THE GAMING ROOM

Draw It or Lose It

CS 230 Project Software Design Template

Version 3.0

Table of Contents

Document Revision History

Version	Date	Author	Comments
1.0	07/19/2025	Bradley Saucier	Initial design
2.0	08/03/2025	Bradley Saucier	Filled Evaluation table, sharpened recommendations, and summary
3.0	08/11/2025	Bradley Saucier	Revised executive summary and recommendations

Executive Summary

This document recommends deploying "Draw It or Lose It" as a containerized Java backend on Ubuntu Server 24.04 LTS with a Progressive Web App client to maximize market reach, performance, and security. Expanding beyond Android is a critical strategic move that will position the game to attract a larger user base and increase market competitiveness. The primary challenge in this evolution is ensuring smooth, synchronized gameplay for all players, regardless of the devices they use to connect. The recommended approach deploys a Java-based backend to manage a single, central game instance in memory. This authoritative service enforces unique identifiers for every game, team, and player, guaranteeing consistent and accurate state management. By exposing functionality through a stateless REST API and storing game data within a relational database, the system can scale efficiently and integrate seamlessly with high-performance hosting solutions. This clear separation between backend services and the frontend interface enables rapid iteration, straightforward cross-platform deployment, and easier long-term maintenance. Overall, this approach delivers immediate value while positioning The Gaming Room for sustainable growth and adaptability in a competitive digital landscape.

Requirements

The Gaming Room needs a web-based version of Draw It or Lose It that can run on multiple platforms. A game must support one or more teams, and each team must have multiple players. All game, team, and player names must be unique. Only one instance of the game should run in memory at a time. The system must respond quickly, keep data secure, and work the same across browsers and devices.

Design Constraints

The development of this game is subject to several design constraints specific to its target deployment environment and functional requirements. First, the application must ensure that only a single instance of the game exists in memory at any time. This constraint is addressed by using the singleton design pattern to centralize all game logic and maintain a consistent state. Second, all names within the system - including game, team, and player names - must be unique to prevent confusion during user interaction and to support reliable data lookup. These identifiers are enforced through both validation logic in the application and constraints at the database level. Another constraint arises from the requirement that the system function as a web-based application. The design must support stateless communication across HTTP to enable scaling. Additionally, the application must respond rapidly to user actions, especially during the one-minute gameplay rounds. To meet these timing demands, the system relies on in-memory caching for read-heavy operations and minimal latency in API responses. Finally, the system must implement security controls to protect user information. All client-server communication occurs over encrypted channels, and user credentials is stored using secure hashing techniques. These constraints shape the overall software design, requiring careful attention to architecture and data integrity.

System Architecture View

The system will follow a simple multi-tier layout, with a client interface, an application backend, and a data storage layer. The client will connect to the backend over secure channels, and the backend will handle all core logic and database interactions. Storage will be separated from the application layer to improve security and maintainability. This layout keeps components isolated, supports future scaling, and provides a clear foundation for adding more detailed architecture specifications as the project evolves.

Domain Model

The UML class diagram illustrates a robust system built on core object-oriented principles aligned precisely with project requirements. Central to this structure is an abstract class named Entity, which encapsulates common attributes such as id and name, promoting code reusability across related classes. Classes representing Game, Team, and Player extend from Entity, streamlining the design by reducing redundancy.

