THE GAMING ROOM

Draw It or Lose It Web Game Application
CS 230 Project Software Design

Version 3.0

Table of Contents

CS 230 Project Software Design 1

Table of Contents 2
Document Revision History 2
Executive Summary 3
Requirements 3
Design Constraints 3
System Architecture View 4
Domain Model 4
Evaluation 6
Recommendations 9

Document Revision History

Version	Date	Author	Comments
1.0	10/16/2025	Michael Rodman	Initial version of software design document for Project One, including completed Executive Summary, Requirements, Design Constraints, and Domain Model sections.
2.0	11/30/2025	Michael Rodman	Added Evaluation section (Server Side, Client Side, Development Tools) for Project Two; no major changes to previous sections
3.0	12/14/2025	Michael Rodman	Added the Recommendations section for Project Three (operating platform, OS architecture, storage, memory, distributed systems/networking, and security recommendations).

Executive Summary

The Gaming Room currently offers a team-based drawing and guessing game, **Draw It or Lose It**, as a native Android application. They would like to expand this game into a **web-based, multi-platform application** so that players can participate from different devices and operating systems. To support this goal, the new solution must manage multiple games, teams, and players at once while ensuring that game and team names remain unique and that only one authoritative instance of the game management service exists in memory.
To meet these needs, I am proposing a **server-side game service** implemented in Java that uses established **object-oriented design principles** and **software design patterns**. A shared Entity base class will provide common identifiers and names for all major domain objects, while a singleton GameService will coordinate the creation and lookup of games, teams, and players. Iterator-based access to internal collections will preserve encapsulation while still allowing the application to verify uniqueness and retrieve objects efficiently. This design will prepare The Gaming Room to deploy Draw It or Lose It in a web-based, distributed environment and to extend the application to additional platforms in the future.

Requirements

Business Requirements

- Support a web-based version of **Draw It or Lose It** that can serve multiple platforms and device types.
- Allow **one or more teams** to participate in a game.
- Allow **multiple players** to be assigned to each team.
- Enforce **unique game names** so users can reliably select an existing game or create a new one.
- Enforce **unique team names** within the system to avoid confusion when players join teams.
- Provide a design that can be extended to additional platforms and features in later phases.

Technical Requirements

- Represent games, teams, and players as Java classes with consistent identifiers and names.
- Provide a **single, shared instance** of the game management service in memory (singleton pattern).
- Generate and manage **unique identifiers** for games, teams, and players.
- Maintain internal collections of games, teams, and players while **preserving encapsulation**.
- Use the **iterator pattern** to traverse these collections when searching for existing names or retrieving specific objects.
- Structure the design so it can run in a **web-based distributed environment**, where multiple clients may access the game service.

Design Constraints

The primary design constraint is that **Draw It or Lose It must run as a web-based, distributed application** rather than as a single, local Android app. This means that the game logic must be centralized in a service that can handle simultaneous requests from many clients, potentially running on different operating systems and devices. The design must therefore separate **server-side game management** from client-side user interfaces, so that future web, mobile, or desktop clients can all interact with the same underlying game service.
Because the game will run in a distributed environment, the application must **manage shared state carefully**. Only one authoritative instance of the GameService should exist in memory at a time to

coordinate games, teams, and players, which constrains the design to use the **singleton pattern**. At the same time, the game and team names must remain unique across all active sessions, so the code must enforce uniqueness via **controlled access to internal collections** rather than allowing direct modification. Using the **iterator pattern** to search for existing names allows the application to check these constraints while still hiding the internal data structures, which supports encapsulation and future changes.

Finally, the solution must account for **scalability and maintainability** within typical web hosting environments. Memory usage has to be efficient because multiple games and teams may be active at once, and the code must be organized so it can be tested, deployed, and extended without major rewrites. Using a shared Entity base class for common attributes, clear object relationships, and standard Java collections helps keep the design simple, reusable, and easier to adapt to different hosting platforms or frameworks later on.

