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	03/17/2025	Paul Garner	Initial draft of the software design document
2.0	04/14/2025	Paul Garner	Added and changed the recommendations and summary for project 3

Executive Summary

The Gaming Room has requested a web-based version of their Android game, Draw It or Lose It. The game must support multiple teams, multiple players per team, and unique game and team names to prevent duplicates. Additionally, only one instance of a game should exist in memory at any given time. To meet these requirements, we will implement software design patterns, including the Singleton Pattern (to ensure a single instance of the game service) and the Iterator Pattern (to enforce unique names). This document outlines the software design, system constraints, and recommendations for deployment across multiple platforms.

Requirements

- **Business Requirements:**
  - Develop a **cross-platform** web-based game.
  - Ensure **unique** game, team, and player names.
  - Support **multiple teams and players** per game session.
  - Maintain a **single instance of the game** in memory.
- **Technical Requirements:**
  - Implement **Singleton Pattern** for game instance control.
  - Implement **Iterator Pattern** for duplicate name checking.
  - Support **distributed environments** for multi-platform access.
  - Utilize **secure data handling** for user authentication and game state management.

Design Constraints

- **Web-Based Distributed Environment:**
  - The game must be accessible via **Mac, Windows, Linux, and mobile browsers**.
  - Latency issues must be minimized for **real-time gameplay**.
  - Server architecture must be **scalable** to handle increasing player loads.
- **Implications on Development:**
  - Requires **cloud-based hosting** (AWS, Azure, or GCP).
  - Must implement **RESTful APIs** for backend communication.
  - Database design must support **fast retrieval of player and game data**.

System Architecture View

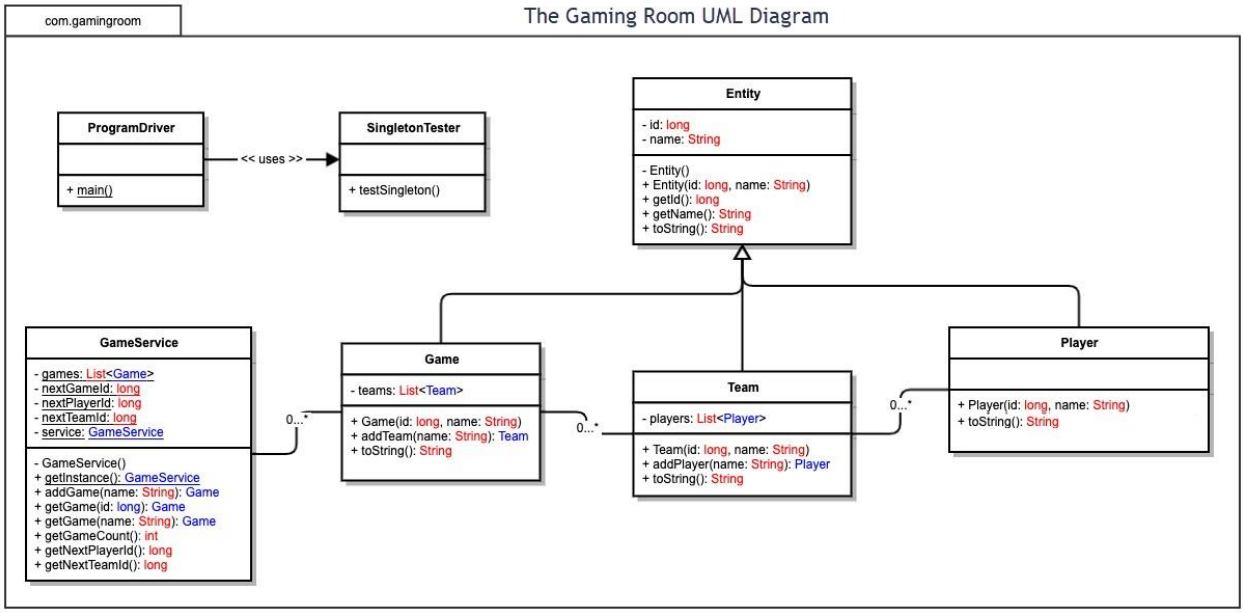
Please note: There is nothing required here for these projects, but this section serves as a reminder that describing the system and subsystem architecture present in the application, including physical components or tiers, may be required for other projects. A logical topology of the communication and storage aspects is also necessary to understand the overall architecture and should be provided.

Domain Model

The UML class diagram represents the object relationships in the game. The Entity class serves as a base for all key objects (Game, Team, Player) to ensure code reusability.

