Draw It or Lose It
CS **230 Project Software Design Template**
Version 2.0

Table of Contents

Document Revision History

Version	Date	Author	Comments
1.0	10/2/2025	Gary Travis	Updated development requirements

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's **Draw It or Lose It** has already proven successful on Android, but its current single-platform limitation restricts audience reach and long-term growth. To meet the client's goals, this project proposes expanding the game into a **web-based distributed application** that runs seamlessly across multiple operating systems (Windows, macOS, Linux) and mobile devices (iOS and Android).
By transitioning to a distributed web architecture, **Draw It or Lose It** will benefit from scalability, cross-platform compatibility, and easier maintenance. The system will support multiple teams and players simultaneously while enforcing unique identifiers to ensure data integrity. A centralized game service—enforced through the singleton design pattern—will guarantee that only one instance of the service runs in memory at a time, preventing conflicts and improving reliability.
From a business perspective, this design lowers long-term costs by leveraging open-source technologies like Linux for hosting, responsive HTML5 for the client interface, and widely supported programming languages and frameworks. It also positions the game for **future growth**, such as adding advanced authentication, analytics, or cloud-based features without rewriting the core system.
Ultimately, this design balances **technical performance, security, and user experience** while providing The Gaming Room with a scalable solution that maximizes reach and ensures **Draw It or Lose It** remains competitive across today's most widely used platforms.

Requirements

To successfully expand **Draw It or Lose It** into a distributed, web-based application, the following **business and technical requirements** must be met:

- **Team and Player Structure**: Each game can support one or more teams, and each team must be able to include multiple players. This ensures flexibility and scalability for both small and large groups of participants.
- **Unique Identifiers**: Every game, team, and player must be assigned a globally unique identifier. This prevents duplication and ensures data integrity across the distributed system.
- **Name Uniqueness**: Game and team names must be unique to avoid conflicts and confusion during gameplay. Iteration and validation processes will be required to enforce this rule.
- **Singleton Game Service**: Only one instance of the game service can run in memory at any given time. This is critical for maintaining consistency and preventing conflicting game states.
- **Cross-Platform Compatibility**: The application must function consistently across major desktop operating systems (Windows, macOS, Linux) and mobile devices (iOS, Android) through a modern, responsive HTML5 interface.
- **Distributed Environment**: The game must operate in a web-based distributed environment to handle potentially thousands of simultaneous users, with load balancing and redundancy for reliability.
- **Security**: Data exchanged between clients and the server must be encrypted and safeguarded against unauthorized access. This includes future-proofing the design for user authentication, secure storage, and compliance with industry security standards.

By clearly defining these requirements, The Gaming Room will have a solid foundation to guide design decisions, prioritize features, and ensure the successful expansion of **Draw It or Lose It** into a multi-platform environment.

Design Constraints

The development of **Draw It or Lose It** as a web-based, distributed application must address several critical constraints that will shape design and implementation:

- **Singleton Enforcement**: Only one instance of the game service may run in memory at a time. This is essential to prevent conflicting game states. The singleton design pattern will be applied to the GameService class to enforce this rule.
- **Name Uniqueness**: All games, teams, and players must have unique names. This requires implementing validation logic and iterator-based checks before creation, ensuring that no duplicates can be introduced into the system.
- **Scalability**: The system must handle potentially thousands of simultaneous players without performance degradation. This requires a distributed architecture, efficient resource management, and the ability to scale horizontally using cloud services or containerized environments (e.g., Docker/Kubernetes).
- **Cross-Platform Consistency**: Since the game must run across Windows, macOS, Linux, and mobile devices, the design must prioritize platform-independent technologies such as Java for backend logic and responsive HTML5 for the frontend. This ensures a consistent user experience regardless of device or operating system.
- **Security Requirements**: As the game expands, protecting player data becomes a high priority. All communication must use TLS encryption, and the architecture should allow for future integration of secure authentication (e.g., OAuth) and database-level protections.
- **Performance and Responsiveness**: The game's timing mechanics—such as one-minute rounds and 30-second drawing completions—require precise server-side scheduling and low-latency communication. Network outages or lag must be mitigated with retry logic and fault tolerance.
- **Cost Considerations**: To remain cost-effective, licensing fees for server platforms and development tools must be minimized. This favors open-source solutions such as Linux for hosting and widely available IDEs like Eclipse, IntelliJ IDEA Community, or VS Code.

These constraints ensure that the system design for **Draw It or Lose It** remains efficient, secure, scalable, and maintainable, while balancing technical limitations with business objectives.

