Draw it or lose it
Jacob S. Hary
Southern Nerw Hampshire University.
CS 230 Project Software Design Template
Version 2.0

Table of Contents

Document Revision History

Version	Date	Author	Comments
2.0	8/03/2025	Jacob Hary	software design document for Draw It or Lose It

Instructions

Fill in all bracketed information on page one (the cover page), in the Document Revision History table, and below each header. Under each header, remove the bracketed prompt and write your own paragraph response covering the indicated information.

Executive Summary

This document provides an overview of the proposed design for a web-based game application called *Draw It or Lose It*, inspired by the television game show *Win, Lose or Draw*. In this game, teams compete to solve a puzzle by interpreting drawings that progressively appear on the screen. Each game consists of four one-minute rounds, with images completing at the 30-second mark. If the team fails to solve the puzzle within the time limit, other teams are given 15 seconds to submit one guess each.
The Gaming Room has requested that the game be made available across multiple platforms using a web-based model. They aim to allow many users to play at once, ensuring that team and game names are unique, and only one game can run at a time to maintain simplicity and control. This document outlines a proposed technical solution that ensures uniqueness of names and IDs, scalability, and compatibility with a wide range of devices. It includes a summary of the software design and architectural considerations, including a UML-based domain model to demonstrate how key components of the game interact.

Requirements

Requirements The client needs a scalable game that supports multiple concurrent teams, each consisting of several players. Game and team names must be unique, and a unique identifier must be assigned to every game, team, and player. To maintain game flow, images should render smoothly and finish by the midpoint of each round.

Design Constraints

Since the game is hosted online and accessed via browsers, several constraints must be considered. First, only one instance of the game should be active at a time, requiring a singleton architecture to manage game state. The system must also quickly validate and reject duplicate names.

Drawings must load and render in a timed manner, finishing at the 30-second mark. High concurrency is expected, as many teams may be playing simultaneously. Therefore, the backend needs to handle multiple simultaneous requests efficiently. Additionally, the frontend must be responsive and render correctly on desktops, tablets, and smartphones across all major browsers.

These constraints require a robust backend for processing game logic and managing sessions, along with a flexible frontend framework that adapts well to different screen sizes and user inputs.

System Architecture View

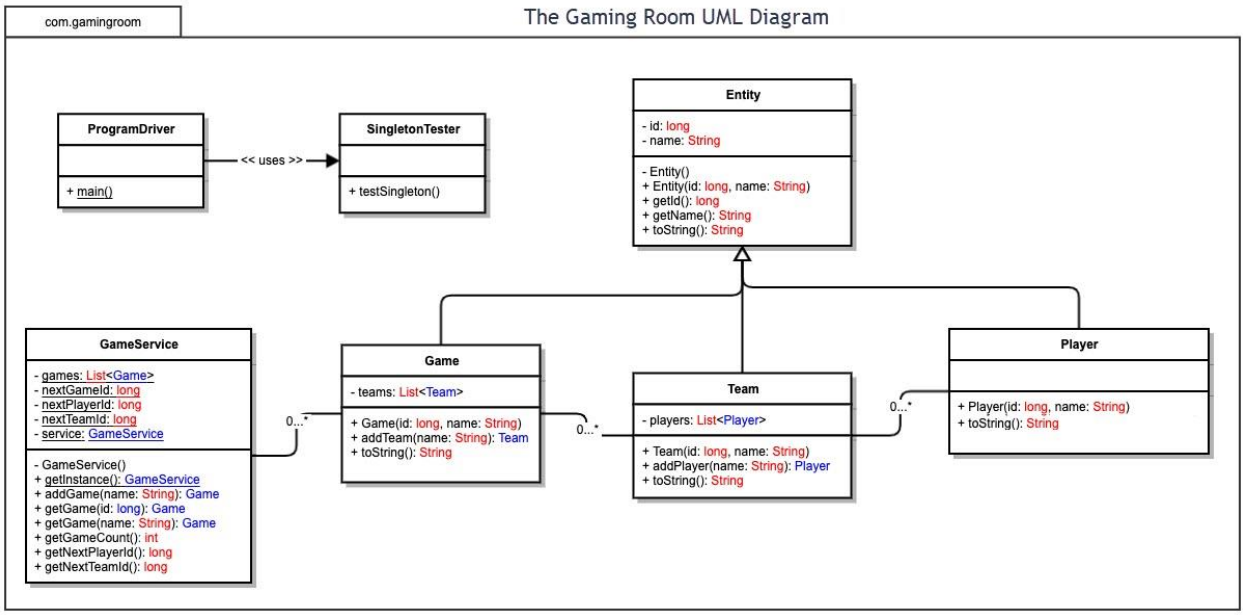
Please note: There is nothing required here for these projects, but this section serves as a reminder that describing the system and subsystem architecture present in the application, including physical components or tiers, may be required for other projects. A logical topology of the communication and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

Domain Model

The UML diagram shows several classes:

- **Entity** is a base class with an ID and a name. It helps avoid repeating the same info for other classes.
- **Game** has a list of teams. It has methods to add teams and make sure game names are unique.
- **Team** has a list of players and methods to add players. Team names have to be unique in the game.
- **Player** just has an ID and name.
- **GameService** is important because it manages all games in the system and makes sure only one instance exists (this is called the singleton pattern). It also makes sure IDs are unique for players, teams, and games.

