

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 3.0

## Table of Contents

[**CS 230 Project Software Design Template** 1](#_Toc115077317)

[**Table of Contents 2**](#_Toc115077318)

[**Document Revision History 2**](#_Toc115077319)

[**Executive Summary 3**](#_Toc115077320)

[**Requirements 3**](#_Toc115077321)

[**Design Constraints 3**](#_Toc115077322)

[**System Architecture View 5**](#_Toc115077323)

[**Domain Model 5**](#_Toc115077324)

[**Evaluation 7**](#_Toc115077325)

[**Recommendations 9**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 9/22/24 | Christopher Davidson | Initial draft for web based game application design for The Gaming Room. |
| 1.1 | 9/24/24 | Christopher Davidson | Incorporated feedback into the executive summary. |
| 1.2 | 10/3/24 | Christopher Davidson | Incorporated feedback into the executive summary and Design Constraints. |
| 2.0 | 10/6/24 | Christopher Davidson | Added the Evaluation section |
| 3.0 | 10/19/24 | Christopher Davidson | Added the Recommended 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](#_sbfa50wo7nsh)

The Gaming Room seeks to develop a web based version of its current Android only game, Draw It or Lose It. The goal is to expand accessibility across multiple platforms, ensuring the game can serve multiple clients at once. The client has limited expertise in setting up a development environment and has contracted CTS for this purpose. The game will feature an extensive library of stock images that will serve as clues during gameplay, with images rendering at a steady rate for ideal performance. The game must accommodate multiple teams, each consisting of multiple players, while ensuring unique team and game names to avoid conflicts. The system must guarantee that only one game instance is running in memory at any given time, requiring the implementation of unique identifiers for teams, players, and games.

## Requirements

Business Requirements:

* The game must be developed as a web based application that supports multiple platforms.
* The game must allow multiple teams to participate simultaneously, with each team consisting of multiple players.
* Both game names and team names must be unique, ensuring no duplication when users create teams or start games.

Technical Requirements:

* Only one instance of the game must exist in memory at any given time. This will be achieved through the use of the Singleton pattern in the GameService class, which ensures there is only one active game service.
* Each game, team, and player must be assigned a unique identifier to prevent duplication. The use of static ID generators will ensure all entities are uniquely identifiable.
* The game must support real time interaction and synchronization across different teams and players during gameplay, ensuring a seamless experience on all supported platforms.

## [Design Constraints](#_2et92p0)

As the application transitions from a mobile only platform to a multi platform, web based game, several key technical and architectural constraints must be considered. These constraints directly impact the system's functionality, performance, and scalability and should be derived from the software requirements and the transition challenges:

* Cross Platform Compatibility
  + Challenge:
    - The game, originally designed for Android, now needs to operate on multiple platforms, including Windows, macOS, Linux, iOS, and Android. Each operating system introduces unique constraints, such as differences in performance capabilities, supported technologies, and rendering mechanisms.
  + Impact:
    - Maintaining consistent behavior across platforms will likely require careful design decisions to minimize dependencies on platform specific features. These decisions will increase development complexity and effort, as the system must support multiple environments.
* Real Time Synchronization
  + Challenge:
    - The game requires real time interactions across multiple players and teams, necessitating synchronization of data with minimal latency. In a distributed web based environment, players will connect from different geographic locations, resulting in varying network conditions that can cause latency or data synchronization issues.
  + Impact:
    - Without careful management of network latency, user experience may suffer due to lag or inconsistent gameplay. The architecture must be designed to handle these potential discrepancies efficiently to ensure timely communication between players.
* Security and Authentication
  + Challenge:
    - Transitioning from a mobile application to a web based environment increases the attack surface for security vulnerabilities. Unlike mobile applications, which often rely on device level security, a web based game requires robust server side security measures.
  + Impact:
    - This increased exposure introduces the need for more comprehensive user authentication, data encryption, and secure access control, adding complexity to the system's architecture and design. Handling sensitive user data securely in a distributed environment is critical to maintaining trust and preventing security breaches.
* Memory and Resource Management
  + Challenge:
    - The game will need to load and render a large library of stock images in real time as part of the gameplay. As the game scales to support multiple players and teams, the demand for memory will increase, especially for managing concurrent instances and efficiently rendering images across various platforms.
  + Impact:
    - Improper memory management could lead to performance degradation, with slower image rendering and gameplay delays. The system must be designed to manage resources efficiently, ensuring smooth and uninterrupted gameplay even under high load conditions.
* Scalability for Concurrent Users
  + Challenge:
    - The game must support multiple teams and players simultaneously, which introduces complexity in handling concurrent user connections and maintaining performance at scale. The architecture, originally designed for a single client mobile application, now faces the challenge of scaling to accommodate multiple users.
  + Impact:
    - Ensuring that the game can scale effectively without degrading performance requires careful management of server resources and network bandwidth. Failure to address scalability could result in bottlenecks, negatively affecting the user experience during peak usage times.
* Data Storage and Management
  + Challenge:
    - The transition to a web based platform will significantly increase storage needs, as the game must store and manage a larger volume of game data, player profiles, and stock images. The system must be able to retrieve data quickly and reliably, especially during real time gameplay.
  + Impact:
    - Delays in accessing stored data, such as game assets, could lead to interruptions in gameplay and a poor user experience. The system must balance storage capacity and retrieval speed, ensuring that data is available in real time without sacrificing performance.
* Administrative Complexity
  + Challenge:
    - The game requires an administrative interface for managing users, teams, and game instances. This interface must be both secure and user friendly, capable of handling a large amount of data while maintaining ease of use for administrators.
  + Impact:
    - The complexity of managing multiple teams, players, and game instances increases as the game scales. A robust and scalable administrative interface will be crucial for efficient game management and will add complexity to the overall system architecture.

## [System Architecture View](#_ilbxbyevv6b6)

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](#_8h2ehzxfam4o)

The UML diagram for Draw It or Lose It provides a clear view of the relationships between different game components. It incorporates key object oriented programming (OOP) principles and software design patterns to address the client’s requirements.

* Entity Class:
  + This serves as a base class for Game, Team, and Player, and it contains common attributes like id and name. By having these attributes centralized in the Entity class, it ensures that all game components can be uniquely identified, which is essential for meeting the client’s need to maintain uniqueness across all game entities.
* GameService Class:
  + This class uses the Singleton pattern to make sure that only one instance of the game service can exist at any time. Static variables are used to assign unique IDs to games, teams, and players, which ensures there are no duplicates. The Iterator pattern is also used to ensure that new game or team names are checked for uniqueness before being created.
* Game Class:
  + This class inherits from Entity and is responsible for managing the list of teams within a game. It ensures that team names are unique by checking for duplicates before new teams are added.
* Team Class:
  + Like Game, the Team class inherits from Entity and manages the list of players. It ensures that each player’s name is unique within a team, helping to avoid conflicts during gameplay.
* Player Class:
  + Inherits from Entity and represents individual players. This ensures each player has a unique ID and name.

Object Oriented Principles:

* Inheritance:
  + The Entity class serves as a superclass for Game, Team, and Player, which helps streamline the code by centralizing shared attributes. This promotes code reuse and helps to keep the design simpler by reducing redundancy across different classes.
* Encapsulation:
  + Each class manages its own data and behavior. For example, Game manages its list of teams, and Team handles its players. This way, each class controls its internal state, preventing external modifications and helping to keep the game’s structure stable.
* Singleton Pattern:
  + In GameService, the Singleton pattern is used to ensure only one instance of the game service exists at any time. This helps with memory management and ensures that there aren’t multiple game services running simultaneously, which aligns with the client’s memory related requirements.
* Iterator Pattern:
  + The Iterator pattern is used in the GameService class to handle the process of checking for duplicate game or team names. This allows the system to efficiently iterate over existing entities and maintain the uniqueness of names.

These OOP principles and design patterns are applied throughout the game’s architecture to meet the client’s requirements, ensuring that game entities remain distinct, and that the application can function smoothly across multiple platforms.