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 UML class diagram for **Draw It or Lose It** illustrates how the core classes interact to support the game's requirements. At the foundation is the **Entity** class, which defines shared attributes such as id and name. By extending from this base class, other objects in the system inherit these core properties, ensuring consistency and reducing redundancy.

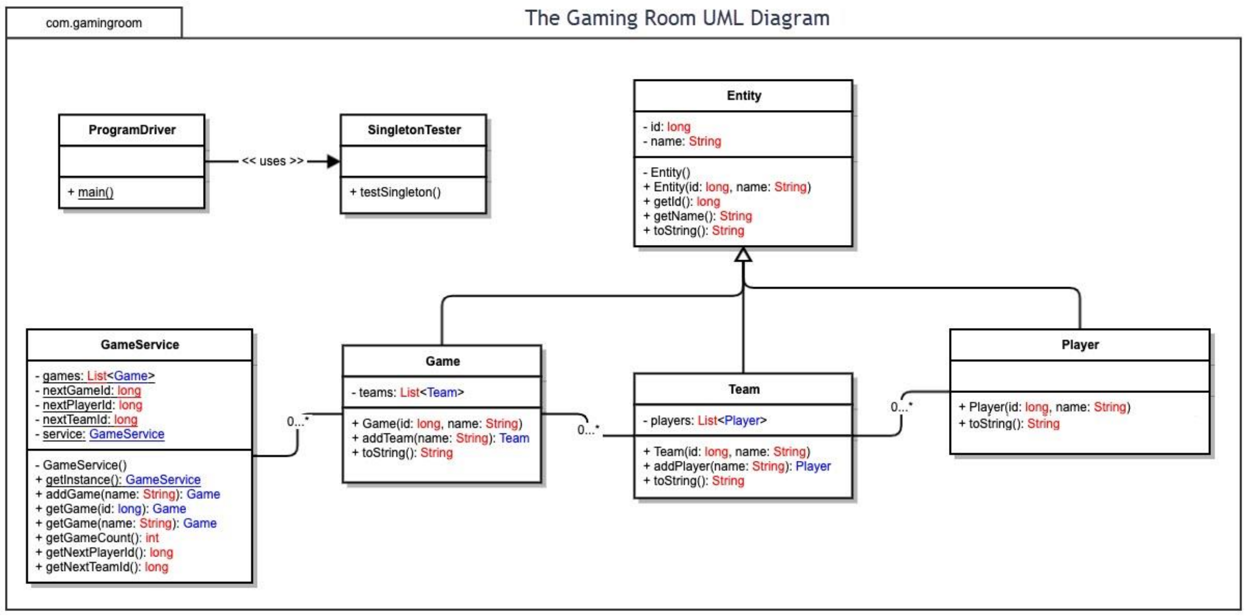
- **GameService**: Implemented as a singleton, this class manages all active games and enforces the constraint that only one service instance exists in memory at a time. It acts as the central controller, coordinating game creation and lifecycle management.
- **Game**: Inherits from Entity and contains a collection of Team objects. Each Game enforces unique team names and serves as the container for the gameplay session.

- **Team**: Also extending from Entity, a Team manages a list of Player objects. Teams are linked to specific games and must maintain unique names within that game context.
- **Player**: A subclass of Entity representing an individual participant. Players belong to a specific team and are uniquely identified by both their name and system-generated identifier.
- **ProgramDriver**: Contains the main method to run the program and initializes the system. This provides the entry point for execution and testing.
- **SingletonTester**: A specialized class designed to validate the singleton implementation of the GameService, ensuring system stability by preventing multiple concurrent instances.

This design showcases several key **object-oriented programming principles**:

- **Inheritance**: Common attributes (id, name) are defined once in the Entity class and reused across Game, Team, and Player.
- **Encapsulation**: Class variables are declared private and accessed only through getters and setters, protecting data integrity.
- **Polymorphism**: Each class can override methods like toString(), allowing customized representations for debugging and logging.
- **Design Patterns**: The singleton pattern ensures centralized control of the game service, while iterator-based validation helps prevent duplicate names.

Together, these relationships create a robust, extensible domain model that supports scalability, enforces uniqueness, and maintains clear organization of games, teams, and players in the **Draw It or Lose It** environment.