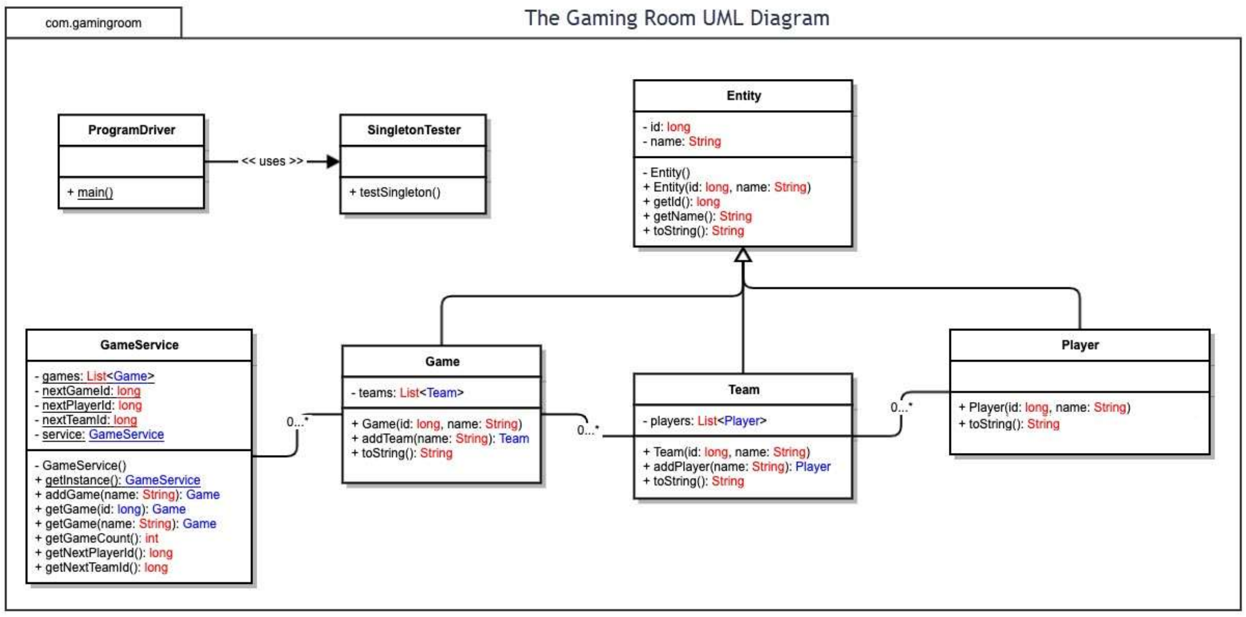
The GameService class functions as the central controller for all gameplay operations. Implemented using the singleton design pattern, it ensures exactly one authoritative instance manages the game state throughout the application's lifecycle. This centralized control guarantees consistent gameplay logic and simplifies state management.

The Game class maintains a direct hierarchical relationship with Team objects, each of which contains multiple Player objects. Although the UML diagram visually depicts these relationships using aggregation (open diamonds) for simplicity, the operational relationship among these classes reflects composition. Teams are inherently dependent on their parent Game, and Players likewise depend directly on their parent Team. This tightly coupled structure enforces clear lifecycle dependencies - when a Game instance is deleted, its associated Teams and Players are automatically removed, preserving data consistency and integrity.

The ProgramDriver class acts as the primary entry point to initiate application logic. It integrates a SingletonTester class specifically designed to validate that the singleton pattern functions correctly, confirming the existence of only one GameService instance throughout runtime.

Key object-oriented concepts - including abstraction, encapsulation, inheritance, and polymorphism - form the technical backbone of this model. The Entity class exemplifies abstraction by isolating shared attributes away from specific implementations. Encapsulation is demonstrated by limiting class visibility, exposing only necessary methods while maintaining internal logic privately. Inheritance facilitates structured attribute sharing among Game, Team, and Player, while polymorphism allows uniform handling of these classes when required.

Collectively, this model delivers a reliable, clean, and maintainable architecture that fully meets the client’s immediate needs and remains adaptable for future expansions.



Evaluation

Each operating platform and device type was evaluated against the project's requirements with a focus on stability, scalability, cost, and operational suitability. The analysis considers how server-side performance, licensing, maintenance, and integration capabilities align with the needs of The Gaming Room.

Development Requirement	**Mac**	**Linux**	**Windows**	**Mobile Devices**
Server Side	Limited scalability, costly hardware, best for testing or iOS builds only.	Stable, scalable, free licensing, strong cloud support with Docker and Kubernetes.	Higher license fees, GUI overhead, slower Java setup, frequent reboots, limited container support.	Inadequate hardware and network reliability for server hosting.
Client Side	Safari requires special testing, high expectations from Mac users, secondary priority.	Mainly developer-focused, reliable for internal tests, strong Chrome and Firefox support.	Largest user base, critical to test extensively on Chrome and Edge across diverse hardware.	Essential target, must support all screen sizes, connectivity issues, and ensure fast response.

Development Tools	Good IDE support, required for iOS builds, additional costs for hardware and developer account.	Robust, free toolset, excellent automation, container-friendly, ideal for full-stack workflows.	Adequate IDE availability, slower processes, added configuration, paid licenses increase costs.	Android Studio and Xcode free, physical device tests required, iOS distribution incurs fees.

