

The Gaming Room

# **CS 230 Project 1 Software Design**

Version 3.0

## Table of Contents

[**CS 230 Project Software Design** 1](#_heading=h.gjdgxs)

[**Table of Contents 2**](#_heading=h.30j0zll)

[**Document Revision History 2**](#_heading=h.3znysh7)

[**Executive Summary 3**](#_heading=h.2et92p0)

[**Requirements 3**](#_heading=h.tyjcwt)

[**Design Constraints 4**](#_heading=h.1t3h5sf)

[**System Architecture View 4**](#_heading=h.4d34og8)

[**Domain Model 5**](#_heading=h.2s8eyo1)

[**Evaluation 5**](#_heading=h.17dp8vu)

**Recommendations [7](#_heading=h.26in1rg)**

## [Document Revision History](#_heading=h.lnxbz9)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 1/27/2024 | Armon Wilson | Initial Release |
| 2.0 | 02/09/2024 | Armon Wilson | Development Requirements section updated. Recommendations section updated. |
| 3.0 | 03/24/2024 | Armon Wilson | Further developed and adjusted the recommendations section. |

**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](#_heading=h.35nkun2)

The Gaming Room seeks to expand its popular "Draw It or Lose It" gaming application to diverse platforms. This transformation necessitates a flexible architecture suitable for deployment across the web, desktop systems (Linux, Mac, Windows), and mobile devices (iOS and Android). Creative Technology Solutions (CTS) proposes a cross-platform solution to seamlessly deliver this new gaming experience.

## Requirements

Business Requirements:

* The application shall be designed and implemented using a web-based distributed architecture, considering aspects such as client-server communication, security, and scalability.
* The game shall support one or more teams, fostering collaborative gameplay where participants work together to guess drawings.
* Each team shall have the capability to include multiple players, engaging experience.
* The system shall enforce the uniqueness of both game and team names, preventing ambiguity and facilitating easy identification.

Technical Requirements:

* A game shall have the ability to have one or more teams involved.
* Each team shall have the capability for multiple players assigned to it.
* Game and team names shall be unique to allow users to check whether a name is in use when choosing a team name.
* Only one instance of the game shall exist in memory at any given time. This can be accomplished by creating unique identifiers for each instance of a game, team, or player.

Requirements Summary:

The application, designed for a web-based distributed architecture, emphasizes collaborative gameplay with multiple teams and players. To ensure clarity and ease of identification, unique names for games and teams shall be enforced. Additionally, only one instance of the game is permitted in memory at any given time, achieved through the implementation of unique identifiers for games, teams, and players.

## [Design Constraints](#_heading=h.1ksv4uv)

Web-based Architecture:

The application needs to be designed for a distributed environment, considering factors like client-server communication, security, and scalability.

Software Implications:

Designing the application for a distributed environment involves optimizing client-server communication, implementing robust security measures, and ensuring scalability through cloud services.

Unique Identifiers:

Each game, team, and player must have unique identifiers to distinguish instances. This constraint influences the design of classes and the usage of identifiers within the application.

Software Implications:

Requiring unique identifiers for each game, team, and player influences the application's design by necessitating the implementation of mechanisms to generate and manage distinct identifiers, impacting class structures.

## [System Architecture View](#_heading=h.44sinio)

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](#_heading=h.2jxsxqh)

This UML visualizes the class structure of the application. The Game, Team, & Player classes all inherit their id and name variables from the Entity class. The GameService class manages the instantiation of each game instance for the Game class with a “zero to many” relationship. The Game class instantiates a team for the Team class with a “zero to many” relationship. The Team class instantiates a player for the Player class with a “zero to many” relationship. The Singleton design pattern is used for each case of instantiation. Each class houses a list for their respective relationships and utilizes the Iterator design pattern to for each attempt to construct a new entity to ensure only one instance of instantiation occurs for any given object.

The ProgramDriver class contains the main() method, creating an instance of the GameService and multiple Entity objects, and then calls the SingletonTester class to verify the singleton pattern is functioning as intended.

**"The Gaming Room UML diagram. The top of the diagram is labeled as com dot gamingroom. Test boxes are placed in two layers. The first layer has three text boxes and the second layer has four of them. In the first layer, the 'ProgramDriver' textbox points to 'SingletonTester' textbox. The 'ProgramDriver' textbox contains the text 'asterisk main round brackets.' The 'SingletonTester' textbox contains the text 'asterisk testSingleton round brackets.' The arrow between these two text boxes are labeled 'open two angle brackets uses close two angle brackets'. In the second layer, there are 'GameService', 'Game', 'Team', and 'Player' text boxes. The 'GameService' textbox has texts arranged in two layers. The first layer contains games colon List open angle bracket Game close angle bracket, nextGamesId colon long, nextPlayer Id colon long, nextTeamId colon long, and service colon GameService. The second layer contains GameService round brackets, getinstance round brackets colon GameService, addGame open parenthesis name colon String close parenthesis colon Game, getGame open parenthesis id colon long close open parenthesis colon Game, getGame open open parenthesis name colon String close open parenthesis colon Game, getGameCount round brackets colon int, getNextPlayerID round brackets colon long, and getNextTeamId round brackets colon long. The 'GameService' box is connected with the 'Game' textbox with a line labeled 'zero dot dt dot asterisk'.  The 'Game' textbox also contains text in two layers. The first layers contains the text teams colon List open angle bracket Team close angle bracket. The second layer has Game open round bracket id colon long comma name colon String close parenthesis, addTeam open parenthesis name colon String close parenthesis Team, toString round brackets colon String. The 'Game' textbox is connected with the 'Team' textbox with a line labeled 'zero dot dt dot asterisk'. The 'Team' textbox also contains text in two layers. The first layers contains the text players colon List open angle bracket Player close angle bracket. The second layer has Team open parenthesis id colon long comma name colon String close parenthesis, addPlayer open parenthesis name colon String close parenthesis colon Player, and toString round brackets colon String. The 'Team' textbox is connected with the 'Player' textbox with a line labeled 'zero dot dt dot asterisk'. It contains the text Player open parenthesis id colon long comma name colon String close parenthesis and toString round brackets colon String. The 'Game', the 'Team, and the 'Player' boxes point to the 'Entity' textbox in first layer. The 'Entity' textbox contains text in two layers. The first layer has the text id colon long and name colon String. The second layer has Entity round brackets, Entity open parenthesis id colon long comma name colon String close parenthesis, getId round brackets colon long, getName round brackets colon String, toString round brackets colon String.**

