Draw It or Lose It

CS 230 Project Software Design

Version 2.0

Table of Contents

Document Revision History

Version	Date	Author	Comments
1.0	01/20/2025	Edwin Martinez	Initial draft.
2.0	02/3/2025	Edwin Martinez	Client requested new requirements. Changes made to evaluation section.
3.0	02/17/2025	Edwin Martinez	Client requested update to recommendation.

Executive Summary

The Gaming Room, a game development company, aims to expand its current Android game, Draw It or Lose It, into a web-based, multi-platform application. This expansion seeks to enhance the accessibility and scalability of the game, allowing access across a broader audience. The game requires robust infrastructure to handle multiple teams, unique naming conventions, and efficient game state management.

To address these needs, this design leverages object-oriented programming principles, a distributed architecture, and a centralized game management system. The proposed solution ensures the game supports concurrent teams, maintains unique identifiers for all entities, and enforces the singleton pattern requirement.

Requirements

1. The system must allow one or more teams to participate in a game.
2. Teams must consist of multiple players.
3. Game and team names must be unique to avoid conflicts.
4. The application must enforce a single active instance of the game in memory.
5. The system must be accessible across multiple platforms.

Design Constraints

1. **Concurrency and Scalability:**
    a. The application must support multiple teams and players simultaneously.
    b. This requires a distributed architecture and efficient resource management to handle high traffic.
2. **Unique Identifiers:**
    a. Games, teams, and players must have unique names and IDs.
    b. This necessitates mechanisms for checking and enforcing uniqueness, increasing database complexity.
3. **Single Instance Enforcement:**
    a. Only one active game instance is allowed at a time.
    b. Unique identifiers will ensure that even in a distributed environment, there is no overlap or duplication of game instances.
4. **Cross-Platform Compatibility:**
    a. The application must operate on various platforms, requiring development tools and frameworks that support portability.

System Architecture View

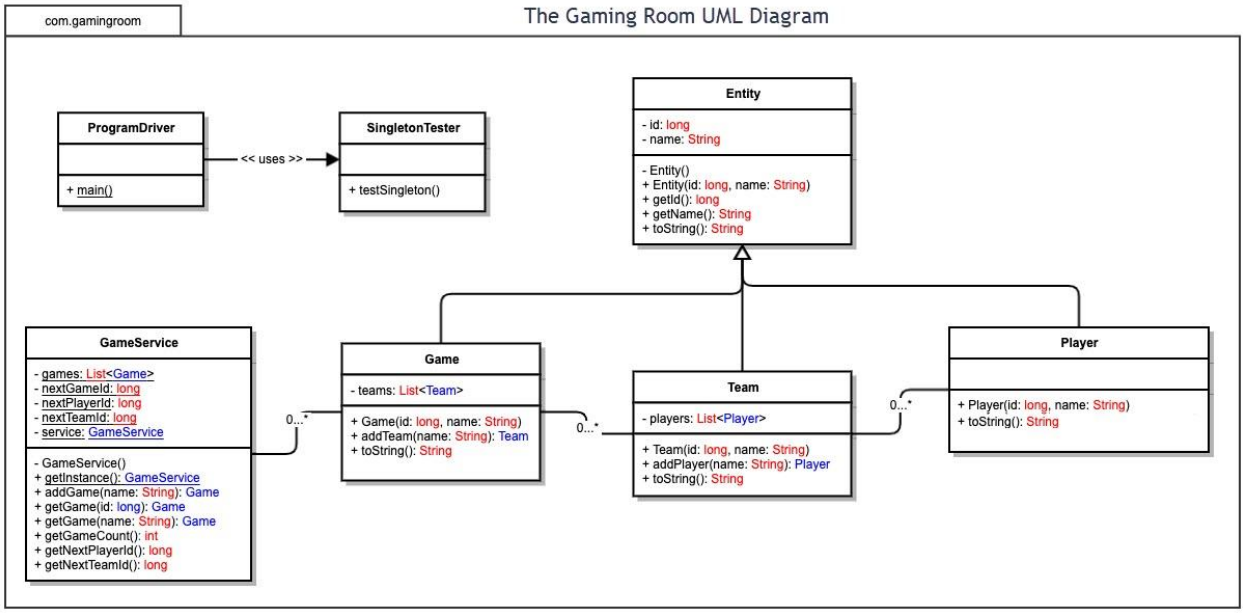
Please note: There is nothing required here for these projects, but this section serves as a reminder that describing the system and subsystem architecture present in the application, including physical components or tiers, may be required for other projects. A logical topology of the communication and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

