

# Win, Lose or Draw

# **CS 230 Project Software Design Template**

Version 1.0

## 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 3**](#_Toc115077323)

[**Domain Model 3**](#_Toc115077324)

[**Evaluation 4**](#_Toc115077325)

[**Recommendations 5**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 03/24/2024 | Devin Wheeler | Initial brief |
| 1.1 | 04/07/2024 | Devin Wheeler | Operating Platforms |
| 1.2 | 4/21/2024 | Devin Wheeler | Recommendations |

## [Executive Summary](#_sbfa50wo7nsh)

Creative Technology Solutions is tasked with developing a web-based version of "Draw It or Lose It”. This game, inspired by the classic game show "Win, Lose or Draw", involves teams guessing words or phrases from stock drawings. The challenge lies in creating a seamless, engaging experience for players across various devices, ensuring real-time interactions and a user-friendly interface. Our solution involves using modern web technologies to build a responsive, scalable web application that supports multiple users in real-time, ensuring a consistent gaming experience across all platforms.

## Requirements

*<* Please note: While this section is not being assessed, it will support your outline of the design constraints below. *In your summary, identify each of the client’s business and technical requirements in a clear and concise manner.>*

## [Design Constraints](#_2et92p0)

* Cross-Platform Compatibility: The game must offer consistent experience across various devices and screen sizes, requiring a responsive design and testing on multiple devices.
* Real-Time Interaction: Implementing real-time game mechanics, such as drawing display and guessing, meaning the use of certain technologies to minimize latency.
* Scalability: As player numbers grow, the system must scale efficiently. This involves choosing a back-end architecture that supports scaling and managing increased data volume and user connections.
* Data Integrity and Security: Ensuring that game and team names are unique and managing user data securely.
* State Management: Given that only one instance of the game can exist in memory at any given time, effective state management is critical to prevent data conflicts and ensure a smooth user experience.

## [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)

* ProgramDriver: This class contains the main() method, serving as the use point for the application. It uses the SingletonTester class.
* SingletonTester: This class has a method testSingleton() to test the singleton pattern implementation in the GameService class, ensuring only one instance of GameService can exist.
* Entity Class: Serves as a base class with common properties - id (long) and name (String) and methods for other entity classes like Game, Team, and Player.

**Subclasses of Entity:**

* GameService: A singleton class responsible for managing games. It includes properties for managing lists of games and generating unique IDs. The getInstance() method enforces the singleton pattern.
* Game: Represents individual games, containing a list of teams. Methods include adding teams and converting game information to a string format.
* Team: Represents teams within a game, holding a list of players. It includes functionality to add players and convert team information to a string.
* Player: Represents individual players, with functionality to return player information as a string.

**Object-Oriented Principles Demonstrated:**

* Encapsulation: Properties and methods within each class are encapsulated, with access modifiers controlling them.
* Inheritance: Demonstrated by the Game, Team, and Player classes extending the Entity class, promoting code reuse and reducing redundancy.
* Aggregation: Shows relationships, such as a game having teams and a team having players, allowing for complex structures without strict ownership.
* Singleton Pattern: Implemented in the GameService class to ensure only one instance of this class can exist within the application, controlling access to game management functionalities.

