

**The Gaming Room – Draw It or Lose It (Web App)**

# **CS 230 Project Software Design Template**

Version 1.0

## Table of Contents

[**CS 230 Project Software Design Template**](#_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 4**](#_Toc115077323)

[**Domain Model 4**](#_Toc115077324)

[**Evaluation 4**](#_Toc115077325)

[**Recommendations 7**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 10/18/25 | Hilnecia Larry | Final Submission — Complete document (Executive Summary through Recommendations). |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room plans to expand its Android-based game *Draw It or Lose It* into a web-based, multi-platform application. The goal is to design a system that allows multiple teams, each with multiple players, to participate in guessing puzzles. To meet the client’s needs, the solution must ensure that game, team, and player names are unique, and that only one instance of the game can exist in memory at any given time.

The proposed solution leverages a centralized Java service layer that manages games, teams, and players through the GameService class. By applying the Singleton design pattern, the system guarantees that only one instance of the game exists. The use of iterators and unique identifiers will enforce name uniqueness and maintain efficient access across all entities. This design ensures scalability, cross-platform compatibility, and long-term maintainability, while also creating a foundation for secure, reliable gameplay across web and mobile clients.

## Requirements

**Business Requirements**

* Support multiple teams and multiple players per team.
* Ensure each game, team, and player name is unique so users can easily identify and join entities.
* Guarantee that only one active game instance can exist in memory at a time.
* Provide a solution that works across multiple platforms, including web browsers, desktops, and mobile devices.

**Technical Requirements**

* Centralized GameService class to manage creation and access to all entities.
* Use of the **Singleton pattern** to enforce the single-instance constraint.
* Unique identifiers (IDs and names) assigned at creation to prevent duplicates.
* Iterator pattern for adding and retrieving games, teams, and players efficiently.
* Platform-independent server logic to support cross-platform clients.
* Secure communication and input validation to protect data integrity.

## [Design Constraints](#_2et92p0)

Developing *Draw It or Lose It* in a web-based distributed environment requires the following constraints:

* **Single Instance Constraint**: The application must enforce that only one game instance exists in memory by applying the Singleton pattern.
* **Uniqueness Constraint**: All game, team, and player names must be unique, requiring centralized lookup and validation before creating entities.
* **Distributed Environment**: The application must function consistently across multiple platforms, with a centralized server handling all game logic and state.
* **Scalability and Concurrency**: The design must handle simultaneous requests from multiple users while preventing race conditions in entity creation.
* **Security**: Web deployment requires secure protocols (HTTPS/TLS), input validation, and proper authentication and authorization.
* **Maintainability and Portability**: The system should be modular, written in Java, and portable across environments, ensuring future updates and expansions can be made with minimal disruption.

## [System Architecture View](#_ilbxbyevv6b6)

## [Domain Model](#_8h2ehzxfam4o)

**"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.**

The UML diagram shows how the classes in the system relate to one another and how object-oriented principles are applied to meet the software requirements. The Entity class provides the shared attributes id and name, which are inherited by Game, Team, and Player, demonstrating inheritance and reducing duplication. A Game can have multiple Team objects, and each Team can have multiple Player objects, which directly supports the requirement for multiple teams and players in the game. The GameService class manages the collections of games and ensures encapsulation by keeping data private and exposing only controlled methods such as addGame() and getGame(). It also uses the Singleton pattern so that only one instance of the service exists in memory at a time, fulfilling the requirement that only a single active game instance can run. Additionally, addGame(), addTeam(), and addPlayer() iterate over existing entities to enforce the rule of unique names. Together, these relationships and principles—inheritance, encapsulation, abstraction, and Singleton—create an efficient and scalable model that satisfies all of the client’s requirements.