Domain Model

1. **Entity Class:**
    a. Acts as a base class for shared attributes and behaviors, such as id and name. These attributes ensure uniqueness and provide consistency across all derived classes which will serve the requirement of enforce single instances of unique entity(game, player, team).
    b. The Entity class demonstrates the principle of inheritance, reducing code duplication by allowing other classes to inherit common attributes and methods like getId and getName.
2. **GameService Class:**
    a. Responsible for managing the lifecycle of games, including adding games and retrieving them by ID or name.
    b. It maintains a list of games (games: List<Game>) and unique identifiers for games, teams, and players (nextGameId, nextTeamId, nextPlayerId).
    c. By centralizing these responsibilities, this class ensures that only one active game instance exists at a time.
3. **Game Class:**
    a. Represents an individual game and contains a list of associated teams (teams: List<Team>).
    b. Provides methods to add teams (addTeam) and retrieve game details. This encapsulates the game's data and ensures the teams belong exclusively to their respective games.
    c. Inherits from the Entity class, benefiting from shared behaviors while maintaining game-specific functionality.
4. **Team Class:**
    a. Represents a team within a game and holds a list of players (players: List<Player>).
    b. Includes methods to add players (addPlayer) and access team-specific information, ensuring that players are properly managed within their teams.
    c. Inherits from the Entity class, benefiting from shared behaviors while maintaining team-specific functionality.
5. **Player Class:**
    a. Represents individual players who belong to a team. Each player is uniquely identified by an id and a name.
    b. Inherits from the Entity class, benefiting from shared behaviors while maintaining player-specific functionality.

Object-Orientated Principles

- **Inheritance:** The Entity class serves as a parent class, streamlining common functionality and attributes for its derived classes (Game, Team, Player).
- **Abstraction:** The Entity class provides a high-level abstraction for shared attributes and methods.
- **Encapsulation:** Data is protected within each class, and only relevant methods are exposed to manipulate or retrieve information.
- **Polymorphism:** Shared methods like toString are overridden in derived classes to provide specific implementations tailored to each class.



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	**Scalability:** Poor for large-scale hosting. macOS is not designed as a web server OS and lacks enterprise-grade hosting solutions. **Licensing Costs:** High as it requires Apple hardware, increasing infrastructure costs. macOS Server is discontinued. **Security:** Secure OS, but lacks enterprise-level security features tailored for web hosting. **Hosting Support:** Limited hosting capabilities. Not widely used for server deployment.	**Scalability:** Highly scalable, widely used for enterprise and cloud hosting. Easily supports large-scale applications with load balancing and clustering. **Licensing Costs:** Free and open-source, reducing overall hosting expenses. No licensing fees unless opting for enterprise support. **Security:** Strong built-in security with frequent updates, role-based access control, SELinux, and AppArmor. Most secure option for web hosting. **Hosting Support:** Preferred choice for most cloud services (AWS, Digital Ocean, Google Cloud, Azure). Offers Apache, Nginx, and other robust web servers.	**Scalability:** Scales well with enterprise-level infrastructure, particularly in Microsoft ecosystems. Supports load balancing and clustering but often requires more system resources than Linux. **Licensing Costs:** High as it requires a Windows Server license, which increases operational costs. Additional fees for enterprise features. **Security:** Strong security features but historically more vulnerable to malware and exploits. Regular updates and patches are required to maintain security. **Hosting Support:** Best for enterprises using Microsoft technologies, but not as widely used for web hosting as Linux.	**Not applicable for hosting**: mobile devices act as clients, not servers. **Security:** Mobile OSs focus on app sandboxing rather than server-level security. **Performance:** Limited processing power and storage make them unsuitable for server hosting.

Client Side	**Compatibility:** Excellent support for web applications via Safari, Chrome, and Firefox. Apple tends to optimize macOS for web browsing. **Performance:** Strong performance with macOS hardware, but Safari has unique rendering behaviors that may require testing for responsiveness. **Development Challenges:** Developers must test for Safari quirks and ensure the UI is responsive on Retina displays. **Development Cost:** Requires a Mac for iOS testing and development.	**Compatibility:** Fully supports web-based applications through modern browsers like Chrome and Firefox. Some browsers like Safari may have limited Linux support. **Performance:** Generally good, but browser-based applications may depend on GPU acceleration, which varies by distribution. **Development Challenges:** Users may need additional dependencies like WebGL support. **Development Cost:** No licensing fees. Development tools like VS Code and Chrome DevTools are free.	**Compatibility:** Fully supports modern web applications via Chrome, Edge, and Firefox. Internet Explorer is deprecated but may still be used by some users. **Performance:** Generally strong performance. Some older Windows machines may struggle with WebGL-heavy content. **Development Challenges:** Need to ensure compatibility across multiple versions of Windows. Edge and Chrome use Chromium but may have small differences. **Development Cost:** No licensing costs for web apps, but Windows development tools like Visual Studio may have paid tiers.	**Compatibility:** Requires a responsive design that adapts to different screen sizes and orientations. Both iOS and Android support web-based apps but have different browser rendering engines. Safari on iOS, Chrome on Android. **Performance:** Mobile devices have less processing power than desktops, so performance optimization is necessary. **Development Challenges:** Ensuring touch-friendly UI, handling offline capabilities, and optimizing network requests for mobile speeds. **Development Cost:** Requires cross-platform mobile testing. iOS development requires Apple hardware.