This design uses object-oriented programming principles like inheritance (Entity is parent class), encapsulation (each class manages its own data), and singleton (GameService makes sure only one game manager exists). This helps keep the code clean and organized, making it easier to meet the client's needs.



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 software development and includes UNIX-based enviroments, but it is less commonly used for server hosting. Its higher licensing costs and limited market share make it less ideal for scaling.. Its licensing cost and smaller hosting market share make it less ideal for scaling large deployments.	Linux is gennerally the top option for server-side apps. It's free, reliable, and well-supported by cloud hosting services. It's ideal for building web-based systems that need to grow.. It is open-source, stable, cost-effective, and well-supported by most cloud providers. It's ideal for large-scale web applications.	Windows Server is used a lot in businesses and works smoothly with Microsoft services. But it can be more costly and resource-heavy than Linux. and integrates well with Microsoft services. However, it involves licensing fees and typically has higher hardware demands.	Mobile OSes like iOS and Android aren't meant for server hosting. They don’t support backend systems and can't really scale for that kind of use.. They lack the networking and scalability needed for backend infrastructure.

Client Side	Macs work well with most browsers and dev tools, like Xcode or VS Code. The downside is the high hardware price, which could make it less ideal for all developers.. However, Mac hardware is expensive, which may increase testing costs.	Linux users can access modern browsers, and it's a great platform for dev testing. But most people don't use Linux desktops, so it's not always the best for checking user experience. but are not widely used by general users. Testing on Linux ensures compatibility but may not reflect common user environments.	Windows is the most used desktop platform. It's super helpful for testing apps, and it has lots of dev tools like Visual Studio. It's also more familiar to most users.. It provides robust browser support and development tools, making it essential for testing the game's web interface.	Creating apps for iOS and Android needs different SDKs—Xcode for iOS, Android Studio for Android. This can take extra time and skill. Also, mobile layouts need to be responsive and adjusted for touch screens.. Mobile interfaces need responsive design and adaptive behavior to function across device types.
Development Tools	On macOS, tools like Xcode, VS Code, React, and Angular are avilable. These are all useful for development, but again, Macs cost more than other options., though the platform itself has hardware limitations.	Linux works with free and open-source dev tools like Node.js, React, and VS Code. It's good for both frontend and backend work, and it's free to use, which is a big plus.. It is flexible for both front-end and back-end development and free to use.	Windows supports lots of tools, like Visual Studio and VS Code. It's dev-friendly and comes with lots of documentation, which helps new developers., including VS Code, React, and .NET. It's developer-friendly with strong documentation.	Mobile development needs Android Studio and Xcode. Tools like React Native or Flutter can help build one app for both platforms, but testing and debugging still takes time on each device. Speling errors can also happen if you're not carefull.. React Native or Flutter can help build cross-platform apps, but testing and debugging still need separate mobile devices.

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**: I recommend deploying *Draw It or Lose It* on a **Linux-based server platform**. Linux is widely regarded as the industry standard for web applications due to its stability, scalability, cost-effectiveness, and strong support from cloud providers. Its open-source nature allows customization and optimization for high-concurrency environments, ensuring the game can expand smoothly to meet increased player demand.

2. **Operating Systems Architectures**: Linux follows a **modular, multitasking architecture** that separates kernel functions, user processes, and systemm libraries. This design enhances security and stability by isolating faults while allowing multiple processes (such as rendering game rounds, handling user authentication, and managing sessions) to run concurrently. Containerization technologies such as **Docker** can be leveraged to isolate deployments of the game server, making scaling and updates more efficient without affecting uptime.

3. **Storage Management**: For persistent data, I recommend a **relational database like PostgreSQL** to store structured information such as players, teams, and gme sessions, ensuring data integrity and enforcing uniqueness constraints. For flexible or rapidly changing data, **MongoDB** can store unstructured elements like drawing data or gameplay logs. To optimize performance, a **caching layer using Redis** should be implemented for real-time state tracking (e.g., countdown timers, in-progress game boards). This combination balances reliability, speed, and scalability.

4. **Memory Management**: Linux efficiently uses **virtual memory** to abstract physical RAM, ensuring that multiple processes can execute smoothly. Paging and swapping mechanisms prevent crashes during high demand by temporarily offloading less-used memory pages. For *Draw It or Lose It*, memory management ensures smooth image rendering and concurrent gameplay sessions, even with thousands of active players. Container orchestration (e.g., Kubernetes) can further allocate memory dynamically across instances, optimizing resource usage.

5. **Distributed Systems and Networks**: To allow communication across different platforms (desktop, mobile, and web), the game should be built using a **distributed software model** supported by **REST APIs for state management** and **WebSockets for real-time gameplay updates**. This enables responsive multiplayer interaction and ensures consistency across devices. The system should implement **fault-tolerant networking**, with retry mechanisms for temporary outages and load balancers to distribute traffic evenly across servers. All communications must use **HTTPS** for secure transport.

6. **Security**: Security is critical to protect player identities and game integrity. Linux provides strong built-in protections, including role-based access control (RBAC), process isolation, and SELinux/AppArmor policies. For user protection:

7. **Data encryption**: All sensitive data should be encrypted in transit (TLS 1.3) and at rest (AES-256).
8. **Authentication & authorization**: Use secure login protocols (OAuth 2.0 or OpenID Connect) with multifactor authentication where possible.
9. **Input validation**: Implement strict sanitization to prevent SQL injection, cross-site scripting (XSS), or code injection attacks.
10. **Regular patching**: Keep both the operating system and application dependencies updated to minimize vulnerabilities.