## [Evaluation](#_2o15spng8stw)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | macOS is not a practical choice for production hosting. It requires Apple hardware, is not widely offered by hosting providers, and the desktop operating system is not licensed or supported as a traditional web server platform. While it is possible to run a Java service on macOS for development, using it in production would add hardware cost and limit scaling options. The better path is to use Linux or Windows for the server. | Linux is the strongest option for hosting the web application and scaling to thousands of players. It supports proven stacks like NGINX or Apache in front of a Java service, and it works very well with containers and orchestration for horizontal scaling. Most major clouds offer Linux images and managed load balancers, so it is simple to add more capacity as traffic grows. Linux itself has no license fee when we use community editions, which keeps hosting costs low. If the client wants vendor support, a subscription from Red Hat, SUSE, or Canonical is available at an added cost, but it is optional. | Windows Server can host the site and the Java service without any major issues. It scales on virtual machines and in containers and can sit behind a load balancer just like Linux. The main drawback is cost. Windows Server is licensed per core and often adds client access licenses. In the cloud, those fees are usually included in the instance price, but they still make Windows more expensive than Linux to run at scale. The upside is solid management tools and easy integration in shops that already use Active Directory. | Android and iOS are not server platforms. They are designed for apps on personal devices and do not provide a safe or reliable way to host a public web service. There are no server deployment options and no server licensing to consider here. The mobile role is on the client side only. |
| **Client Side** | On macOS, the app runs in Safari and Chromium-based browsers, so my plan is to ship a single responsive web UI that sticks to web standards and avoids browser-specific tricks. I’ll budget time to test layout, fonts, and keyboard support on Safari since it behaves a little differently than Chrome. To keep things compatible, I’ll transpile modern JavaScript, use semantic HTML, and build accessible forms and navigation. The costs here are mainly developer time and a small QA effort to cover the latest Safari and Chrome on macOS. No extra licenses are needed for the client itself, just steady cross-browser testing and polish. | On Linux desktops, the app runs in Chrome and Firefox, so the same responsive UI will work without a separate build. I’ll focus on standards-based CSS and TypeScript, then verify rendering and input behavior across common Linux distributions. The work is mostly QA time to confirm visuals, fonts, and performance, not new code. There are no client licensing costs on Linux. The main expertise needed is good front-end practice, accessibility checks, and a small test matrix for Chrome and Firefox. | On Windows, the app runs in Edge and Chrome, so I’ll keep the same codebase and validate interaction with mouse, touch, and keyboard on common laptop and desktop setups. I’ll watch high-DPI scaling and ensure the layout holds up on different zoom settings. Like macOS and Linux, the effort is in testing rather than platform-specific coding. There are no client license fees for browsers. The primary cost is time for cross-browser QA and performance tuning so it feels fast on typical Windows hardware. | On phones and tablets, I’ll ship the same site with a mobile-first layout: larger tap targets, short forms, and resilient networking for spotty connections. I’ll test on a small device set to cover current Android and iOS, and I’ll add basic offline friendliness with safe caching where it makes sense. If the client later wants an installable feel, we can layer in PWA features. There are no app store costs for a web delivery. The real cost is QA time for touch input, viewport quirks, and performance on mid-range devices. |
| **Development Tools** | For macOS development I’ll use OpenJDK with Maven or Gradle for the server, and Node.js with a modern bundler for the web client. VS Code or IntelliJ IDEA Community handles the IDE side, which keeps licensing near zero. If we later decide to publish a native iOS wrapper, we’ll add Xcode and enroll in Apple’s developer program, which does carry an annual fee. Team-wise, one cross-functional web team can handle both server and client; a small iOS effort would only be added if native becomes a requirement. | On Linux I’ll use OpenJDK, Maven or Gradle, Node.js, and either VS Code or IntelliJ Community. Docker is straightforward on Linux, which makes local environments and CI builds easy to reproduce. These tools are free, so the main cost is engineer time. A single team can own the full stack and the pipeline, with a light DevOps role to keep builds, containers, and deployments running smoothly. | On Windows I’ll run the same stack: OpenJDK, Maven or Gradle, Node.js, and VS Code or IntelliJ Community. WSL2 helps when I want a Linux-like shell, and Docker Desktop is fine for local containers. Most of these tools are free. Optional costs show up only if the team prefers paid IDEs or enterprise CI services. One full-stack team is still enough; we don’t need separate Windows specialists since we’re delivering a standards-based web client. | Since we’re delivering through the mobile browser, the core tools stay the same: TypeScript, Node.js, the web test stack, and real-device or cloud-device testing. If the client later asks for native shells, Android Studio is free for Android and Xcode is free to install for iOS, with Apple’s developer program fee only when we publish to the App Store. This work would justify a small, focused mobile track, but only if native becomes a firm requirement. Until then, one web team can support desktop and mobile with the same codebase and toolchain. |

