Draw It or Lose It

CS 230 Project Software Design Template

Version 1.0

Table of Contents

Document Revision History

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 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)

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.

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

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.

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.

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.