

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 | 09/18/2025 | Gerus Hays | Added Executive Summary; captured business & technical requirements; documented web-based design constraints; outlined system architecture; and explained the Domain Model with OO principles (encapsulation, inheritance, composition, singleton). |

**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 wants to take its Android-only party game, Draw It or Lose It, to the web so anyone on a phone, tablet, or desktop can join without installing an app. The core gameplay doesn’t change teams guess a phrase, title or thing while drawings reveal over 60 seconds, with a single guess window at the end. What does change is the platform and the way we handle scale, identity, and fairness in a browser-based, multi-user setup.

My solution is a simple, distributed web application with a singleton game service on the server that owns all states like games, teams, players and unique IDs. Clients are stateless beyond a session token; the server enforces one instance per unique name and guarantees unique IDs for every game, team, and player. Real-time updates are delivered via WebSockets (like TCP connection allowing users to play together) so teams see drawings and timers in sync. We’ll deploy behind a load balancer, persist game metadata in a relational store, cache images, read-heavy data like player lobby names. Security like, input validation, rate limiting on guesses can be built in. The outcome would be a responsive, fair, and scalable browser version that’s easy to maintain and ready to add new rounds or rule tweaks later.

## Requirements

**Business**

* Support multiple games at once; each game can have one or more teams, and each team can have multiple players.
* Enforce unique names like games, teams and players, so players don’t join the wrong lobby.
* Keep the original flow: 4 rounds, 60-second reveal, final 15-second guess window, one guess per remaining team.
* Make joining easy: no app store dependency; users join from a link/room code on modern browsers.
* Keep moderation tools: host can create/end games, add/remove teams, kick players, and lock guesses.

**Technical**

* Server-side authorizing state with a singleton service managing IDs and instances.
* Real-time transport for drawings/timers; fallback when the server isn’t available.
* Concurrent-safe creation of games/teams/players with unique ID generation and name-availability checks.
* Persistence for lobby/game metadata and audit logs; cache memory for faster reads.
* Security: input validation, authorization/session tokens
* Scalability/Deployable: stateless web tier behind a load balancer, structured logging, and metrics.

## [Design Constraints](#_2et92p0)

Distributed web environment: Browsers are untrusted clients. The server must be the source for timers, guess windows, scoring, and unique names. No trusting client clocks or UI state; everything important is validated server-side.

Singleton Game Service: Only one instance coordinates ID counters and the global list of games. Implement as a safe singleton and, for horizontal scale, with a shared store so multiple app nodes stay consistent.

Concurrent and unique: Multiple users can try to create “Team Falcon” at the same time. Protect by checking at the data layer with unique indexes/constraints and synchronize in the service layer to avoid race conditions and duplicates.

Real-time Delivery: Draw frames and countdowns need low latency. Prefer WebSockets; fall back to a poll. Batch messages where possible and cap update rates to protect the server.

Security & Fair Play: Guess spamming or tampering ruins the experience. Rate-limit actions, validate inputs, authorize per-room roles (host, team member), and never trust client timers.

## [System Architecture View](#_ilbxbyevv6b6)

Client (Browser): React-style SPA (or simple JS) that handles input, renders drawings/timers, and opens a WebSocket to the server. It stores only transient UI state plus a session token.

App Server: Java service exposing REST (create/join, admin actions) and channels (room updates, drawing frames, timer ticks). Contains the GameService (singleton) that manages games, teams, players, and ID sequences with thread-safe access.

Data Layer: Relational DB like MySQL with unique constraints on names and sequences for IDs.

Infrastructure: Load balancer/ingress, autoscaling app nodes, centralized logs/metrics/alerts.

## [Domain Model](#_8h2ehzxfam4o)

**Game Service**:

* Entity (id: long, name: String) Base for nameable things with a unique ID. Promotes inheritance and avoids repeating across classes.
* Game (extends Entity) Owns a list of Team objects. Implement operations to add teams and print/debug info. Demonstrates aggregation and encapsulation.
* Team (extends Entity) Owns a list of Player objects. Same pattern with controlled add, string representation.
* Player (extends Entity) A single participant with a unique ID and display name.
* Game Service A singleton that holds the authoritative list of games, manages ID generation (nextGameId, nextTeamId, nextPlayerId), and exposes methods like addGame(name), getGame(id/name), and getGameCount(). This is where we centralize uniqueness checks, thread safety, and lifecycle (create/find).

