

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 | <09/12/25> | Maximo Winfield | Initial draft of the full software design template |

**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 seeks to expand its Android game *Draw It or Lose It* into a **web-based, multi-platform** application. The goal is to preserve the fast-paced team-based drawing and guessing experience while enabling play from any modern device (Windows, macOS, Linux, or mobile). The new system must allow multiple teams and players per game, enforce **unique names** for games and teams, and guarantee that **only one instance of the game service** exists in memory at any time.  
 To meet these needs, this design recommends a **distributed client–server architecture** hosted in a secure cloud environment. A RESTful API will provide a single authoritative game service, and a relational database will maintain player, team, and game data. Applying proven software design patterns such as **Singleton** and **Iterator** will ensure that the game service enforces uniqueness and supports concurrent users efficiently.

## Requirements

### Business Requirements

* **Multi-team gameplay:** A game supports one or more teams.
* **Multiple players per team:** Each team can have several players.
* **Unique naming:** Game names must be globally unique; team names must be unique within a game.
* **Single authoritative service:** Only one running instance of the game service manages all games and IDs.
* **Web-based access:** Users play from modern browsers on Windows, macOS, Linux, and mobile.
* **Fast, fair play:** Gameplay must feel real time for all teams during timed rounds.

### Technical Requirements

* **Client-server architecture:** Browser clients call a central REST API over HTTPS.
* **Singleton pattern:** GameService is a singleton to enforce a single in-memory authority.
* **Iterator pattern:** Used when adding or retrieving entities to verify unique names.
* **Base entity model:** Entity class provides id and name; Game, Team, and Player inherit from it.
* **Auto IDs:** Unique numeric IDs are generated for each game, team, and player.
* **Data storage:** Central database persists users, teams, games, and state.
* **Concurrency:** Support many simultaneous sessions with consistent game state.
* **Security:** TLS for transport, input validation, basic authentication, and role-based access.
* **Cross-platform UI:** Responsive web UI that works across major browsers and mobile screens.
* **Observability:** Basic logging of requests, errors, and game events for troubleshooting.

### Out of Scope (for this phase)

* Hardware sizing and deployment automation
* Native mobile apps (web access only)

## [Design Constraints](#_2et92p0)

Because *Draw It or Lose It* will operate in a **web-based distributed environment**, development must address several key constraints:

* **Single Instance Enforcement** – Only one instance of the game service may exist at a time. The Singleton pattern ensures a single authoritative object manages all game sessions and identifiers.
* **Unique Names** – Game and team names must remain unique across the distributed system. The Iterator pattern allows the service to traverse collections and validate uniqueness before adding new entities.
* **Cross-Platform Compatibility** – Clients will access the game from different operating systems and browsers, requiring adherence to open standards (HTTP/HTTPS, JSON) and responsive design.
* **Concurrency and Latency** – Multiple teams will play simultaneously. The system must handle concurrent requests while minimizing network latency to keep gameplay fair and real-time.
* **Security** – User data and game states must be protected with encrypted communication (TLS/SSL) and secure authentication.

These constraints shape the choice of technologies (e.g., Java or similar server-side language, RESTful web services) and guide testing across multiple platforms.

## [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 shows a hierarchical relationship among classes in the com.gamingroom package:

* **Entity (Base Class)** – Holds common attributes id and name and provides getter methods.
* **Game**, **Team**, and **Player** – Each inherits from Entity.
  + Game contains a list of Team objects.
  + Team contains a list of Player objects.
* **GameService** – Maintains collections of games, tracks next available IDs, and provides methods to add or retrieve games. It is implemented as a **Singleton** to guarantee a single service instance.
* **ProgramDriver** – Contains the main() method to start the application.
* **SingletonTester** – Verifies that only one instance of GameService is created.

Object-oriented programming principles demonstrated include:

* **Inheritance** – Game, Team, and Player extend Entity to share common attributes and behaviors.
* **Encapsulation** – Private fields with public getters protect internal data while exposing necessary functionality.
* **Design Patterns** –
  + **Singleton** in GameService ensures one authoritative service object.
  + **Iterator** in addGame() and getGame() methods checks for unique names by traversing collections.

