**Draw It or Lose It**

**CS 230 Project Software Design Document**

**Table of Contents**

1. Document Revision History
2. Executive Summary
3. Requirements
4. Design Constraints
5. System Architecture View
6. Domain Model
7. Evaluation
8. Recommendations

**Document Revision History**

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| 1.0 | 05/22/2025 | Adil Patel | Initial submission with complete software design document. |
| 2.0 | 06/12/2025 | Adil Patel | Updated Evaluation and Recommendations for Project Two. |

**Executive Summary**

The Gaming Room has requested the development of a web-based version of its game, **Draw It or Lose It**, which is currently only available on Android. The goal is to expand its availability across web, desktop, and mobile platforms while maintaining performance, security, and a seamless user experience.

This document outlines a scalable software design leveraging Java and patterns such as **Singleton** and **Iterator** to manage game instances and modular components. The design prioritizes reusability, scalability, and compatibility with distributed system environments, using Linux-based infrastructure for deployment.

**Requirements**

* Allow multiple teams per game
* Each team has multiple players
* Unique names must be enforced for games, teams, and players
* Only one instance of a game may exist in memory at a time
* System must check and prevent duplicate names

**Design Constraints**

This application must run in a **web-based distributed environment**, which introduces challenges such as:

* Ensuring **only one game instance** is active at a time using the Singleton pattern
* Enforcing **unique identifiers** across game entities
* Supporting **real-time, cross-platform client communication**
* Maintaining **data consistency and synchronization** across distributed components
* Optimizing performance for low-resource devices and ensuring **high availability**

**System Architecture View**

This section serves as a placeholder for future projects. It typically includes system tiers, physical infrastructure, and a logical communication/storage topology for full understanding of system design.

**Domain Model**

The domain model includes:

* **Entity superclass**: with fields id and name
* **Game**, **Team**, and **Player** subclasses:
  + Game contains a list of Team objects
  + Team contains a list of Player objects

Design patterns used:

* **Singleton**: ensures only one game instance exists in memory
* **Iterator**: allows controlled traversal through lists of teams and players

This object-oriented structure supports **inheritance**, **encapsulation**, and **modularity**, making the system scalable and maintainable.

**Evaluation**

**Development Requirements**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Category** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| **Server Side** | Viable with Apache/Nginx but less common in production due to hardware and cost limitations. | Preferred for scalability, performance, and cost-effectiveness. Excellent for web servers. | Popular in .NET environments. GUI support, but more costly and less flexible than Linux. | Not suitable for hosting due to limited resources and connectivity. |
| **Client Side** | Requires compatibility testing for Safari; Xcode needed for native apps. | Lower market share; primarily supports browser-based clients. | Most used desktop OS. Must support browsers and possibly .exe packages or cross-platform tools. | Dominant platforms; must support Android (Java/Kotlin) and iOS (Swift). |
| **Dev Tools** | IntelliJ, Eclipse, and Xcode supported; Git and Terminal used. | Supports Java, Python, C++, web development; tools include VS Code, Eclipse, etc. | Strong support via Visual Studio, IntelliJ, Eclipse; good for both .NET and Java development. | Not development platforms themselves; apps built using Android Studio and Xcode. |

**Recommendations**

1. **Operating Platform**  
   Recommend using **Linux (Ubuntu Server LTS)** for hosting. It is scalable, secure, open-source, and highly supported for distributed and cloud-based deployments.
2. **Operating System Architecture**  
   Use a **client-server model** with modular components. The backend runs on Linux using Apache or Nginx. Clients interact via **REST APIs** or **WebSocket** for real-time functionality. Docker and Kubernetes can containerize and orchestrate services for scalability and resilience.
3. **Storage Management**  
   Use cloud-based object storage (e.g., Amazon S3, Azure Blob) to store game assets, drawings, and player data. Local storage can use **XFS with LVM** for flexibility and performance. Cloud storage enables automatic backups and scalability.
4. **Memory Management**  
   Linux handles memory through paging and caching. It can manage heavy loads via **swap space** and **OOM handling**. Using containers, memory limits can be enforced per service to isolate performance bottlenecks.
5. **Distributed Systems and Networks**  
   Recommend adopting **microservices architecture** with stateless services for scalability. Use REST APIs for general communication and **WebSockets** for real-time game interactions. Kubernetes and Docker will manage load balancing and service discovery.
6. **Security**
   * Encrypt data **in transit (TLS/SSL)** and **at rest (AES-256)**
   * Use **bcrypt or Argon2** for password hashing
   * Authenticate with **OAuth 2.0 or JWT**
   * Protect backend with **firewalls, OSSEC**, and regular patches
   * Leverage Linux’s file permission and user management system