**"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.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Limited hardware choice (must use Apple hardware). Not as widely used in server environments, leading to difficulty finding hosting providers if you don’t want to buy the hardware.  **Discontinued as a stand-alone product, limiting its future use and support.** | Open source, widely used for servers. Supports a wide range of server software including Apache, Nginx, and Node.js for web-based applications.  Customizable, community support, and typically lower resource requirements than the other options. No licensing costs for the operating system which can significantly reduce server costs if self-hosting. Otherwise, lots of hosting options like Azure or AWS.  No nice interface to interact with. Can require more experience so set up | Commercial operating system designed for server use.  Integrated with Microsoft products and services. Easier for admins familiar with Windows. Direct support from Microsoft.  Licensing costs can be significant. Usually costs per user. Generally, requires more resources than Linux. | Mobile devices are not designed for high-performance computing, making them bad for hosting applications with high traffic or intensive processing needs. |
| **Client Side** | Development can be more expensive due to the high cost of Apple hardware and software requirements.  MacOS is written in SWIFT and Objective C which are lesser know languages so developers will const more.  Macs only make up 16.23% of computer users compared to Windows at 73% | Generally lower costs due to a wide range of free and open-source software tools available.  Linux users only make up around 3.77% of computer Oss. This may not be worth the cost to develop. | Licensing costs for Windows OS and some development tools. There are many free tools also available.  Windows uses C# and .NET which are common to find developers for.  Windows makes up 73% of all computers, making it the best option for a ROI. | Higher costs due to the need for multiple devices for testing, and fees for app store submissions.  Supporting multiple screen sizes and OS versions can increase development and testing time.  Requires knowledge in mobile-specific design principles, cross-platform development tools, or native development for iOS (like SWIFT) and Android. |
| **Development Tools** | Swift, Objective-C for native applications.  Xcode for native macOS and iOS applications. This is around 100% per year per developer.  GitHub or Git for version controls, free with paid upgrades. | Python, Java, C/C++ for native applications  Python ides can be found for free. Some good examples are visual studio code and PyCharm. As for C/C++, Eclipse can be a good option or visual studio.  GitHub or Git for version controls, free with paid upgrades. | C#, .NET for native applications.  Visual Studio for a comprehensive development environment.  Visual Studio Code for a lighter option.  GitHub or Git for version controls, free with paid upgrades. | Swift for iOS, Kotlin or Java for Android on their native apps.  Xcode for iOS, Android Studio for Android.  React Native or Flutter for cross-platform mobile app development.  GitHub or Git for version controls, free with paid upgrades. |

## 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**: Linux is widely recognized for its stability, security, and performance in server environments, making it a preferred choice for hosting web servers and applications. Very easy to host yourself or find hosting services such as AWS or Azure.
2. **Operating Systems Architectures**: Linux's architecture is great for hosting a backend, designed for multitasking and multi-user operations, making it highly efficient in handling multiple requests — a critical feature for a gaming application expected to serve thousands of players. Linux servers can run on various hardware platforms, offering flexibility in deployment options. Its design allows for custom configurations tailored to specific needs, enhancing performance and security.
3. **Storage Management:** For storage management, going for self-hosting means the upfront purchase of either Hard Disk Drives (HDDs) or Solid-State Drives (SSDs). HDDs offer a cost-effective solution with larger storage capacity but lower access speeds, while SSDs provide faster data access and improved performance at a higher cost. Choosing a cloud service provider like AWS allows for scalable storage solutions, where you can dynamically adjust capacity based on demand, leading to a more flexible and cost-effective payment structure.
4. **Memory Management**: Linux uses advanced memory management techniques such as virtual memory, demand paging, and swap management, meaning efficient use of system resources. To further optimize this, we can implement a caching strategy to store frequently accessed data in memory, reducing latency and load times. For "Draw It or Lose It" this means that the server can handle large numbers of connections and data exchanges with optimal memory usage, lowering the risk of slowdowns or crashes due to memory leaks or overload.
5. **Distributed Systems and Networks**: For "Draw It or Lose It," using a distributed system architecture supported by cloud services means robust communication and seamless gameplay across different platforms. Using cloud platforms such as AWS or Azure allows for dynamic scaling to efficiently manage game loads, ensuring smooth performance even during peak usage. These services employ microservices architecture, allowing for modular development and easy updates. Cloud providers offer failover and redundancy features. So, if a server instance fails, the system reroutes traffic to operational servers, minimizing downtime. Additionally, Content Delivery Networks (CDNs) are used to cache static game content at edge locations, reducing latency for a wide player base. Real-time data synchronization across devices is achieved through technologies like WebSocket, ensuring consistent game states. This approach not only enhances the gaming experience by maintaining high availability and reliability but also ensures data integrity and security across the network.
6. **Security**: To safeguard user information across different platforms, using encryption is a must. Utilizing Transport Layer Security (TLS) encrypts data in transit, ensuring secure communication between clients and servers. For data at rest, using the cloud provider's encryption services protects stored information. Additionally, adopting a principle of least privilege across the system minimizes access to sensitive data, while regular security audits and compliance checks help identify and fix potential vulnerabilities, maintaining a secure environment for users' data across all platforms.