These principles and patterns support efficient resource management, maintainable code, and the software requirements for unique identifiers and single-instance enforcement.

**"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 host a web server stack (Nginx or Apache with a JVM or Node.js), but it is rarely used for production hosting. Apple discontinued macOS Server, so long-term enterprise support is limited. Cloud images for macOS are restricted to Apple hardware, which raises costs and complicates scaling. Licensing comes with the hardware purchase; there is no separate server OS license, but total cost of ownership is high. Verdict: acceptable for development or demos, not recommended for production. | | Linux is the industry standard for web servers. It offers excellent stability, performance, and security with first-class support for Nginx, Apache, Java, Node.js, and container runtimes. All major clouds provide Linux images and managed services that align with autoscaling, load balancing, and IaC. Licensing is typically free for the OS (e.g., Ubuntu, Debian, Rocky), with optional paid support for enterprise distros. Verdict: best choice for production hosting and scaling to thousands of players. | Windows Server supports IIS and reverse proxies for Java or Node.js backends, integrates smoothly with Active Directory, and has strong GUI tooling. It works well in Microsoft-centric shops and supports .NET if a service or admin tool ever needs it. Licensing costs are higher than Linux and should be budgeted per core and per CAL in some scenarios. Verdict: solid option if you need AD and Windows ops alignment, otherwise more costly than Linux. | Mobile devices are not suitable as servers due to limited CPU, memory, storage, battery, and network constraints. They can host local dev servers for testing, but not production. Verdict: not viable for hosting. Use mobile only as clients. |
| **Client Side** | |  | | --- | |  | | Focus on Safari and Chromium testing. Ensure modern browser APIs are used with progressive enhancement. Use a responsive layout, touch and keyboard support, and media optimization so gameplay remains smooth on MacBook displays and external monitors. Cost and time are moderate, mostly in QA coverage. | | Support Chromium and Firefox across popular distros. Variability in font rendering and GPU drivers means extra QA for canvas or WebGL animations. Automated tests in CI plus a small manual QA matrix keeps costs predictable. Linux player base is smaller, but support is straightforward for a web app. | Windows is the largest desktop client segment. Test across Edge and Chrome, and verify high DPI scaling on common laptop resolutions. Input devices vary (mouse, stylus, touch on Surface devices), so verify pointer events and timing accuracy in the game canvas. QA effort is higher here due to user volume and device diversity. | Deliver a responsive, mobile-first web UI with touch targets sized to 44–48 px. Optimize network calls to keep round latency low. Use Add to Home Screen and offline caching for assets via a PWA so re-joins are fast. Validate performance on mid-tier Android devices and recent iPhones. Budget more time for device lab testing and throttled-network checks to guarantee fair timing in one-minute rounds. |
| **Development Tools** | Common tools include Java, Python, or JavaScript with IDEs such as IntelliJ IDEA, Eclipse, or Xcode. | Linux supports Java, Python, Node.js, and many other languages with IDEs such as Eclipse, IntelliJ IDEA, or VS Code. | Windows supports Java, .NET, Python, and Node.js with Visual Studio, IntelliJ IDEA, or Eclipse. | For mobile, developers use Java/Kotlin with Android Studio for Android and Swift/Objective-C with Xcode for iOS. Cross-platform frameworks such as React Native or Flutter can reduce development time. |

## 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**

* **Recommendation:** Run the production backend on **Linux** in a major cloud provider.  
   **Why:** Linux gives the best mix of performance, cost efficiency, portability, and first-class support for container orchestration and managed databases. It is widely used for high-concurrency, low-latency web services and has strong community and vendor support.

## **Operating System Architectures**

* **Server architecture on Linux (64-bit):**
* **Process model:** Stateless API services packaged as Docker containers. One service is authoritative for game state and round timing.
* **Orchestration:** Kubernetes or a managed equivalent for rolling updates, self-healing, horizontal autoscaling, and per-pod resource limits.
* **Networking:** Ingress or an API gateway in front of services, TLS termination at the edge, and internal service discovery.
* **File system:** ext4 or XFS on block volumes for VM nodes and container workloads. Object storage for assets.
* **Observability:** Centralized structured logs, metrics, and traces (for example, OpenTelemetry) to diagnose latency during one-minute rounds.
* **Client architectures:**
* **Desktop browsers (Windows, macOS, Linux):** HTML5 canvas or WebGL for the draw stream, WebSockets for real-time events, progressive enhancement for older browsers.
* **Mobile browsers:** Responsive UI, touch input, network request throttling, asset caching through a PWA to speed reconnects.

