

Image Rendering - Win, Draw, or Lose It

**CS 230 Project Software Design Template**

Version 1.0

**Table of Contents**

**Document Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Comments |
| 1.0 | 09/27/25 | Renee | Filled out all sections of the document |

**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 software application will be an image rendering of *Win, Draw, or Lose It,* where players have 30 seconds to identify the slowly-rendering Image before it completes. Gameplay is structured by rounds and opportunities for bonuses. This document outlines the software design for the initial development phase of the game. Requirements are focused on gameplay mechanics, and instance management. This will lay the groundwork for hardware integration.

**Requirements**

**Team and Player Management**

* A game must support one or more teams.
* Each team must support multiple players.
* Team and game names must be unique to prevent naming conflicts and allow real-time validation during setup.

**Game Lifecycle**

* A game consists of four rounds, each lasting one minute.
* Images visualize by pixel and are fully rendered at the end of 30 seconds.
* If the active team fails to guess the puzzle, other teams may submit one guess each within a 15-second window.

**Instance Control**

* Only one active game instance may exist in memory at any given time.
* The service will be web-based and validated in real time.

**Environment Setup**

* The application must include setup instructions and tooling to assist The Gaming Room staff in deploying and managing the game environment.

**Design Constraints**

* **Single Game Instance:** The system must enforce a singleton pattern to prevent multiple game instances.
* **Unique Naming:** Game and team names must be validated against existing entries in real time to ensure uniqueness.
* **Rendering Timing:** Pixels are revealed over 30 seconds, with no additional drawing after that point.
* **Web-Based Architecture:** The application must be browser-accessible and platform-independent, requiring no client-side installation.
* **User Interface:** The UI should be intuitive for non-technical staff and players.

**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**

**UML Classes**

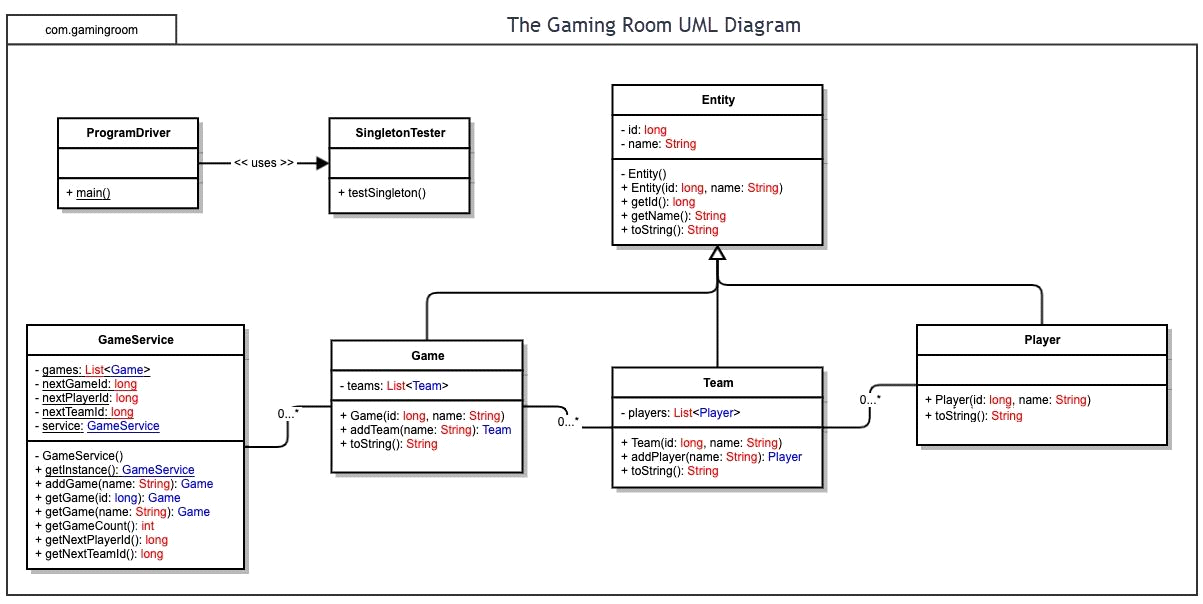
* ProgramDriver
* Acts as the entry point for the application.
* Uses the SingletonTester class to verify singleton behavior.
* SingletonTester
* Contains logic to test the singleton pattern used in GameService, as a public method called testSingleton().
* Entity (Superclass)
* Abstracts shared attributes: id and name.
* Inherited by: Game, Team, and Player.
* Uses private attributes long id and String name.
* Has private constructor, and uses public get id and name, creation, and toString() methods.
* GameService (Singleton)
* Central manager for all game-related operations.
* Maintains lists of Game objects and tracks ID generation.
* Associates with:
* Game (0...\*): manages multiple games.
* Provides methods to add, retrieve, and count games.
* Uses private attributes:
* List games
* Long nextGameId
* Long nextPlayerId
* Long nextTeamId
* Gameservice service (instance)
* Uses private constructor and public methods:
* Get Methods for Instance, Id, Game, Game count, Next player Id, and Next team Id.
* addGame
* Game
* Inherits from Entity.
* Contains a list of Team objects.
* Associates with:
* Team (0...\*): each game has multiple teams.
* Uses private attribute List teams.
* Uses public methods Game creation, add Team, and toString.
* Team
* Inherits from Entity.
* Contains a list of Player objects.
* Associates with:
* Player (0...\*): each team has multiple players.
* Uses private attribute List players.
* Uses public methods Team creation, add player, and toString()
* Player
* Inherits from Entity.
* Represents individual participants.
* Uses public method Player creation and toString.