**"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](#_2o15spng8stw)

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 is Unix based, making it secure and stable but not commonly used for large scale web hosting. It integrates poorly with cloud providers (compared to Linux) and has high hardware and licensing costs. * macOS provides strong development tools, but deploying scalable apps on macOS servers is inefficient. * Suitable for niche, smaller scale server hosting solutions or development environments, not large scale production environments. | * Linux is the most popular server platform due to its scalability, flexibility, and cost effectiveness. * It integrates well with containerization technologies like Docker and Kubernetes, making it ideal for distributed applications. * Linux is an excellent choice for enterprise environments that need robust, customizable server solutions. Expertise in configuration and management is required, making it a great fit for large scale web hosting. | * Windows Server is commonly used in enterprise environments, integrating well with Microsoft services like IIS, .NET, and Azure. * It supports enterprise level web applications and integrates with an Active Directory for identity management. * Windows is a heavy resource compared to Linux and does not scale as efficiently. Its licensing costs are higher than Linux, but it offers strong enterprise support and seamless integration with Microsoft’s ecosystem. | * Mobile platforms don’t function as web hosts but rely on real time communication with backend servers. * The server must handle network variability, latency, and packet loss to ensure smooth gameplay for users. * Mobile servers must be optimized for handling latency and varying bandwidth to maintain a seamless user experience across different mobile devices. |
| Client Side | * macOS is a popular consumer OS, especially for creatives. It is essential to support Safari as a browser. * Performance tuning for Safari is necessary, as it has unique rendering quirks. Compatibility testing between Safari and other browsers like Chrome is crucial. * Macs have high hardware performance, so optimizing for smooth rendering on Safari while maintaining cross browser compatibility is necessary. | * Linux is less common as a consumer platform but supports major browsers like Firefox and Chrome. * It's typically used by tech enthusiast users and developers, so ensuring cross platform functionality and compatibility with Firefox and Chrome is essential. * Some additional testing may be required for font rendering and browser inconsistencies on Linux systems. | * Windows dominates the desktop market, making it essential to ensure compatibility with Chrome, Edge, and Firefox. * Windows users span various devices, so web apps must perform well across all major browsers. Special attention should be given to touch integration for hybrid devices like Surface Pro. * The Edge browser, now based on Chromium, simplifies cross browser testing. | * The mobile experience requires responsive design with a focus on touch interactions. The client must be optimized for performance consistency across iOS and Android. * Cross browser compatibility testing is critical for iOS (Safari) and Android (Chrome) to ensure parity with desktop experience. * Apps must be optimized for varying network conditions, especially 4G and slower connections. |
| Development Tools | * macOS provides excellent development tools, including Xcode for iOS/macOS development and VS Code for cross platform apps. * macOS locks developers into using specific tools like Swift for iOS/macOS development, which may limit flexibility for cross platform applications. * CI/CD pipelines with tools like Jenkins or CircleCI are possible but may require manual setup. | * Linux offers a flexible development environment with tools like VS Code, IntelliJ IDEA, and Eclipse. * Linux supports a variety of programming languages and integrates well with Docker, making it ideal for containerized applications. * Its command line interface (CLI) allows for efficient CI/CD pipeline management, perfect for scalable applications. Automated testing and deployment can be easily handled through open source tools. | * Windows provides strong development support through Visual Studio and seamless integration with Azure DevOps. * The Windows Subsystem for Linux (WSL) allows developers to run Linux distributions on Windows, offering cross platform development flexibility. * Licensing costs for development tools like Visual Studio can be significant but are justified in enterprise environments. CI/CD pipelines are well supported through Azure DevOps. | * Mobile development requires platform specific tools like Xcode for iOS and Android Studio for Android. * Cross platform frameworks like React Native and Flutter streamline mobile development by allowing code reuse across platforms. * Testing tools like Appium are essential for automated UI testing across multiple devices, and performance testing for different network conditions is critical for ensuring a smooth user experience. |