**OO principles:**

* Encapsulation: Fields are private, exposed through methods, this protects invariants like unique names and valid membership.
* Inheritance: Entity provides common ID/name behavior for Game, Team, and Player, cutting duplication and keeping constructors consistent.
* Composition/Aggregation: Game has Teams; Team has Players. Clear ownership lines make it easy to manage cleanup and traversal.
* Single Responsibility: Each class does one job (service coordinates games; game manages teams; team manages players).
* Singleton Pattern: Ensures only one coordinator for IDs and global collections, which is important for avoiding duplicate instances and keeping state consistent in memory.

**"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** | Fine for building and quick demos, but nobody really hosts large-scale servers on macOS. Macs are great development machines but not ideal for production. They’re expensive, not built for server uptime, and don’t integrate as cleanly with common deployment tools. For The Gaming Room, a Mac-based server would be better suited for local testing or frontend builds, not for hosting the real game. | This is the go-to platform for servers. It’s stable, lightweight, and used by pretty much every major cloud provider. It has strong security tools, open-source flexibility, and no licensing fees, which saves money long-term. It’s ideal for running the Draw It or Lose It backend since it scales easily and can handle thousands of concurrent users without lag. | Works great if your existing team is already used to Windows-based systems. It’s reliable and user-friendly but more resource-heavy than Linux and comes with licensing costs. For a production server, it’s capable but not as cost-effective. | Phones and tablets aren’t built to host full server applications. They don’t have the consistent processing power, storage, or network stability needed for that kind of load. Instead, mobile devices connect to the hosted server through an API or web service. For this project, mobile devices would strictly act as clients accessing the server, not as hosts. |
| **Client Side** | Mostly about making sure the app looks good in Safari and Chrome. Mac users expect clean visuals, smooth animations, and good handling on high-resolution screens. It’s not a big development issue, just something to keep an eye on during testing. | Runs fine in any modern browser, though users are a smaller audience. The main goal here is to ensure browser compatibility, no special coding needed. It’s more of a testing point than a major development focus. | The largest user base by far, so it’s important to test across different screen sizes and resolutions. The interface should scale properly and respond well to both keyboard and mouse input. This platform gives the widest reach, so testing performance here is key. | Phones and tablets are where performance and simplicity matter most. The app should be fast to load, easy to tap through, and forgiving if the connection drops. Design needs to prioritize clarity and touch-friendly navigation to keep it playable on the go. |
| **Development Tools** | Smooth experience for developers. Setup is easy, and tools like Eclipse run great. It’s a solid choice for developers building or testing front-end code. | Excellent for backend work and testing. It’s reliable, runs fast, and supports all the major frameworks. It’s great for team collaboration since most server environments run on Linux anyway. | Works fine for coding with editors like Visual Studio, but setup can be a little heavier. Still, most developers are comfortable here, and it’s easy to manage projects if you’re already using Windows systems across your team. | Not used for actual coding, but useful for testing and debugging how the game feels and looks on smaller screens. It’s basically the final step for ensuring the game translates smoothly from desktop to handheld. |

## 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**: Go with Linux in the cloud for the server side and modern browsers for clients. It’s the simplest, cheapest path to run anywhere and scale up when a bunch of people jump in at once.
2. **Operating Systems Architectures**: Use a classic three-tier setup: stateless web with app nodes, a shared database, and optional cache. Multiple app nodes sit behind a load balancer so if one dies, the game keeps rolling.
3. **Storage Management**: Use a managed relational database like MySQL for games/teams/players and audit logs. Toss any big assets into object storage and add daily backups and point-in-time restore so we can undo mistakes.
4. **Memory Management**: Keep memory usage predictable by keeping rooms small and short-lived: when a game ends, we clean up its lists (teams/players) and drop any timers/sockets tied to it. We cap message sizes, expire idle lobbies, and put a limit on how many active rooms a single server can host; if we get close to that limit, we spin up another server. For data we need briefly (like “is this name taken?”), we cache it with short timeouts, so it doesn’t pile up. So, we avoid leaks, keep footprints small, and scale out smoothly instead of letting one box get bloated.
5. **Distributed Systems and Networks:** Real-time updates over WebSockets, all traffic through a load balancer. Make every request safe to retry if it runs twice, we don’t double-create anything, use quick retries with a short pause, send heartbeats to drop ghost clients, and watch metrics/alerts so we spot slowdowns early.
6. **Security**: We run HTTPS everywhere, so traffic is encrypted and you’re hitting the real site. Players get short-lived session tokens, and every action check role (host vs. player). We validate inputs and rate-limit noisy actions like guesses/name checks to stop spam. Keys or tokens live in secure config, not code. We log only what’s needed and keep an eye on alerts so we can get rid of anything corrupting or ruining the game fast.