THE GAMING ROOM

Draw it or lose it

CS 230 Project Software Design Template

Version 1.0

Table of Contents

Document Revision History

Version	Date	Author	Comments
1.0	11/15/25	Cornell Drakeford	Initial draft: core sections completed

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

The Gaming Room seeks to expand Draw It or Lose It from a single platform Android app into a web based, multi-platform experience accessible via browsers and backed by a scalable distributed system. The solution centers on a stateless, service oriented backend (RESTful APIs with real-time channels), a responsive web client, and an identity aware game orchestration layer that enforces uniqueness and one instance in memory constraints. We will use a shared domain model with a base Entity class to encapsulate identifiers and common behavior, plus dedicated services for game lifecycle, team management, player sessions, puzzle rendering, and scoring. This design supports horizontal scaling, secure multi-tenant operation, and cross-platform communication (desktop and mobile). Critical decisions include using a cloud-hosted Linux environment for server workloads, a CDN for static assets and stock drawings, WebSockets for live play, and a managed database for transactional integrity. With these foundations, The Gaming Room can reliably deliver real-time rounds, unique naming, and single-instance game semantics across diverse client devices.

Requirements

- Business requirements: Enable teams to compete in time-boxed rounds; allow name checking and unique game/team naming; expand reach to web and desktop platforms; provide responsive, low-latency gameplay with simple onboarding and clear round flow; support fair-play rules and auditability.
- Functional requirements:
  - One or more teams per game; multiple players per team.
  - Unique names for games and teams; name availability check.
  - Single active instance per game in memory; unique identifiers for game, team, player.
  - Real-time rendering cadence (images reveal steadily; complete at 30s); timers for 60s round, 15s steal window.
  - Guess submission, validation, scoring, and round progression APIs; lobby creation/join/leave; session management.
- Nonfunctional requirements: Low latency (<200ms event propagation), high availability (99.9% target), horizontal scalability, security (authentication, authorization, data protection), observability (logs, metrics, traces), portability across Mac/Linux/Windows client environments, and maintainability via modular architecture and clear separation of concerns.

Design Constraints

- Web-based distributed environment: Must operate over the public internet with variable latency and intermittent connectivity. Imposes stateless service design with client-side resilience (retry, reconnection, idempotent requests).
- Real-time interactions: Requires server push (WebSockets or SSE) and precise timing. Game state transitions must be atomic to prevent race conditions (e.g., simultaneous guesses or round end).
- Single-instance game rule: Enforce uniqueness in memory via a game registry with locking or transactional guarantees. Sharded/clustered deployments must coordinate across nodes (e.g., distributed lock keyed by gameId).
- Uniqueness of names: Global uniqueness implies consistent data access patterns (unique indexes) and conflict handling during concurrent creation.
- Asset delivery: Large stock drawings require efficient CDN distribution and caching. Rendering cadence must not depend on client CPU/GPU capabilities; fallbacks for low-end devices.
- Security and privacy: Authentication with session tokens, transport encryption (TLS), least-privilege data access, and secure storage of PII. Multi-platform support increases threat surface; consistent policies are needed.

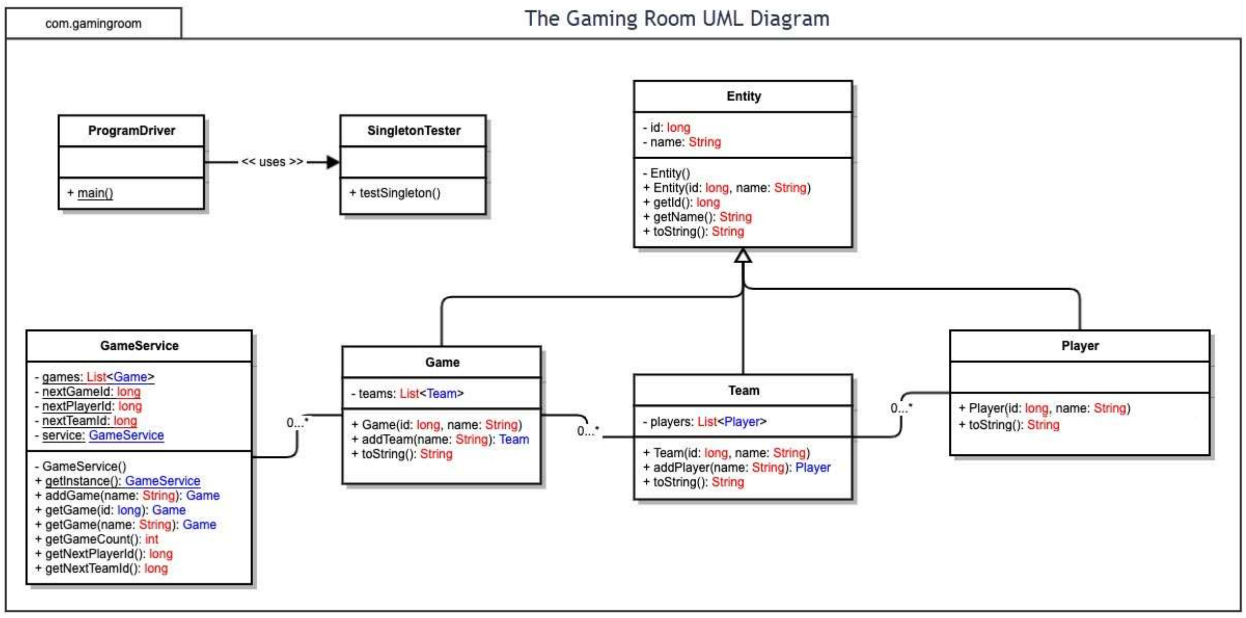
- Portability: Client must run on major browsers and devices; server toolchains should favor Linux for cost and operability but remain portable to Windows/Mac for development.
- Cost and operability: Prefer managed cloud services for database, caching, and observability to reduce operational burden; design for cost-aware scaling (auto-scaling and tiered storage).