## [Evaluation](#_heading=h.z337ya)

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** | Robust options (Node.js, Python). Seamless Apple integration, potentially higher licensing costs.  Mac servers are reliable and secure, best suited for environments requiring seamless integration with Apple technologies. | Highly customizable (Node.js, Python, Java). Open-source focus reduces licensing costs. Linux servers offer high customization and security but will be more complicated to implement. | Familiar environment (Node.js, Python, .NET). Potential licensing implications. Windows servers provide a user-friendly environment but may involve licensing fees. | Cloud-based backend simplifies mobile integration. Mobile devices offer a vast user base and app store distribution but face challenges due to OS and device fragmentation. |
| **Client Side** | Web technologies adaptable. Cross-platform frameworks recommended.  Considerations include expertise in macOS technologies and varying costs based on application complexity and licensing fees. | Web technologies compatible. Requires Linux-specific expertise.  Considerations involve compatibility across distributions and the need for expertise in Linux-specific APIs. | Web technologies adaptable. UI consistency variations possible.  Considerations include compatibility with different versions and potential variations in user experience across diverse devices. | Platform-specific development:  1.) Native: for best performance  2.) Cross-platform: for efficiency  Considerations involve platform-specific languages, such as Swift for iOS and Java for Android, and thorough testing on diverse devices. |
| **Development Tools** | Xcode (IDE), Swift (primary language), Objective-C (possible) | Various IDEs (VS Code, Eclipse), (C++, Python, Java) | Visual Studio (IDE), C#, .NET, and other compatible options | Android: Android Studio (IDE), Kotlin/Java  iOS: Xcode (IDE), Swift |

## 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:**

The best approach for "Draw It or Lose It" is a hybrid solution. A cloud-based backend (like AWS or Azure) using Node.js or Python will ensure scalability and work seamlessly with any device. For the frontend, React Native or Flutter provide a "write once, deploy everywhere" experience, guaranteeing consistency across a variety of platforms.

1. **Operating Systems Architectures:**

The choice of a cloud-based backend (AWS, Azure, etc.) offers significant flexibility in operating system selection. Provisioning virtual machines (VMs) on-demand allows us to tailor the environment for "Draw It or Lose It." A Linux distribution is ideal due to its security, cost-effectiveness, and customizability.

Cross-platform frontend frameworks like React Native or Flutter play a crucial role in abstracting away the complexities of different operating systems. They use pre-built, OS-specific components (for UI elements, file access, etc.) and handle the low-level interactions under the hood. This significantly streamlines deployment, as a single codebase can adapt to iOS, Android, Windows, and other platforms without major rewrites.

1. **Storage Management:**

Successful storage management for "Draw It or Lose It" hinges on carefully determining the necessary capacity for its image library. First, accurately estimate the total storage needed by considering the individual image file sizes and the overall image count. Don't forget to factor in other essential game data like player usernames, passwords, past scores, and potential future features as these will expand storage needs.

While the utilization of a local storage solution is one option, exploring a cloud-based solution merits consideration. The cloud offers easy synchronization across devices, greater flexibility, and the potential for "pay-as-you-go" cost savings, minimizing the need for up-front hardware investments. Additionally, optimizing storage usage through data compression and efficient file organization will help keep the application streamlined, regardless of whether local or cloud storage is chosen.

With the assumption of 200+ HD image files at 8Mb each and accounting for operating system, program files, and user profile storage, the recommendation for minimum operating storage will be 20Gb. This ensures that there is ample storage space for the 1.6Gb needed for the 200 image files (estimated 8Mb each), reserving additional capacity for operating system, user profile database, and program files.

1. **Memory Management:**

For image handling, employ techniques like downsampling (loading lower-resolution versions when appropriate) and lazy loading (only loading images as they scroll into view) to minimize memory footprint. Caching frequently used images in the frontend is essential. Node.js's garbage collector periodically identifies unused objects, reclaiming their memory. React Native and Flutter have their own mechanisms for UI component lifecycle management, freeing up memory associated with views that are no longer on screen. Profiling tools in Node.js and within developer consoles for these frameworks help pinpoint memory leaks or inefficient image handling.

1. **Distributed Systems and Networks:**

Employing a RESTful API over HTTP facilitates real-time interactions between different components of the distributed system. Designing for reliability involves implementing mechanisms for graceful handling of network disruptions and maintaining data consistency through synchronization techniques. These measures ensure seamless communication and consistent gameplay experience across various platforms.

1. **Security:**

Implement OAuth 2.0 for robust authentication and authorization, encrypting data in transit with HTTPS. Consider database-level encryption for sensitive data at rest. Rigorously sanitize user input on the backend to prevent SQL injection and XSS attacks. Leverage Linux's granular permissions and Iptables firewall for advanced control. Mitigate frontend vulnerabilities, ensure secure API communication, and promote user awareness for comprehensive protection.