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	macOS Server can technically host web apps but is rarely used in enterprise environments. It has limited scalability and fewer modern deployment options. Licensing is inexpensive compared to Windows, but support for large-scale hosting is weaker. Best suited for development/testing, not production.	Linux is the industry standard for hosting. It supports Apache, Nginx, Docker, and Kubernetes at low/no licensing cost. It scales easily, has strong community support, and is the most cost-effective option for cloud deployment. Main weakness: requires skilled administrators.	Windows Server offers robust support for .NET and enterprise systems. It integrates well with Microsoft services (Azure, Active Directory). Licensing and CAL costs are higher, and overhead is greater, but enterprise support is strong. Scales well but at higher cost.	Mobile devices are not practical as hosts for web apps. They lack processing power, storage, and scalability. Instead, they act as clients. Hosting a server on mobile is not feasible for this project.
Client Side	Modern macOS browsers (Safari, Chrome, Firefox) fully support responsive HTML5 apps. Development requires ensuring cross-browser compatibility. Costs are moderate, as Mac clients need minimal additional adaptation. Time impact is low once responsive design is in place.	Linux desktop clients rely primarily on Chrome/Firefox. Support for web standards is strong, but testing may take more time due to many Linux distributions. Costs are minimal, and the expertise needed overlaps with general web development.	Windows is the largest desktop client base. Edge, Chrome, and Firefox support HTML5 fully. Requires testing across multiple versions. Cost is mainly in QA and browser compatibility. Expertise overlaps with Mac/Linux, so development time is manageable.	Mobile support requires ensuring responsive design across iOS Safari and Android Chrome. Development considerations include touch input, varied screen sizes, and performance optimization. Extra QA and expertise are needed to test multiple devices. This adds more cost and time than desktop clients.

Development Tools	Languages: Java, Python, JavaScript, Swift (for local macOS tools). IDEs: Xcode, IntelliJ IDEA, VS Code. Most have free/community editions. Licensing costs are low, but the team needs macOS hardware to use Xcode.	Languages: Java, Python, JavaScript, PHP. IDEs: Eclipse, IntelliJ IDEA, VS Code. Tools are open-source or low-cost. Linux is developer-friendly and integrates easily with CI/CD pipelines. Expertise is common in the open-source community.	Languages: Java, C#, JavaScript, Python. IDEs: Visual Studio (paid), IntelliJ IDEA, VS Code. Visual Studio licensing may add costs. Windows provides strong dev support for .NET but can be more expensive to equip and maintain.	Languages: Java (Android), Swift (iOS), Kotlin, JavaScript (with responsive frameworks). Tools: Android Studio, Xcode, cross-platform frameworks like React Native or Flutter. Expertise requires mobile-specific skills. Licensing is minimal, but device and OS fragmentation increase QA costs.

__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**:

   For hosting **Draw It or Lose It**, **Linux** is the recommended operating platform. It provides the strongest balance of **scalability, cost-effectiveness, and reliability** compared to macOS or Windows Server. Linux is the industry standard for modern web hosting and offers mature support for web servers such as Apache and Nginx, as well as containerization technologies like Docker and Kubernetes.

   Unlike Windows Server, which incurs higher licensing and client access costs, Linux distributions are open-source and free, making them highly attractive for long-term operational expenses. macOS, while functional for development and testing, is not widely used in production environments and lacks the enterprise-level hosting ecosystem that Linux provides.

   By deploying the server-side application on Linux, The Gaming Room ensures **maximum flexibility for scaling**, extensive community and vendor support, and seamless integration with leading cloud platforms (AWS, Azure, GCP), all of which position **Draw It or Lose It** for sustainable growth and stability.

2. **Operating Systems Architectures**:

   The recommended architecture for **Draw It or Lose It** follows a **multi-tier, distributed systems design**. The server tier will run on a Linux-based environment to host the backend application logic, web services, and database. This architecture separates concerns across layers, improving scalability, maintainability, and security.

- **Server Tier**: Hosts the application backend, APIs, and game logic. This tier enforces singleton rules for the game service and manages session control, team/player interactions, and timing mechanics.
- **Database Tier**: Maintains persistent data, including unique identifiers for games, teams, and players, as well as user session information. A relational database like PostgreSQL or MySQL ensures strong consistency and referential integrity.
- **Client Tier**: Provides the user interface via modern browsers on Windows, macOS, Linux, and mobile devices. Clients communicate with the server using HTTPS, ensuring secure and efficient data transfer.

   This layered approach provides **loose coupling** between components, making it easier to scale specific tiers as demand grows. For example, additional web servers can be deployed behind a load balancer to handle peak traffic, while the database tier can be scaled vertically or clustered for redundancy.

   By adopting this distributed systems architecture, **Draw It or Lose It** will be able to accommodate thousands of concurrent users while maintaining performance, fault tolerance, and a consistent user experience across platforms.

3. **Storage Management**:

   To support scalability and maintain data integrity, **Draw It or Lose It** should utilize a **relational database management system (RDBMS)** such as **PostgreSQL** or **MySQL**. Both are highly reliable, open-source solutions that provide strong transaction support, referential integrity, and proven performance in distributed environments.