System Architecture View

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

- Entity (base class): Provides a unique identifier (e.g., UUID), creation/update timestamps, and possibly equality/hash behavior. All domain classes derive from Entity to standardize identity and lifecycle.
- Game: Aggregates teams, current round state, timers, puzzle reference, and status (active, completed). Manages round flow, rendering cadence, and scoring rules. Has a unique name and enforces single active instance.
- Team: Contains players, team name (unique within the system), score, and participation state. Maintains associations to the Game and supports events like guess submission and roster changes.
- Player: Represents a participant with display name, session info, and team membership. Carries permissions and rate-limits for guesses.
- Puzzle/Clue (or Drawing): References stock drawing assets and solution strings; includes metadata for difficulty and category.
- Relationships:
  - Game 1. Team; Team 1. Player; Game 1...1 current Puzzle per Round; Player .. Team (composition).
  - Entity inheritance demonstrates reuse of identity, timestamps, and shared behaviors.
- OO principles demonstrated:
  - Encapsulation: Game encapsulates timing/scoring; Team encapsulates roster and points; Player encapsulates session state.
  - Inheritance: Entity base class provides common identity and audit fields.
  - Polymorphism: Services/handlers may operate on Entity types or domain events uniformly; different render strategies can implement a shared interface.
  - Composition/Aggregation: Game composes Teams; Teams aggregate Players.
  - Single responsibility: Distinct classes for game orchestration, team management, and player sessions improve clarity and testability.
  - Interface segregation: Separate interfaces for guess handling, timing, and rendering prevent bloated types.
  - Dependency inversion: Application services depend on repositories and event buses via abstractions, enabling swapping implementations (e.g., in-memory vs. database-backed).



Evaluation

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	Mac servers are less common in enterprise hosting. They offer strong integration with Apple ecosystems but limited scalability compared to Linux.	Linux dominates server hosting due to stability, scalability, cost-effectiveness, and open-source flexibility. Ideal for cloud deployments.	Windows Server integrates well with Microsoft tools and enterprise environments but has higher licensing costs.	Mobile devices are not suitable for hosting; limited resources and battery constraints make them client-only platforms.

Client Side	Development for Mac requires macOS compatibility testing. Costs include Apple hardware and expertise in Swift/Objective-C for native apps.	Linux clients are less common but important for developer audiences. Requires testing across distributions and desktop environments	Windows clients are widespread; development must ensure compatibility across versions. Strong support for .NET and C#.	Mobile clients require cross-platform frameworks (Flutter, React Native) or native development (Java/Kotlin for Android, Swift for iOS). Testing fragmentation is a major cost.
Development Tools	Xcode, Swift, Objective-C, and cross-platform frameworks (React Native, Unity).	GCC, Java, Python, Node.js, Eclipse, IntelliJ, VS Code. Strong open-source ecosystem.	Visual Studio, .NET, C#, Unity, and cross-platform frameworks. Rich enterprise tooling.	Android Studio (Java/Kotlin), Xcode (Swift), Unity, Flutter, React Native. SDKs provided by Apple and Google.

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**: Primary recommendation: Host the backend on Linux in a managed cloud environment (Kubernetes or containerized services) for cost efficiency, reliability, and tooling breadth. Use a responsive web client (PWA-ready) to reach Mac, Linux, Windows, iOS, and Android browsers with minimal duplication.

2. **Operating Systems Architectures**: - Backend: Microservices or modular monolith with stateless REST APIs and a real-time gateway (WebSockets). Run containers on Linux nodes; scale horizontally with an ingress controller and service mesh (optional) for resilience. Use distributed locks or a coordinator service to enforce “single active game instance” semantics across replicas.
3. - Frontend: SPA/PWA with a component-based framework (React, Vue, or Angular). Service worker for caching and offline tolerance; responsive layout for mobile and desktop.

4. **Storage Management**: - Transactional storage: Managed PostgreSQL with unique indexes on game_name and team_name to enforce global uniqueness; ACID guarantees for state transitions and scoring.

5. **Memory Management**:

Server memory: Maintain minimal in-memory state per game using Redis as the authoritative transient store. Use efficient object pooling in hot paths, bounded queues for events, and per-game TTL to free memory after completion. Avoid memory leaks via rigorous lifecycle hooks and telemetry alerts.

6. **Distributed Systems and Networks**:

- Real-time communication: WebSockets with backpressure handling; fall back to SSE/long-polling where necessary. Use a message bus (e.g., Redis pub/sub) to broadcast state changes to subscribed clients.

7. - Resilience:
   Idempotent APIs for guess submissions; retries with exponential backoff; distributed locks for round transitions. Implement session rejoin logic on reconnect and authoritative server-side timers.
8. - Dependencies and outages:
   Graceful degradation when cache or CDN is unreachable; circuit breakers and rate limits; event sourcing for critical transitions (round start/end, score changes) to allow recovery.

9. **Security**:

   - Transport security: Enforce TLS end-to-end; HSTS and secure cookies.
   - Identity & access: Token-based auth (JWT or opaque tokens) with short lifetimes and refresh flow; role-based controls (player, host, admin). Validate inputs server-side; limit guess rate to prevent abuse.

- Data protection: Store minimal PII; hash sensitive identifiers where appropriate; parameterized queries to prevent injection; unique constraints enforce integrity.