## Recommendations

1. **Operating Platform**: I recommend hosting Draw It or Lose It on a Linux-based platform in a major cloud (AWS, Azure, or Google Cloud). Linux gives you the best mix of scalability, cost control, and vendor choice. It is the default target for container orchestration, managed load balancers, and autoscaling groups, and it avoids the per-core licensing overhead that comes with Windows Server. Sticking with Linux also aligns with the Java service you’ve already built and keeps the deployment portable across providers, which helps the client expand to other computing environments without rework. This platform choice lets The Gaming Room expand to additional computing environments without refactoring the service layer or changing the deployment model.
2. **Operating Systems Architectures**: On Linux, the application should follow a simple three-tier, stateless web architecture:

* Edge/Delivery: A managed load balancer in front of an NGINX reverse proxy to terminate TLS, enforce HTTPS, and route traffic to service replicas.
* Application tier: A containerized Java service (the existing service layer) running behind the proxy. Containers make horizontal scaling and zero-downtime updates straightforward.
* Data tier: Managed services for persistence—PostgreSQL for relational data (games, teams, players), Redis for short-lived caching (session tokens, rate limits), and object storage for images (see storage section below).

1. **Storage Management**: The game needs fast, reliable delivery of a large image library. Use cloud object storage (for example, S3, Azure Blob, or GCS) as the system of record, fronted by a global CDN for low-latency reads. Enable versioning and lifecycle policies so rarely used images move to lower-cost tiers automatically, while frequently used assets remain hot on the CDN. Keep metadata (IDs, tags, usage count) in PostgreSQL and reference images by immutable keys. Using managed object storage also gives us built-in durability guarantees (eleven-nines class) and easy cross-region replication if we need it. For uploads or updates, write once to object storage, invalidate the CDN path, and let the cache refresh naturally. This approach scales cheaply, survives node failures, and avoids filling application disks.
2. **Memory Management**: At the application level, keep services stateless and memory-lean. Stream images from object storage and the CDN rather than loading full files into memory; when an in-memory cache is justified, cap sizes and use LRU eviction to protect memory headroom. Tune the JVM with sensible heap sizes, G1/Generational GC, and metrics on allocation rates. Use connection pools for the database and Redis, and time-bound all I/O to prevent threads from piling up. Finally, set autoscaling rules on CPU and memory pressure so new containers come online before the service approaches saturation. These practices keep latency consistent during peak play and prevent out-of-memory failures.
3. **Distributed Systems and Networks**: To support play from browsers on Linux, macOS, Windows, and mobile, expose a RESTful HTTPS API from the Java service. For real-time updates like turn changes or countdown timers, add WebSocket endpoints (or server-sent events) alongside REST. Place a managed load balancer in front, with health checks and rolling deployments. Build resilience into the client and server: idempotent endpoints, retries with back-off, and circuit breakers for downstream calls (database, Redis, storage). Log to a centralized service and set alerts on latency, error rates, and saturation. This combination tolerates transient outages, maintains state consistency, and keeps the experience responsive across platforms.
4. **Security**: Security needs to be consistent in transit and at rest. Terminate TLS at the edge and enforce HTTPS only. Use OAuth 2.0 with short-lived JWTs for session tokens, backed by role-based access control so admins, hosts, and players have scoped permissions. Validate and sanitize all inputs at the API, apply rate limiting per IP/user, and protect image requests with signed URLs or least-privilege CDN policies. Encrypt data at rest in PostgreSQL and object storage, rotate keys with a cloud secrets manager, and keep production secrets out of code and CI logs. Finally, capture structured audit logs for authentication and game-state changes to support investigations and compliance needs. These measures protect user information both on the server and as it moves between platforms.