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	11/16/2025	Angel I. Rivera Perez	Initial draft with all required 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

Creative Technology Solutions (CTS) is expanding The Gaming Room's Android title, Draw It or Lose It, to a web-based, multi-platform architecture. The goal is to deliver simultaneous gameplay to many users and teams across desktop and mobile devices while preserving low latency, unique entity identities, and consistent state.

I recommend a cloud-hosted, service-oriented web application that centralized game logic in a Game Service layer. This design enforces unique identifiers for games, teams, and players; prevents duplicate instances via the Singleton pattern; and safely traverses collections via the Iterator pattern. The solution supports horizontal scaling, secure session management, and straightforward client delivery through standard browsers (Windows, macOS, Linux) and mobile devices.

Key implications: the application must run in a distributed environment, handle current access, and maintain authoritative state on the server while keeping clients thin. These constraints drive choices in operating systems, storage, memory management, networking, and security discussed in later sections.

Requirements

Business Requirements:

- The game must run on a modern browser across Windows, macOS, Linux, and mobile devices (iOS/Android).
- Support (Real-time play) simultaneous players and teams with responsive rounds and scoring.
- Service resilient to traffic spikes; target 99.9%+ availability during events.
- Add capacity by adding instances (horizontal scaling).
- Low friction access: No native installs required; users join via links/auth flow.
- Prefer commodity cloud services and open standards to keep operation expenses predictable.

Technical requirements:

- Web-based architecture: Thin clients (browser) + authoritative server holding all game state.
- All entities (Game, Team, Player) must have globally unique IDs; no duplicates by name within a game.
- Concurrency control by server preventing race conditions when creating/joining games/teams.
- Use WebSockets or long-polling for live updates; REST for setup flows.
- Secure session management (tokens/cookies), rate limiting on sensitive endpoints.
- Durable storage of games, teams, players and results; support recovery after restart.
- Patterns:
  Singleton for the GameService (single orchestrator instance per process).
  Iterator to traverse operations safely(e.g., find name/ID).
- Server-side validation for inputs (names, team size, round length).
- Structured logs and basic metrics (requests/sec, latency, errors).
- Portability: Codebase and tooling must build/run on Linux CI and deploy to cloud hosts.
- Security baseline: HTTPS everywhere, input sanitizations, least-privilege access to data stores.
- Unit tests for core logic (entity creation, uniqueness, scoring); smoke test for API.

Design Constraints

Web-based Delivery:
The game must run in browsers on Windows, macOS, Linux, and mobile devices. This requires using standard web technologies (HTML5, JavaScript) and keeping the client lightweight.

Distributed Client-Server Model:
The server manages all game logic while clients send and receive updates in real time. This setup supports multiple users playing simultaneously.

Concurrency and Uniqueness:
Each game, team, and player must have a unique identifier. The Singleton pattern in GameService ensures only one instance manages all game data safely.

Scalability:
The system should support many active players by allowing additional server instances. Stateless design and efficient memory use help maintain performance.

Persistence:
Game data must be stored securely so progress is not lost during restarts or crashes.

Security:
All communication uses HTTPS, and the server validates all input to protect against unauthorized access and data corruption.

Network Reliability:
Because players connect over various networks, the game must handle slow or unstable connections gracefully.

Usability:
The interface should be responsive and accessible on different screen size and devices.

Cost and Tools:
Development must use open-source tools and remain affordable to host and maintain.

System Architecture View

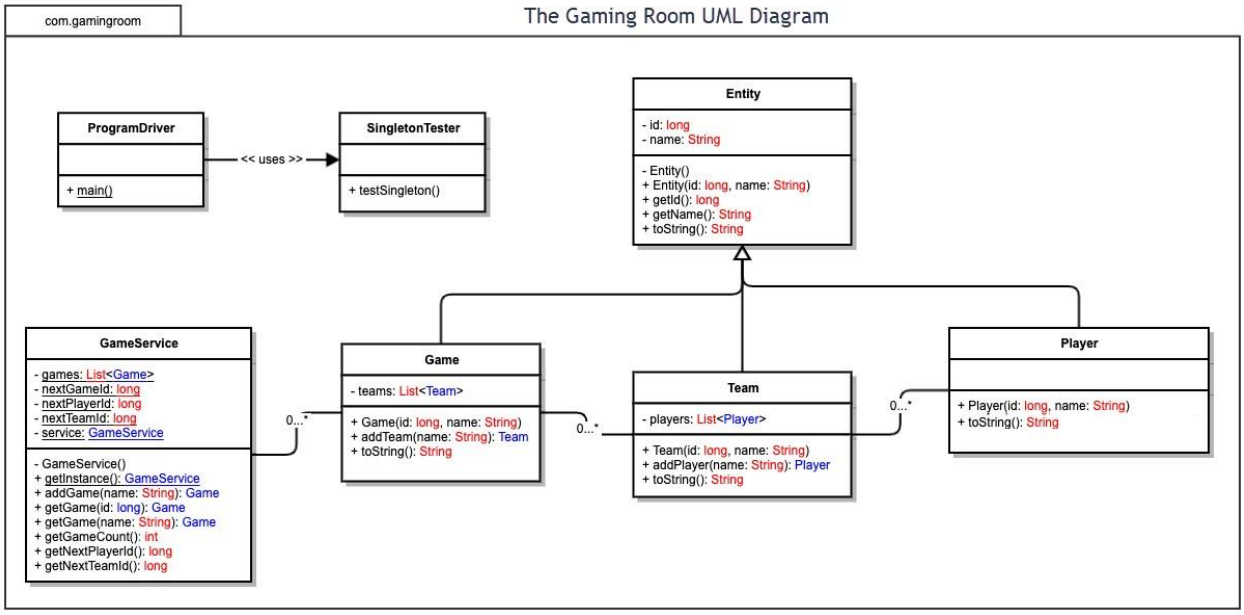
Please note: There is nothing required here for these projects, but this section serves as a reminder that describing the system and subsystem architecture present in the application, including physical components or tiers, may be required for other projects. A logical topology of the communication and storage aspects is also necessary to understand the overall architecture and should be provided.