## 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**:
   * Recommendation:
     + For "Draw It or Lose It," the recommended operating platform is a Linux-based server hosted on Amazon Web Services (AWS) or Microsoft Azure. Both platforms offer robust cloud services, providing the flexibility and scalability needed to support the game’s expansion across different platforms, including desktop, mobile, and web.
   * Reasoning:
     + The decision to recommend Linux is driven by its widespread adoption in cloud environments and its proven reliability in handling high-demand server operations. Linux is an open-source operating system, which means it can be customized to meet specific needs without the licensing costs associated with other operating systems. Its strong security model and efficient resource management make it ideal for server environments where uptime and stability are critical. According to OpenSource.com (2018), Linux’s ability to handle a high volume of tasks efficiently and securely is one of the main reasons for its dominance in server environments.
     + AWS and Azure are top choices for hosting due to their ability to offer on-demand scaling. This is particularly important for a game like "Draw It or Lose It," which will likely experience fluctuating traffic. These platforms enable auto-scaling, where server resources can expand or contract based on user demand, ensuring that performance remains optimal without incurring unnecessary costs. AWS’s EC2 instances and Azure’s Virtual Machines support a variety of operating systems, but Linux stands out because of its cost-effectiveness and performance in cloud-native environments. Furthermore, both AWS and Azure provide extensive support for microservices architectures, which will allow the game to be developed and deployed in a more modular, scalable way (Chapel, 2021).​​​.
2. **Operating Systems Architectures**:
   * Recommendation:
     + The best architecture to support the game is a microservices based architecture running on a Linux operating system. This architecture will allow for better scalability, flexibility, and maintainability, which are all crucial for a game like "Draw It or Lose It" as it expands.
   * Reasoning:
     + The choice of a microservices architecture is driven by its modular nature, where the application is broken down into smaller, independent services. Each service can be developed, deployed, and scaled independently. This ensures that if one part of the game needs to be updated or fixed, the entire game does not need to go offline. Microservices also support distributed systems, which are ideal for a cloud-based environment like AWS or Azure. By deploying different components of the game, such as user authentication, game logic, and data storage as independent services, we can ensure that each part of the system can scale based on its own demand without affecting the overall system (Arsov, 2021).​​
     + Linux’s architecture makes it a strong fit for managing microservices due to its compatibility with containerization tools like Docker. Containers allow each microservice to run in its own environment, independent of the others, which increases reliability and simplifies deployment. This approach also supports horizontal scaling, where multiple instances of each service can be run simultaneously to handle increased traffic. Given that Linux is highly customizable, we can optimize it for specific tasks related to the game, such as managing network traffic or handling large amounts of player data.
3. **Storage Management**:
   * Recommendation:
     + The game should use cloud-based storage through Amazon S3 or Azure Blob Storage. These options provide highly scalable, secure, and cost-effective solutions for storing the game's assets, including player data, images, and game history.
   * Reasoning:
     + Both Amazon S3 and Azure Blob Storage are designed to store large amounts of unstructured data, making them ideal for the type of data "Draw It or Lose It" will generate. These services provide near-unlimited scalability, ensuring that as the game’s user base grows, the storage can expand without the need for costly hardware investments. Cloud storage solutions also provide high availability, which is essential for minimizing downtime and ensuring that users can always access their game data​​​.
     + From a cost perspective, both AWS and Azure offer flexible pricing models, allowing The Gaming Room to pay only for the storage they actually use. Chapel (2021) notes that cloud storage services like S3 and Blob Storage are especially cost-effective for applications with variable storage needs, as they automatically scale without manual intervention. This reduces the operational overhead of managing physical servers and ensures that the game can scale its storage needs efficiently. Both services also include advanced security features, such as encryption for data at rest and in transit, ensuring that sensitive player data is protected from potential breaches.
4. **Memory Management**:  
   * Recommendation:
     + The recommended cloud platforms (AWS and Azure) use dynamic memory allocation techniques, including paging and virtual memory, which will support efficient memory management for "Draw It or Lose It.”