Recommendations

The Gaming Room operates "Draw It or Lose It" on Ubuntu Server 24.04 LTS with a Progressive Web App client to deliver a secure, high-performance, and scalable gaming platform. This is a proven, stable operating system used worldwide for high-demand cloud applications, and it holds a significant market share in cloud deployments (Canonical, 2024). It is fast, secure, does not require expensive licenses, and receives official updates and security patches for the next 10 years. Using Linux ensures the game can run efficiently inside containers like Docker and be managed with Kubernetes, which makes it easy to add or remove server capacity as needed (Cloud Native Computing Foundation, 2023). While some teams are more familiar with Windows, Linux has a large support community, which mitigates the learning curve.

On the player side, the game is delivered as a Progressive Web App (PWA) built with React. This means players can run it in a web browser on almost any device - Windows, Mac, Android, or iOS - without needing to install a traditional app. Native mobile apps could be added later if needed.

Linux's design is ideal for the game's high-traffic, real-time needs. It isolates different services using kernel features like cgroups and namespaces, so one problem cannot crash the entire system (Khronos Group, n.d.). Its networking tools, which leverage efficient I/O event handling mechanisms like epoll, can handle thousands of players connected at the same time without slowing down (Khronos Group, n.d.). On mobile, iOS devices generally have stronger built-in security than Android devices, which vary in their protection levels. The game's design will account for this by using stricter security rules for devices that do not have the same hardware-based protections. Its modular kernel structure allows fine-grained control over processes and drivers, and its preemptive multitasking with advanced scheduling ensures consistent performance under load.

Data is stored using the right tool for each type of information. Player accounts, game settings, and scores is kept in PostgreSQL 16, a powerful and reliable database. Player drawings and avatars is stored in MinIO, a system built to handle large files efficiently and securely. All data will be backed up regularly following the 3-2-1 rule - three copies of the data, stored on two types of media, with one copy kept offsite. This strategy ensures minimal downtime and data loss.

Memory will be managed carefully to keep the game fast and stable. The operating system will load data into memory only when needed, and large memory pages will be used to speed up the database. Each container will have limits to prevent one service from taking all available memory. The Java system running the backend uses the Garbage-First Garbage Collector (G1GC), the default in modern JDKs, to clean up unused memory without slowing the game. The application will reuse certain objects in memory rather than constantly creating new ones.

The system is designed to run across multiple servers and handle failures without bringing the game down. Communication will be split into two channels: regular requests like logging in or starting a game uses standard web requests (REST), while live gameplay updates uses WebSockets for instant, two-way communication. To prevent cascading failures between services, the system implements a circuit breaker pattern, allowing the system to fail fast and recover gracefully when a dependency is unresponsive (Object Management Group, 2023). Detailed monitoring of logs, metrics, and system traces makes it possible to spot and fix problems quickly.

Security will be layered and thorough. All data sent between systems is encrypted with the latest version of TLS. Data stored in databases, object storage, and backups is encrypted so it cannot be read even if stolen. Players will log in using short-lived tokens that expire quickly and are stored securely - in cookies for web browsers and in built-in secure storage on mobile devices. Sensitive information like passwords and keys is kept out of the code and stored in a secure secrets management system. All player input will be checked to prevent malicious attacks, and all output will be encoded to stop harmful scripts from running. The application's security controls is designed to meet the OWASP Application Security Verification Standard (ASVS) Level 2, which is appropriate for applications handling sensitive user data (OWASP Foundation, 2023). The game only collects the personal information it truly needs and will protect it with strong access controls and auditing.

This plan provides The Gaming Room with a platform that is fast, reliable, secure, and ready to grow. It minimizes costs, protects player data, and ensures the game runs smoothly for everyone, regardless of their device.

References

Canonical. (2024). *Ubuntu Server powers the world's cloud*. Canonical. https://ubuntu.com/server

Cloud Native Computing Foundation. (2023). *Kubernetes production usage report*. CNCF. https://www.cncf.io/reports/

Khronos Group. (n.d.). *Vulkan and Linux kernel features for high-performance networking*. The Khronos Group. https://www.khronos.org

Object Management Group. (2023). *Circuit breaker pattern for microservices*. OMG. https://microservices.io/patterns/reliability/circuit-breaker.html

OWASP Foundation. (2023). *Application Security Verification Standard 4.0*. OWASP. https://owasp.org/www-project-application-security-verification-standard/