## **Storage Management**

* **Operational data (strong consistency):**
* **Relational database:** **PostgreSQL** in a managed service. Use normalized tables for Game, Team, Player, and Round with unique constraints to enforce name uniqueness. Use transactions for atomic create and join flows.
* **Indexes:** B-tree indexes on name and foreign keys. Partial or composite indexes for frequent queries such as “active rounds for game”.
* **Backups and recovery:** Automated daily backups with point-in-time recovery. Target RPO under 5 minutes and RTO under 15 minutes for critical restores.
* **Low-latency state and leaderboards:**
* **In-memory cache:** **Redis** for ephemeral round state, timers, presence, and leaderboard increments. Use TTLs to release memory after rounds complete. Persist authoritative results to PostgreSQL at round end.
* **Idempotency:** Idempotency keys on write endpoints to avoid duplicate inserts during retries.
* **Assets and logs:** Object storage for static assets and archived logs. Use a CDN for image and script delivery. Version assets to allow safe rollbacks.

## **Memory Management**

* **Server heap and allocation:**
* **If Java:** Use a modern JDK with container-aware heap sizing and the G1 or ZGC collectors. Set max heap based on container limits. Avoid large object churn in the hot path.
* **If Node.js:** Avoid blocking I/O. Use a connection pool for the database and reuse buffers for frequent messages.
* **Techniques that improve stability and latency:**
* **Object pooling** for frequently created message frames.
* **Connection pooling** for PostgreSQL to cap concurrent connections.
* **Backpressure** on WebSocket streams when clients fall behind.
* **Circuit breakers and bulkheads** to isolate failures.
* **Cache discipline:** Time-box cached entries and invalidate on authoritative updates to avoid stale reads.

## **Distributed Systems and Networks**

* **Real-time communication:**
* **Transport:** REST over HTTPS for setup flows and metadata. **WebSockets** for live round events, drawings, clocks, and guesses.
* **Authoritative timing:** The server publishes the official clock tick to all clients to ensure fairness. Clients render locally but never advance the round themselves.
* **Message design:** Small JSON or binary frames with versioned schemas. Include sequence numbers to detect loss or reordering.
* **Retries and idempotency:** Clients retry on disconnects. The server treats duplicate messages safely.
* **Outage planning:**
* **Graceful degradation:** If Redis is unavailable, pause new rounds and let active rounds finish based on last authoritative tick.
* **Multi-AZ deployment:** Spread pods and databases across availability zones.
* **Health checks and auto-restart:** Liveness and readiness probes gate traffic during rollouts.
* **Clock sync:** NTP on all nodes to keep logs and metrics aligned.
* **Dependencies and coupling:**
* Keep the game service authoritative and stateful only where necessary. Push all other workloads to stateless services to enable horizontal scaling without complex coordination.

## **Security**

* **Transport and identity:**
* **TLS everywhere:** Enforce HTTPS for clients and mTLS or private networking between services.
* **Authentication and authorization:** Short-lived tokens (for example, OAuth 2.1 or JWT) for players. Role-based access for admins. Rotate keys regularly.
* **Data protection:**
* **Encryption at rest:** Managed disk encryption for nodes, and server-side encryption for PostgreSQL, Redis snapshots, and object storage.
* **PII minimization:** Store only what is necessary for gameplay. Hash any credentials with a strong algorithm if you host auth.
* **Input validation:** Validate all payloads server-side. Enforce strict size limits on uploads and messages.
* **Abuse and integrity:**
* **Rate limiting and bot mitigation:** Per IP and per session throttles for REST and WebSockets.
* **Cheat prevention:** All scoring, timers, and win conditions are computed on the server. Clients cannot alter results.
* **Operational security:** Centralized logging, anomaly alerts, WAF rules for common attacks, and least-privilege IAM for services.
* **Compliance readiness:** Log retention, data export and deletion workflows, and regional data residency settings to support privacy regulations.