   * Reasoning:
     + Memory management is crucial for ensuring that the game operates smoothly, even during periods of high demand. Both AWS and Azure allow for dynamic memory allocation, where resources are allocated based on the system's current needs. This ensures that memory is not wasted on idle processes, and that there is always enough available to handle spikes in user activity. Virtual memory allows the game to operate as though it has more memory than is physically available, by temporarily storing less frequently used data on disk. This is essential for a game that involves real-time interaction and large datasets, such as images and player profiles (Silberschatz, Galvin, & Gagne, n.d.}.
     + Paging further supports efficient memory management by dividing memory into fixed-sized pages and only loading the necessary pages into memory. This minimizes the amount of memory needed at any given time and reduces the risk of system overload. Additionally, Linux’s architecture supports advanced memory management techniques, making it an ideal choice for hosting the game. Linux’s support for containerization also means that each service can be allocated its own memory, ensuring that no one part of the system consumes more than its share of resources.
5. **Distributed Systems and Networks**:  
   * Recommendation:
     + The game should use a RESTful API architecture to facilitate communication between different platforms. This architecture will support distributed systems, which are necessary for the game to function across various operating systems and devices.
   * Reasoning:
     + A RESTful API provides a standardized way for different parts of the system to communicate, regardless of the platform they are running on. This is essential for a game like "Draw It or Lose It," which needs to operate on multiple platforms, including web, mobile, and desktop. RESTful APIs use HTTP requests, making them platform-agnostic and easy to implement in distributed environments. By using this architecture, the game can ensure that data is transferred efficiently between the server and the client devices.
     + The choice of a distributed system is driven by the need to ensure high availability and fault tolerance. By distributing the game’s services across multiple servers in different regions, we can ensure that if one server goes down, others can take over without disrupting the player experience. This also improves the game’s ability to handle large numbers of concurrent users, as traffic can be balanced across multiple servers. AWS and Azure offer load balancing services that can automatically distribute traffic based on server load, ensuring that no one server is overwhelmed (Silberschatz et al., n.d.).
6. **Security**:
   * Recommendation:
     + TLS encryption should be implemented for securing data in transit, with AWS Key Management Service (KMS) or Azure Key Vault used for securing data at rest. Multi-factor authentication (MFA) should also be implemented to protect user accounts.
   * Reasoning:
     + Security is a critical concern for any online game, especially one that involves user data. TLS encryption ensures that all data transmitted between the client and server is encrypted, protecting it from potential eavesdropping or tampering. This is particularly important for securing sensitive data, such as login credentials and player information. In addition to securing data in transit, it is essential to secure data at rest. Both AWS KMS and Azure Key Vault provide tools for managing encryption keys, ensuring that stored data remains encrypted and cannot be accessed without the proper keys (Silberschatz et al., n.d.).
     + Multi-factor authentication (MFA) adds an additional layer of security by requiring users to provide two forms of verification before accessing their accounts. This greatly reduces the risk of unauthorized access, as even if a password is compromised, the attacker would still need access to the second form of verification. Both AWS and Azure provide built-in MFA options, which can be easily integrated into the game’s authentication system.

Reference List:

Arsov, K. (2021, May 18). Microservices vs. SOA — Is There Any Difference at All? Medium. <https://medium.com/microtica/microservices-vs-soa-is-there-any-difference-at-all-2a1e3b66e1be>

Chapel, J. (2021, December 10). Cloud Storage Cost Comparison: AWS vs. Azure vs. Google. Medium. <https://jaychapel.medium.com/cloud-storage-cost-comparison-aws-vs-azure-vs-google-844dfff3d324>

OpenSource.com. (2018, May 31). What is a Linux Server and Why Does Your Business Need One? - Linux.com. Linux.com. https://www.linux.com/news/what-linux-server-and-why-does-your-business-need-one-0/

Silberschatz, A., Galvin, P. B., & Gagne, G. (n.d.). Operating System Concepts, 8th Edition. O’Reilly Online Learning. <https://learning.oreilly.com/library/view/operating-system-concepts/9780470128725/silb_9780470128725_oeb_c14_r1.html#h1>

Writers, S. (2024, August 9). Top 14 Best Coding Languages for Computer Programming. ComputerScience.org. https://www.computerscience.org/resources/computer-programming-languages/