System Architecture View

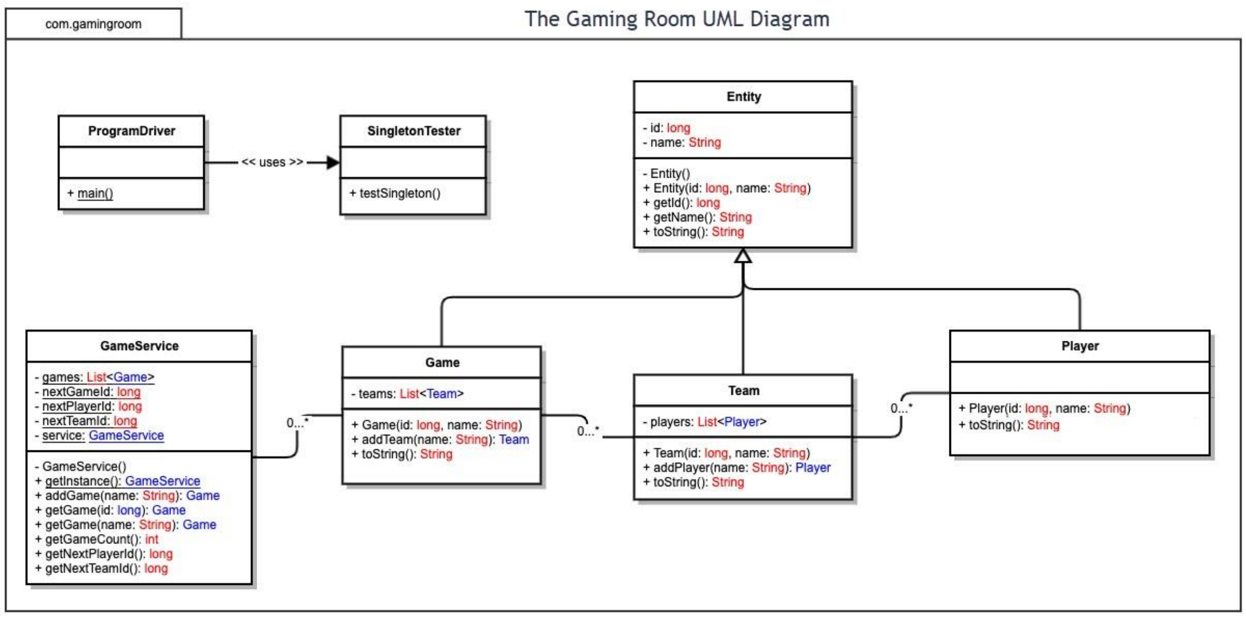
Draw It or Lose It will use an n-tier web architecture with client devices (browsers) connecting to a server-side game service. Requests will flow through a load balancer to stateless application instances. Persistent data (users/teams/game metadata) will be stored in an RDBMS, while image assets will be stored in object storage and delivered via a CDN. Centralized logging/monitoring will be used to track application health and performance across instances.

Domain Model

The UML class diagram for The Gaming Room application defines a small set of related classes that model the game domain. At the core is an abstract Entity base class that holds the common attributes id and name and provides basic behavior such as constructors, accessors, and string representation. The domain classes Game, Team, and Player all **inherit from Entity**, which means they automatically share the same identifier and naming structure while adding their own specialized behavior. This use of **inheritance and encapsulation** avoids duplicated code and makes it easier to maintain consistent identity rules across the application.

The Game class represents a single instance of Draw It or Lose It and maintains a collection of Team objects, reflecting the requirement that a game can have one or more teams. Similarly, each Team maintains a collection of Player objects, satisfying the requirement that teams have multiple players. These associations are modeled as one-to-many relationships in the UML diagram (a game has many teams, and a team has many players). The GameService class manages a list of Game objects and is responsible for creating new games, teams, and players and for enforcing uniqueness of game and team names. The ProgramDriver and SingletonTester classes act as clients that use GameService to exercise and verify the behavior of the system.