- The storage system will handle the following critical data:
- **Unique Identifiers**: Games, teams, and players must each have globally unique IDs to prevent duplication.
- **User and Session Data**: Information about players, teams, and active sessions must be stored consistently to allow real-time gameplay.
- **Game State and History**: Round results, timing data, and past games should be recorded for auditing, analytics, and possible expansion into leaderboards or player statistics.

By selecting a relational model, the system ensures **consistency and accuracy of data**, which is critical given the requirement for unique names and identifiers. Scalability can be achieved through techniques such as database replication, sharding, or managed cloud services (e.g., Amazon RDS, Google Cloud SQL), which offer automatic scaling and backups.

This approach provides The Gaming Room with a **cost-effective, secure, and flexible** data storage solution that is both future-proof and easy to integrate with the Linux-based hosting environment.

4. **Memory Management**:

The **Draw It or Lose It** application will rely on the **Java Virtual Machine (JVM)** for memory management on the Linux server platform. The JVM provides **automatic garbage collection**, efficient heap allocation, and robust error handling, which together ensure that memory resources are used effectively during gameplay.

Because the game service must enforce strict timing for one-minute rounds and simultaneous player actions, memory management is critical to avoid latency or interruptions. The singleton implementation of the GameService reduces memory overhead by ensuring only one instance of the service runs at a time, while collections of games, teams, and players are dynamically allocated and freed as sessions start and end. Additional strategies include:

- **Heap Size Tuning**: Configuring JVM heap settings to optimize performance under heavy load.
- **Object Reuse**: Minimizing unnecessary object creation to reduce garbage collection cycles.
- **Caching**: Leveraging in-memory caching (e.g., Redis or JVM-based caching) for frequently accessed data like player names or game states.

Together, these practices ensure that **Draw It or Lose It** can maintain responsive performance even with thousands of concurrent players, while also providing the flexibility to scale memory usage as the application grows.

5. **Distributed Systems and Networks**:

The distributed nature of **Draw It or Lose It** requires a reliable and scalable network design to support thousands of concurrent players across desktop and mobile platforms. The architecture will follow a **client-server model**, with clients communicating securely with the centralized Linux-based server through HTTPS.

Key elements include:

- **Load Balancing**: Incoming traffic will be distributed across multiple application servers using a load balancer. This ensures that no single server becomes a bottleneck and improves both performance and reliability.

- **Fault Tolerance and Redundancy**: Redundant servers and failover mechanisms will be in place so that the game remains available even during outages or maintenance. This includes replication at the database tier for high availability.
- **Scalable Communication**: Real-time gameplay requires efficient communication channels. RESTful APIs will handle most client requests, while technologies like WebSockets or Server-Sent Events (SSE) can provide low-latency updates for game timers and round status.
- **Content Delivery Networks (CDNs)**: Static assets such as images in the drawing library can be distributed via CDNs to reduce latency and ensure fast loading times globally.
- **Monitoring and Alerts**: Network traffic, latency, and error rates will be continuously monitored with tools such as Prometheus, Grafana, or cloud-native monitoring services.

By adopting these distributed systems and networking strategies, **Draw It or Lose It** will achieve the scalability, resilience, and low-latency performance needed to deliver a seamless gaming experience to users on any platform.

6. **Security**:

Security is a critical requirement for **Draw It or Lose It**, ensuring that player data and game sessions remain protected across all platforms. The distributed system must defend against unauthorized access, data breaches, and common web vulnerabilities while maintaining smooth gameplay.

Key security measures include:

- **Encryption**: All communication between clients and servers will use SSL/TLS protocols to secure data in transit. Sensitive information, such as player credentials, will be encrypted both at rest and in motion.
- **Authentication and Authorization**: A secure login system should be implemented using modern standards such as OAuth 2.0 or OpenID Connect, allowing for role-based access control (e.g., player, administrator).
- **Data Protection**: The database layer will enforce strong security practices including input validation, SQL injection prevention, and least-privilege access control for users and applications.
- **Network Security**: Firewalls, intrusion detection systems, and API gateways will help filter malicious requests and monitor suspicious traffic patterns.
- **Application Security**: The backend application will be developed with secure coding practices, regularly updated to patch vulnerabilities, and protected by automated vulnerability scans and penetration testing.
- **Scalability of Security**: As the game grows, cloud providers' native security tools (e.g., AWS IAM, Azure Security Center, or GCP Identity & Access Management) can be integrated to maintain compliance and support future expansion.

By implementing these protections, **Draw It or Lose It** will safeguard its users while meeting industry security standards, building trust with players, and ensuring the platform remains resilient against evolving threats.