**Object-Oriented Principles Demonstrated**

* Encapsulation
* Each class contains its own data and methods.
* GameService encapsulates game management logic, while Team encapsulates player management.
* Inheritance
* Game, Team, and Player inherit from Entity.
* Shared attributes like id and name are centralized.
* Abstraction
* Entity serves as an abstract representation of all named, identifiable objects.
* Polymorphism
* Polymorphism could be applied through different types of players or teams.
* Singleton Pattern
* GameService uses a singleton pattern to ensure only one instance exists in memory.

**How This Design Fulfills Requirements Efficiently**

* Team and Player Structure: Game - Team - Player hierarchy supports multiple teams and players.
* Unique Names and IDs: Entity provides a consistent structure for managing unique identifiers and names.
* Singleton Enforcement: GameService ensures centralized control and prevents multiple game instances. The SingletonTester ensures the functionality is there.



**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** | **Characteristics:**  Unix-based system.  **Advantages:**  Strong dev tools and easy integration to Apple's ecosystem. Stronger secruity measures than Windows.  **Weaknesses:** Limited options for server-grade hardware. Cross- compatibility may be more difficult. | **Characteristics:**  Open-source, highly customizable depending on environment, supports range of web stacks.  **Advantages:**  Reduced cost, secure, stable for long-term hosting.  **Weaknesses:**  Increase expertise requirement, more challenging UI for developers than others. | **Characteristics:**  Commonly used for hosting ASP.NET, or SQLServer Applications.  **Advantages:**  Strong compatibilty with other Microsoft services and technologies. User-friendly UI for development.  **Weaknesses:**  Costs for licensing will be higher, more at risk for security. | **Characteristics:**  Less useful for server-grade hosting due to memory and storage limits.  **Advantages:**  Can test mobile-first applications easily.  **Weaknesses:** Memory & Storage limits, sandoxing, and battery life make this a poor option. |
| **Client Side** | **Cost**: Requires Apple Hardware, which can be more expensive than other equipment.  **Time**: macOS development works quickly with Swift and XCode.  **Expertise**: Niche skills for macOS developers. | **Cost**: Because Linux tools and OS are open-source, the cost is much less than others.  **Time**: Time is increased for testing across distributions with a lesser used OS.  **Expertise:** May require more expertise depending on the specific OS. | **Cost:** Moderate option. More afforable equipment than Apple but expensive licensing costs.  **Time:** Reduced due to fleshed out development ecosystem.  **Expertise:** Will be easier to come by, as most developers work with Windows spaces. | **Cost:** Higher to develop for both iOS and Android.  **Time**: Can take longer to develop in both platforms. Some tools specifically for cross-platform can save time.  **Expertise**: Must know mobile specific development and UX best practices. |
| **Development Tools** | Languages:  Swift,  XCode, JavaScript.  IDEs: Visual Studio Code, IntelliJ IDEA.  Tools: Cocoa frameworks, Homebrew, Docker for Mac. | Languages: Python, PHP, JavaScript, Ruby, C++.  IDEs: VS Code, Eclipse  Tools: Git, Docker, Apache/Nginx, MySQL | Languages: C#, JavaScript, Python.  IDEs: Visual Studio, VS Code  Tools: IIS, .NET Core, SQL Server, | Languages: Java (Android), Swift (iOS)  IDEs: Android Studio, Xcode, VS Code.  Tools: Firebase, Expo, Gradle, CocoaPods. |

**Recommendations**

Analyze the characteristics of and techniques specific to various systems architectures and make a recommendation to The Gaming Room. Specifically, address the following:

* **Operating Platform**: I recommend developing “Draw It or Lose It” on a Windows server. It is stable, scalable, and has a thorough ecosystem for all elements of the application. It is more cost-effective than MacOS, and requires less specific expertise than Linux environments. Also, if in future enhancements we want to consider using the cloud for additional scalability, it will integrate seamlessly with Microsoft Azure.
* **Operating Systems Architectures**: Windows Server supports a **client-server architecture** with layered services:
* **Presentation Layer**: Delivered via web browsers using HTML5, CSS, and JavaScript
* **Application Layer**: Hosted on Internet Information Services, running backend services built in Java.
* **Data Layer**: Managed through SQL Server
* **Storage Management**: I recommend Microsoft SQL Server. It offers high-performance relational data management, supports real-time queries, and integrates seamlessly with Windows-based applications. In terms of hardware, SSD storage with a minimum of 500 GB should be used to account for increasing the image volume and game logs as time goes on.
* **Memory Management**: Because the game uses a singleton pattern ensuring one active game for memory, the game minimizes memory consumption. I recommend a RAM of 32GB for concurrent gameplay, as well as a dedicated GPU to render images on the server-side, instead of the client-side.
* **Distributed Systems and Networks**: To enable communication across platforms, the game should use RESTful APIs hosted on Windows Server. With no installation on the client-side, cross-platform compatibility can be achieved with HTML5, CSS, or JavaScript.
* **Security**: To protect user information, security must be at the forefront of development. We should configure the Windows Server with an SSL certificate to enable HTTPS, which encrypts data across the web between the server and the client. To verify users, I would implement JSON Web Token authentication and use role-based access to restrict information. In the Java scripting and SQL server queries, I would protect again SQL injection attacks. Finally, I would log event with Windows Event Logs and Security manager to monitor any sort of unusual activity. The tools identified above should be maintained and updated on a regular basis to capture security patches before hackers can infiltrate.