Several **object-oriented programming principles and design patterns** are illustrated in this domain model. Inheritance is used through the shared Entity base class, while encapsulation is preserved by keeping collections (lists of games, teams, and players) private and exposing them only through controlled methods. The **singleton pattern** is applied to GameService so that only one instance manages all domain objects in memory. The **iterator pattern** is used when traversing collections to search for existing names or entities, allowing access to the elements without exposing the underlying list implementation. Together, these principles and patterns help the application efficiently enforce unique names, manage identifiers, and satisfy the client’s requirements in a maintainable way.


com.gamingroom
The Gaming Room UML Diagram
ProgramDriver
+ main()
<< uses >>
SingletonTester
+ testSingleton()
Entity
- id: long
- name: String
- Entity()
+ Entity(id: long, name: String)
+ getId(): long
+ getName(): String
+ toString(): String
GameService
- games: List<Game>
- nextGameId: long
- nextPlayerId: long
- nextTeamId: long
- service: GameService
- GameService()
+ getInstance(): GameService
+ addGame(name: String): Game
+ getGame(id: long): Game
+ getGame(name: String): Game
+ getGameCount(): int
+ getNextPlayerId(): long
+ getNextTeamId(): long
0...*
Game
- teams: List<Team>
+ Game(id: long, name: String)
+ addTeam(name: String): Team
+ toString(): String
0...*
Team
- players: List<Player>
+ Team(id: long, name: String)
+ addPlayer(name: String): Player
+ toString(): String
0...*
Player
+ Player(id: long, name: String)
+ toString(): String

<u>Evaluation</u>

Development Requirements	Mac	Linux	Windows	Mobile Devices
Server Side	macOS can host a web-based application using tools like Apache, Nginx, or Node.js, but it is not commonly used as a production server OS. It is more suited for development and small internal deployments than for large-scale internet-facing hosting. Licensing is included with Mac hardware, but running multiple macOS server instances in the cloud is limited and usually more expensive. Overall, Mac is viable for development or light hosting, but not ideal as the primary production server platform.	Linux is the dominant platform for hosting web-based applications and is highly suitable for Draw It or Lose It's server side. It supports popular stacks like LAMP/LEMP (Linux, Apache/Nginx, MySQL/PostgreSQL, PHP/Java/Node.js) and scales well to thousands of concurrent users. Most Linux distributions (Ubuntu, Debian, CentOS, etc.) have no OS licensing cost, which keeps server expenses low. It also has strong community and enterprise support, security hardening options, and extensive tooling for deployment and monitoring.	Windows Server is a strong option for hosting if the team wants tight integration with .NET, C#, and IIS for web hosting. It offers good management tools (e.g., GUI admin tools, Active Directory integration), but licensing costs can be significantly higher than Linux, especially as more cores and instances are added. It scales well and has robust enterprise support but may require more careful cost planning and patch management. Overall, it is a good choice if the tech stack is already Windows/.NET-focused.	Mobile devices (Android/iOS phones and tablets) are not suitable as primary web servers for a large-scale game like Draw It or Lose It. They have limited resources, unstable connectivity, battery constraints, and are not designed for persistent hosting. At most, a mobile device could host a small local development server for testing, but production hosting must stay on desktop/server operating systems. Therefore, mobile platforms should be treated only as clients, not server hosts.

