Game, Team, and Player all inherit from a shared Entity class, promoting reuse and consistency. GameService acts as a Singleton, managing all game elements from a single, centralized instance. Each class maintains its own internal lists games contain teams, teams contain players showing clear relationships and solid encapsulation. The use of inheritance, encapsulation, and the Singleton pattern ensures the design is efficient, scalable, and easy to manage. It all kicks off through ProgramDriver, which checks the system’s stability via SingletonTester, making sure everything runs as expected from the start.

**"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)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Great for development thanks to its Unix base but not built for hosting at scale. Limited enterprise server tools and hardware support. | Lightweight, powerful, and secure. It is the go-to for hosting web apps. Steeper learning curve, but worth it for performance and flexibility. | Easy to manage with strong .NET support. Ideal in enterprise setups, but licensing and overhead can be costly. | Not built for hosting but can support local or peer-based services in specific use cases. Limited by hardware and network constraints. |
| **Client Side** | Stable and secure for client use, but more expensive to scale. Great for creative and development-focused users. | Open-source and efficient, perfect for technical users. Less friendly for non-tech clients, which could mean more support needs. | Universal compatibility and familiarity make it ideal for broad client support. Requires consistent updates and security oversight. | Essential for today’s users. Demands responsive design and constant testing across platforms. Fast-moving, but rewarding market. |
| **Development Tools** | Xcode leads to iOS devs, with plenty of cross-platform tools available. Some limitations with backend toolsets. | Dev favorite. Terminal power, full control, and top IDE support make it unbeatable for backend and open-source projects. | Strong IDEs like Visual Studio. Versatile, but may need workarounds for some open-source tools. | Coding happens off-device using IDEs like Android Studio or Xcode. Real devices are used for testing and deployment. |

## Recommendations

1. **Operating Platform**: To support the growth of Draw It or Lose It across computing environments, I recommend adopting Linux as the core operating platform. It is lightweight, scalable, open-source, and ideal for running distributed applications, making it a smart choice for cloud deployment and cross-platform compatibility.
2. **Operating Systems Architectures**: Linux’s monolithic architecture allows efficient system call management and customization. It supports both x86 and ARM-based environments, which is crucial for flexibility in web and mobile deployment.
3. **Storage Management**: For storage management, Linux pairs well with modern file systems like ext4 or XFS, offering reliability, journaling, and support for high-throughput data operations perfect for user-generated content and session data in multiplayer gaming.
4. **Memory Management**: On the memory management side, Linux utilizes demand paging, swapping, and slab allocation to optimize RAM usage. This ensures Draw It or Lose It remains responsive even with fluctuating loads and concurrent user sessions.
5. **Distributed Systems and Networks**: In terms of distributed systems and networks, Linux supports RESTful APIs, WebSockets, and container orchestration with Kubernetes and Docker, enabling seamless communication between web, desktop, and mobile clients even under network strain or partial outages.
6. **Security**: Security is managed robustly through built-in tools like SELinux, iptables, and encrypted communication over SSL/TLS. With support for OAuth2 and secure token-based authentication, user data remains protected both at rest and in transit, across every platform.