Domain Model

The domain model for the game application is built around several core classes that represent the problem space: GameService, Game, Team, Player, ProgramDrive, and SingletonTester.

GameService is the central class that manages all Game objects. It is implemented as a Singleton, ensuring that only one instance controls the list of active games and their related data. Each game has a unique identifier and name, and is associated with one or more Team objects. Each Team is associated with one or more Player objects. This creates a clear hierarchy: one GameService manages many games, each game manages many teams, and each team manages many players.

The model uses core object-oriented principles. Encapsulation is applied by keeping fields such as id, name, and games private and exposing them through public getter methods. Abstraction is present in the way classes represent real-world entities (games, teams, players) without exposing internal implementation details. The Iterator pattern is used inside GameService to safely traverse collection of games when searching by ID or name. ProgramDriver acts as the entry point that initializes game data, while SingletonTester demonstrates that all parts of the application share the same GameService instance.



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	While stable and UNIX-based, it lacks strong industry support, has higher hardware cost, and is not optimized for large-scale hosting.	It offers high performance, scalability, strong security, and low cost. Its open-source ecosystem and dominant presence in cloud environments make it ideal for a distributed, web-based application with many users.	Windows server provides good tooling and compatibility with enterprise systems, but licensing costs are high, and performance is generally less efficient under heavy load than Linux.	Mobile devices are not designed to host server applications. Their operating systems restrict background services, network access, and resource use.
Client Side	Mac clients access the game through a browser with minimal compatibility issues. Developers mainly need to ensure smooth performance on Safari and maintain responsive UI layout across screen sizes.	Linux clients require testing across different distributions, but modern browsers behave consistently. As long as the game is browser-based, support is straightforward and low-cost.	Windows clients are the most common. Ensuring compatibility across major browsers (Edge, Chrome, Firefox) and varied hardware configurations is important, but overall client support is easy to maintain.	Mobile clients require touch-friendly design, fast loading, and resolution flexibility. iOS and Android browsers differ slightly, so testing must cover both platforms.
Development Tools	macOS provides strong development tools like IntelliJ, Eclipse, VS Code, Xcode, and UNIX command-line utilities. It works well for building and testing Java and web applications.	Linux aligns closely with most production environments. Tools such as Docker, Git, Java SDK, and JetBrains IDEs run efficiently, making it an excellent choice for server-side development and testing.	Windows supports a wide range of IDEs and tools, including Visual Studio, IntelliJ, Eclipse, and WSL for Linux-like development. It is flexible but may require extra configuration for Linux-targeted developments.	Mobile devices are only useful for testing. Development occurs on desktop platforms using tools like Android Studio or Xcode. Smartphones and tablets lack resources needed for full development workflows.

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**: Linux is the best platform for hosting the game. It provides strong performance, low cost, high reliability, and broad support across cloud environments, this makes it ideal for a scalable, web-based application like Draw It or Lose It.

2. **Operating Systems Architectures**: A simple client-server architecture Is recommended; splits the work between a single powerful server and many lightweight clients. The Linux server handles all game logic, storage, and authentication, while clients on Windows, macOS, Linux, and mobile devices connect through standard web browsers. This architecture keeps the system lightweight, maintainable, and easy to expand.

3. **Storage Management**: Use cloud-based storage with redundant backups, such as AWS S3 or equivalent. This ensures durability, fast data access, and easy scaling as more players join. It also reduces the risk of data loss.

4. **Memory Management**: Linux efficiently handles multiple simultaneous requests through virtual memory, caching, and process isolation. These features ensure the game remains responsive even as traffic increases. The platform automatically frees unused memory and optimizes available resources.

5. **Distributed Systems and Networks**: To support multiple platforms, the game should run on a distributed system using RESTful API calls over HTTPS. The server processes all game events while clients only send and receive data. This separation ensures consistent gameplay across devices and reduces dependency on individual client hardware or network conditions.

6. **Security**: Security should focus on protecting user sessions and data in transit. Use HTTPS, secure authentication, regular updates, and server-side validation. Linux's strong built-in security model and limited attack surface help prevent unauthorized access and keep user information safe.