Client Side	On macOS, the client will run inside a modern browser (Safari, Chrome, Firefox) using a responsive HTML/CSS/JavaScript frontend. Development effort focuses on ensuring the web UI is tested across Mac browsers and different screen sizes. Cost and time are mainly in QA and front-end optimization rather than OS-specific coding. Expertise needed: web technologies (HTML5, CSS, JS/TypeScript, a front-end framework) and basic Mac testing.	On Linux, players will also interact with the game through web browsers such as Firefox or Chrome. Because the client is browser-based, no Linux-specific version of the game code is required, but testing across common Linux distros and browsers is important. Development cost and time remain manageable, as most work is shared with other desktop platforms. Expertise focuses on web development plus some familiarity with Linux environments for testing and troubleshooting.	On Windows, the HTML-based client will run in Edge, Chrome, Firefox, or other browsers, so the main concern is cross-browser compatibility and performance. Ensuring the UI behaves correctly with different DPI settings, resolutions, and input devices (mouse, touch screens on some devices) requires thorough testing. Development cost is mostly in browser testing and possibly minor adjustments for Windows-specific quirks. Expertise needed is the same core web stack with additional QA on common Windows setups.	On mobile (Android and iOS), the game will run in a mobile browser as a responsive web app. The primary development considerations are responsive design, touch input, and performance on smaller screens and lower-power CPUs. Extra time is required for testing across different devices, screen sizes, and mobile browsers. The team may also consider adding a “progressive web app” (PWA) experience so users can “install” the game from the browser, which adds some offline and home-screen features without building full native apps.

Development Tools	On macOS, developers can use cross-platform languages and tools such as Java, C#, JavaScript/TypeScript, and frameworks like Spring Boot, ASP.NET Core, Node.js, or Django. Common IDEs and editors include IntelliJ IDEA, Visual Studio Code, Xcode (for any Mac-specific or iOS-related work), and JetBrains tools. Most tools have free tiers or community editions, though some JetBrains/enterprise IDEs require paid licenses. macOS is a good "all-rounder" dev environment for both server and client web development.	Linux works very well for full-stack web development, especially for the same stack that will run in production. Languages like Java, Python, JavaScript/Node.js, Go, and PHP are first-class citizens, and tools like VS Code, IntelliJ IDEA, and command-line utilities (Git, Docker, CI/CD tools) are widely available. Many of these tools are open source or have free editions, keeping licensing costs low. Using Linux for development can reduce environment differences between dev and production servers.	Windows supports a wide variety of languages and tools, including strong support for C#, .NET, and Visual Studio, which is ideal if the team chooses an ASP.NET Core + IIS stack. It also supports Node.js, Java, Python, and containers via Docker Desktop, as well as editors like VS Code and JetBrains IDEs. Some tools (like full Visual Studio Professional/Enterprise or certain JetBrains products) require paid licenses, which can increase costs. The flexibility is high, but careful tool selection helps manage licensing and training.	Mobile devices themselves are not primary development environments, but they are critical test targets. For mobile client testing, developers use tools and emulators on desktop OSes—Android Studio (Android emulators), Xcode (iOS simulators on macOS), and browser dev tools for responsive testing. No additional OS license costs exist on the devices, but there may be costs for Apple Developer accounts or Google Play-related services if a PWA or hybrid/native wrapper is later distributed. The main impact is extra testing effort and familiarity with mobile debugging and profiling tools.

Recommendations

Operating Platform:
I recommend hosting the Draw It or Lose It web-based game service on a **Linux server platform** (for example, an Ubuntu Server LTS distribution) running in a **cloud environment**. Linux is the dominant server OS for modern web hosting and distributed applications, and it provides strong performance, stability, security hardening options, and cost advantages due to the lack of per-instance OS licensing. A cloud-hosted Linux server platform also supports the client's need to expand to additional computing environments by enabling elastic scaling, high availability, and standardized deployments as player demand grows. Windows Server would also work (especially for a .NET/IIS stack), but Linux is recommended here due to lower licensing costs and stronger alignment with common cloud-native hosting patterns.

Operating Systems Architectures:
Linux uses a **monolithic kernel architecture** (with loadable kernel modules) that provides efficient access to core services such as CPU scheduling, memory management, device I/O, and the networking stack. Applications run in **user space**, separated from kernel space by hardware-enforced protection, which improves stability and security (a user process cannot directly access protected kernel memory). Linux supports a robust **process and thread model** for concurrency, which is important for handling many simultaneous games/teams/players. It also provides a mature **TCP/IP networking stack**, firewall capabilities, and service management (commonly via **systemd**) that supports reliable startup, monitoring, and recovery of the game service. In practice, this architecture is also well suited to **container-based deployments** (e.g., Docker) because Linux natively supports key isolation primitives such as namespaces and control groups, allowing consistent builds and predictable runtime behavior across development, testing, and production environments.

