CS 230 Project Three – Software Design Document

Student: Jared Borck
Course: CS 230
Date: 6/21/2025

Version History

Version	Date	Author	Comments
3.0	06/21/2025	Jared Borck	Final version with Project Three complete

---

Executive Summary

The Gaming Room has requested a web-based version of their current Android game, Draw It or Lose It. The updated version allows multiple teams and players to compete by guessing drawings during timed rounds. Our goal is to build a scalable and maintainable version of the game that works across platforms and devices.

To support this, the Java application uses the Singleton pattern to manage a single instance of the game engine and the Iterator pattern to enforce name uniqueness when adding games, teams, and players. Inheritance is used to streamline shared properties between entities. This structure supports easy expansion and adapts well to distributed, multi-platform deployment.

Requirements

- The game must allow one or more teams per match.
- Each team must support multiple players.
- Game, team, and player names must be unique.
- Only one instance of the game engine should exist in memory.
- The application must be scalable and web-compatible.

Design Constraints

The application must operate in a distributed, web-based environment.

- Name uniqueness must be enforced for games, teams, and players.
- Only one instance of the game service must be created using the Singleton pattern.
- The application must support eventual deployment on multiple operating systems and platforms.

These constraints require using design patterns that manage memory, prevent duplication, and enforce data consistency across a shared environment.

System Architecture View

(No input required for this section.)

Domain Model

The UML diagram provided includes the following key classes: GameService, Game, Team, and Player. Each of these classes, except GameService, inherits from a common base class called Entity, which provides shared attributes like id and name.

- GameService uses the Singleton pattern to ensure only one instance runs at a time.
- Game contains a list of teams and ensures each team added has a unique name.
- Team contains a list of players and ensures each player added has a unique name.
- Player holds player-specific info and also inherits shared behavior from Entity.

Evaluation

Requirement	Mac	Linux	Windows	Mobile Devices (iOS & Android)
Server Side	Uncommon and expensive. Requires Apple hardware.	Excellent for hosting. Free, stable, secure. Works well with Docker/Kubernetes.	Usable, but less stable in production. Licensing costs may apply.	Not suitable for hosting. May use Firebase for light tools.
Client Side	Needed to support Mac users. Expensive hardware. Safari testing needed.	Smaller user base, but web compatibility is good.	Most common desktop platform. Easy to support.	Essential. Largest user base. Use responsive design or mobile frameworks.
Development Tools	Xcode, IntelliJ, Java, Maven. High cost for iOS support.	Eclipse, IntelliJ, Java, Maven. Open-source.	IntelliJ, VS Code, Java, Maven. Familiar and widely supported.	Android Studio, Xcode, Java, Kotlin, Swift. May require separate or cross-platform dev skills.

Recommendations

1. Use Linux for Server Hosting

Linux offers excellent performance, security, and scalability at no licensing cost. It supports tools like Docker and Kubernetes, making it the best choice for deploying the game server in a distributed environment.

2. Support Cross-Platform Clients via Responsive Web Design

To reach the largest audience, develop a responsive HTML/CSS/JS interface that functions across Windows, macOS, Linux, and mobile browsers. Frameworks like React or Vue can streamline this while maintaining performance across devices.

3. Minimize Platform-Specific Development

Native apps for iOS or Android are optional. Focus on web-based delivery to reduce development overhead. Only invest in native clients if future demand or performance requirements justify it.

4. Unify Development Stack Around Java + Web Tools

Continue using Java for the backend to maintain consistency. Pair it with Maven for dependency management and IntelliJ/Eclipse for IDE support. Frontend tooling can use standard web technologies (HTML, CSS, JS) to avoid splintering dev efforts.

5. Scale Securely

Implement HTTPS, user authentication, and session security using proven web standards. Use a layered approach with database validation, input sanitation, and role-based access control.

Operating System Architecture

Linux uses a monolithic kernel architecture, which places all core system functions—including memory management, device drivers, and system calls—into a single address space. This design provides faster performance and lower overhead by avoiding the need for excessive context switching. For Draw It or Lose It, this results in better responsiveness and system efficiency, especially during high user activity.

Storage Management

With over 1.6GB of image assets, Draw It or Lose It should use a hybrid storage strategy. Only essential images will be included in the app install package, while the rest can be downloaded from the cloud on demand. Images should be stored in optimized formats like WebP to reduce storage size. Organizing assets with a lightweight local database like SQLite will allow for quick indexing and access during gameplay, improving load times and keeping the app size manageable for users with limited space.

Memory Management

The game should implement lazy loading, where only the image needed for the current round is loaded into memory. After the image is rendered and the round ends, it should be released to free memory. A caching system for recently used images can improve performance in later rounds where reuse may occur. On mobile platforms like Android and iOS, native memory management tools such as BitmapFactory and memory pools should be leveraged to avoid crashes and ensure consistent performance, especially on lower-end devices.

Distributed Systems and Networks

Draw It or Lose It will benefit from a distributed microservices architecture. Key components—such as game logic, player sessions, and scoring—can be separated into services using Docker and orchestrated with Kubernetes. Services will communicate through REST APIs or WebSockets for real-time interaction. This structure supports high availability, automatic scaling, and failover handling. In the event of a node failure, Kubernetes can automatically restart services or redirect traffic, ensuring minimal disruption to gameplay.

Security

User data and game state integrity are critical. All data in transit should be encrypted using HTTPS. Authentication should be managed using modern standards like OAuth or JWT. Server-side validation and input sanitization should be enforced to prevent injection attacks and data corruption. Role-based access control (RBAC) can restrict actions based on user type (e.g., admin vs. player). On the server side, Linux-based tools like SELinux, iptables, and automatic security patching will provide strong system-level protection.