Development Tools	**Languages & Frameworks:** Supports the same web technologies as Linux, plus native iOS development with Swift and Objective-C. **Development Tools:** Xcode, VS Code, WebStorm, and Chrome DevTools. **Licensing Costs:** Xcode is free, but Apple requires a paid developer license to publish iOS apps. **Impact on Development Team:** iOS testing requires Mac hardware, meaning some developers may need Macs.	**Languages & Frameworks:** Supports full-stack development with Node.js, Python (Django/Flask), PHP, and JavaScript (React, Angular, Vue.js). **Development Tools:** Common tools include VS Code, Vim, Eclipse, and JetBrains WebStorm. **Licensing Costs:** Most tools are free and open-source. **Impact on Development Team:** No additional team needed, Linux supports all major development frameworks.	**Languages & Frameworks:** Supports all major web development stacks, including ASP.NET for Windows-specific development. **Development Tools:** Visual Studio, VS Code, WebStorm, and Chrome DevTools. **Licensing Costs:** Visual Studio Enterprise requires a license, but other tools like VS Code are free. **Impact on Development Team:** No additional team needed, Windows supports all necessary development frameworks.	**Languages & Frameworks:** Mobile web apps rely on JavaScript frameworks (React Native, Flutter) for cross-platform support. **Development Tools:** Android Studio, Xcode, React Native CLI for hybrid development. **Licensing Costs:** Apple requires a fee for iOS publishing, but Android development is free. **Impact on Development Team:** Mobile testing requires both Android and iOS expertise, possibly requiring a dedicated mobile team.

Recommendations

Operating Platform:

Linux is best for server hosting due to its reliability, scalability, and cost-effectiveness. It is widely used in cloud and enterprise environments, making it ideal for supporting The Gaming Room's expansion of Draw It or Lose It. Additionally, Linux is open-source, eliminating costly licensing fees while offering excellent support for high-traffic web applications. Linux also offers different distribution like Ubuntu, Debian, RHEL and more.

Operating Systems Architectures:

For Linux, I recommend utilizing Nginx as a web server for handling HTTP requests, reverse proxying, load balancing, and serving static assets efficiently. For containerization, Docker or Kubernetes can be used to ensure scalability and easy deployment across multiple server instances. Linux should be structured as a microservices architecture to enable modular development, reducing dependency issues and improving maintainability. This architecture ensures optimized performance for a distributed web application environment, allowing thousands of concurrent users to access the game seamlessly.

Storage Management:

For storage, I recommended a relational database management system (RDBMS), specifically MySQL or PostgreSQL. Because they provide data integrity for unique identifiers like games, teams and players. They also provide scalability through replication and clustering as well as optimized querying for real-time game state updates. Additionally, Redis, an in-memory caching system, can be used to store frequently accessed data to improve response time.

Memory Management:

To optimize memory usage, I recommend:

1. Garbage Collection & Process Management: The server will manage memory allocation dynamically, ensuring unused objects are efficiently cleared.
2. Caching Mechanisms: Redis will handle frequent database queries, reducing the load on the primary database and enhancing game performance.
3. Load Balancing & Auto-Scaling: By using cloud-based infrastructure, server resources can dynamically scale based on real-time demand. Cloud services provided by AWS or Digital Ocean allow you to set this up.

Distributed Systems and Networks:

Since Draw It or Lose It must support multiple platforms, I recommend the system to use a distributed architecture to facilitate seamless communication between clients and the backend. These components are:

- WebSockets: Real-time game updates will be achieved through WebSockets, reducing latency and ensuring a smooth multiplayer experience.
- Content Delivery Network (CDN): A CDN will be used to cache and distribute static assets across multiple geographic locations, improving load times.

- Failover and Redundancy: Multiple redundant servers will be deployed across data centers to ensure continued service availability during failures.
- Cloud Infrastructure: Cloud-based infrastructure will provide auto-scaling to adjust server resources based on user traffic.

These distributed system techniques will enhance connectivity and fault tolerance, ensuring players can connect reliably even if a server experiences issues.

Security:

Security measures I recommend are:

- Encryption:
    - HTTPS with TLS 1.2/1.3 to encrypt all network communications, this requires using a service like Let's Encrypt to gain a certificate.
    - AES-256 encryption middleware algorithm for storing sensitive data.
- Authentication & Authorization:
    - Implement OAuth 2.0 for secure login across platforms.
    - Role-Based Access Control (RBAC) to restrict access based on user roles.
- Secure API Design:
    - API requests will be validated using JWT (JSON Web Tokens) to prevent unauthorized access.
    - Rate limiting will be enforced to mitigate DDoS attacks.
- Regular Security Audits:
    - Hire cybersecurity firm to conduct penetration testing and automated security scans to identify vulnerabilities.
    - Implement intrusion detection and prevention systems to monitor and prevent threats.
- Data Backups:
    - Perform regular automated backups with redundant storage solutions, ensuring rapid recovery in case of system failure or data corruption.
- Server Maintenance:
    - Provide regular updates to the server to ensure it is using the latest version to avoid exploits.
    - Scan images and provide updates to packages and dependencies within the containerized environment.