Storage Management:
A hybrid storage approach is recommended to match the game's data types and performance needs. Use:

- A **relational database management system (RDBMS)** (e.g., PostgreSQL or MySQL) to store structured data such as users, teams, game sessions, and configuration (this supports integrity, relationships, and reliable transactions).
- **Object storage** for the stock drawing/image library and other large static assets, since storing large binaries in a database is typically less efficient and can increase database load.
- A **CDN** (content delivery network) in front of object storage to reduce latency and bandwidth on the application servers for frequently requested images.
  On the Linux server itself, use a reliable journaling file system such as **ext4 or XFS** for the OS/application runtime and logs. Implement scheduled backups and retention policies for the database and object storage to support recovery from accidental deletion, corruption, or operational failures.

Memory Management:
Linux provides **virtual memory** per process, using paging to map each process's address space to physical RAM, which helps isolate workloads and prevents one process from directly impacting another. For Draw It or Lose It, the server should avoid holding large image binaries in memory; instead, the application should keep only lightweight **metadata** (IDs/URLs) in memory and deliver image assets from object storage/CDN. The application server should use **bounded caching** (with size limits and eviction policies) for frequently accessed metadata (such as puzzle lists or image references) so memory usage

remains predictable under heavy concurrency. If the Java-based game service is deployed, memory management also includes **JVM heap configuration** (right-sizing the heap, monitoring GC behavior) to reduce latency spikes and prevent out-of-memory conditions. If containers are used, Linux control groups (**cgroups**) can enforce memory limits so that a single instance cannot exhaust host memory, improving overall stability when scaling out multiple instances.

Distributed Systems and Networks:
To support communication across many client platforms (Windows/macOS/Linux browsers, and mobile browsers), implement Draw It or Lose It as a **server-authoritative web service** accessed over standard network protocols. A recommended approach is:

- Clients communicate with the server over **HTTPS** for standard requests (account actions, joining games, retrieving game state).
- For real-time gameplay updates (timers, guesses, round state), use **WebSockets** or server-sent events so clients receive timely updates without excessive polling.
- Deploy the application tier as **stateless** instances behind a **load balancer** so the system can scale horizontally and continue operating if an instance fails.
- Store shared state (active game sessions, team/player membership, scores) in a centralized data store (database and/or cache) so any server instance can handle a request consistently.
  Because distributed systems depend on connectivity, the design should account for outages and transient failures by using timeouts, retries with backoff, and health checks. If connectivity between a client and server is interrupted, the client should be able to reconnect and restore state from the authoritative server store. This approach allows the game to operate across multiple platforms reliably while minimizing single points of failure.

Security:
User protection should be applied **in transit**, **at rest**, and **within the platform**. Recommended controls include:

- **TLS/HTTPS everywhere** to encrypt traffic between clients and the server and to protect credentials and gameplay data in transit.
- Strong authentication and authorization (for example, session tokens or JWTs) and **least-privilege access** for internal services (database, storage, logging).
- **Encrypt sensitive data at rest** (database encryption where supported and encrypted storage volumes). Never store plaintext passwords; store salted hashes using a strong password-hashing algorithm (e.g., bcrypt/Argon2).
- Use secure secret handling (environment variables or a secrets manager) rather than hardcoding secrets in source code.
- Enable OS-level protections on Linux such as strict file permissions, firewall rules, timely patching, and optional mandatory access controls (SELinux/AppArmor) for hardened deployments.
- Add application-layer protections such as input validation, rate limiting (to reduce brute force and abuse), and logging/monitoring to detect suspicious behavior.
  These measures protect user information on and between platforms while leveraging Linux's mature security model and operational tooling for secure hosting.