Object-Oriented Principles Used:

Inheritance: The Entity class allows Game, Team, and Player to share attributes like id and name.
Encapsulation: Each class keeps attributes private with getter methods to ensure data integrity.
Polymorphism: Methods like addGame(), addTeam(), and addPlayer() utilize the Iterator Pattern to enforce unique names.



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 supports Unix-based environments, making it stable for server hosting but less common for large-scale deployment.	Linux is the preferred choice for web applications due to stability, security, and scalability.	Windows is not commonly used for high-traffic web servers but may be required for Windows-specific services.	Not suitable for hosting, but mobile clients must be optimized for real-time communication.
Client Side	Development for macOS requires Swift or cross-platform frameworks.	Linux clients use Electron.js or web applications for browser-based play.	Windows development supports native applications or web-based solutions using frameworks like .NET and React.	Mobile clients must be optimized for performance using React Native, Flutter, or Progressive Web Apps (PWA).
Development Tools	Xcode, Swift, React, JavaScript frameworks.	VS Code, Java, Spring Boot, Node.js, React.	Visual Studio, .NET Core, Java, React.	Android Studio, Xcode, Flutter, React Native.

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**: We recommend running everything on a Linux-based server like Ubuntu or Red Hat using a cloud provider like AWS or Azure. Linux is reliable, secure, and cost-effective, plus it works well with most web development tools. It's a solid choice for getting Draw It or Lose It up and running on different devices and systems.

2. **Operating Systems Architectures**: The backend should run on Linux servers, while the frontend should be developed using cross-platform frameworks such as React or Angular for the web and Flutter for mobile. Linux uses a layered system that separates hardware interaction, kernel operations, and user processes. This makes it flexible and efficient for handling server loads, especially in a distributed setup like this game will use. It also supports microservices well, which lets us break up the game logic, player sessions, and image rendering into manageable parts.

3. **Storage Management**: **Database:**

   - MySQL or PostgreSQL for relational data storage.
   - **Cloud Storage:** AWS S3 or Firebase for handling game assets (e.g., images, user data).
   - **Caching:** Redis for optimizing real-time gameplay.

   With 200 high-definition image files averaging 8 MB each, we're looking at around 1.6 GB of asset storage just for the drawings. We'll use PostgreSQL or MySQL for structured game and player data, and Amazon S3 or Firebase to store the image files. These cloud services offer fast access, scalability, and redundancy. We'll also add Redis caching to quickly load frequently used images during gameplay.

4. **Memory Management**: The game server should use **garbage collection** for resource cleanup and **session management** to avoid memory leaks. Each round of gameplay involves rendering images rapidly, so efficient memory use is key. We'll preload necessary assets before the round begins and immediately release memory after the round ends. Linux's virtual memory and paging help manage system memory well, and tools like htop or memory profilers will help us monitor resource use and avoid memory leaks.

5. **Distributed Systems and Networks**:

   - The **web-based architecture** allows multi-platform communication via **RESTful APIs**.
   - **Load balancing** should be implemented to handle multiple game sessions simultaneously.

- **WebSockets** should be used for **real-time updates** between players.

Since the game needs to support multiple platforms and users at once, we'll rely on a distributed setup. RESTful APIs will handle communication between client and server, and WebSockets will keep everything in sync in real time. Load balancing will make sure no one server gets overloaded. If a server goes down, traffic can automatically be rerouted to another, keeping the game available to users.

6. **Security**: **User Authentication:**

- OAuth 2.0 or JWT for secure login.
- **Data Encryption:** HTTPS and AES-256 encryption for secure data transmission.
- **Role-Based Access Control (RBAC):** To restrict access to administrative features.

Security is a must. We'll use OAuth 2.0 or JWT to handle user logins securely, and all data will travel through HTTPS connections with TLS encryption. Sensitive data, like player credentials or session tokens, will be encrypted using AES-256. RBAC will make sure only authorized users can access certain parts of the system. We'll also regularly test the system and validate inputs to protect against attacks.

Final summary

The proposed Linux-based web application will provide a scalable, secure, and cross-platform solution for Draw It or Lose It. By leveraging Singleton and Iterator Patterns, we ensure a robust game engine that supports unique names and optimized memory management. This document outlines the software architecture and technical requirements needed to successfully deploy the game in a distributed environment. This updated design outlines how Draw It or Lose It can be expanded to a distributed, secure, and high-performance web-based game. By choosing Linux for the backend, cloud storage for assets, and tools that help us manage memory and security properly, we can deliver a smooth experience across all platforms. Everything in this setup is designed to keep the game fast, scalable, and user-friendly no matter where it's played.