

<Name-of-Software-Application>

# **CS 230 Project Software Design Template**

Version 1.0

## Table of Contents

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

[**Table of Contents 2**](#_30j0zll)

[**Document Revision History 2**](#_3znysh7)

[**Executive Summary 3**](#_2et92p0)

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

[**Design Constraints 3**](#_1t3h5sf)

[**System Architecture View 3**](#_4d34og8)

[**Domain Model 3**](#_2s8eyo1)

[**Evaluation 4**](#_17dp8vu)

[**Recommendations 5**](#_26in1rg)

## [Document Revision History](#_lnxbz9)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 09/20/24 | Jade-Pineda | Initial Commit |

**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](#_35nkun2)

The Gaming Room, a client of Creative Technology Solutions (CTS), aims to expand the scope of their Android-based game *Draw It or Lose It* to a web-based platform that supports multiple operating systems and devices. The primary challenge is to create a scalable, efficient solution that maintains the core functionality of the game while allowing seamless integration across multiple platforms, including web, mobile, and desktop environments.

The key objective of this software design is to implement a flexible and modular solution that meets the client’s need for an extensible game application. The design will utilize object-oriented principles, such as inheritance and encapsulation, to create a clear and efficient structure. The Singleton design pattern will be implemented in the GameService class to ensure that only one instance of the game runs in memory at any given time, fulfilling the requirement for uniqueness and preventing duplication. In addition, we will use the Iterator pattern to efficiently manage the addition and retrieval of games, teams, and players, ensuring that names remain unique and easily manageable.

By centralizing game management, the proposed solution will not only meet the immediate technical requirements but also offer a solid foundation for future enhancements, such as additional gameplay features and support for more platforms. This strategic approach allows The Gaming Room to expand its player base while maintaining control over game instances and preventing any performance bottlenecks. With scalability in mind, this design provides the necessary flexibility to support future business growth.

## 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](#_1ksv4uv)

Developing a web-based version of *Draw It or Lose It* introduces several design constraints that need careful consideration to ensure smooth and efficient application performance across different platforms.

1. **Scalability**: The game must support multiple users simultaneously without overwhelming server resources. This requires a distributed architecture where game instances are handled efficiently, preventing overload during peak times. The use of Singleton pattern in the GameService ensures that only one instance of the game exists, which aids in managing resources effectively across various environments.
2. **Platform Compatibility**: Since the game will operate across multiple platforms—web, mobile, and desktop—the software must be designed in a platform-agnostic manner. This includes addressing differences in operating systems like Linux, Windows, macOS, and mobile devices, ensuring consistent user experience and performance across platforms. Each platform may also have different hardware limitations, which will impact memory and resource allocation.
3. **Latency and Network Considerations**: Given that the game will involve multiple users interacting in real-time, latency is a major concern. A well-designed network communication layer is critical to ensure that game state updates, guesses, and team interactions are transmitted with minimal delay. This requires an efficient distributed system and robust error-handling mechanisms to manage potential network issues such as packet loss or timeouts.
4. **Data Consistency**: The application must ensure that all players receive the same updates in real time, regardless of the platform they are using. This requires synchronization across the distributed environment, which can become challenging as the game scales up in terms of the number of players or teams involved.
5. **Security**: Protecting user data is critical, especially when dealing with a multi-platform system. Each platform has different security vulnerabilities that must be addressed. Encryption of user data, secure transmission protocols, and protection against unauthorized access are essential to maintaining the integrity and security of the game.

These constraints not only influence the architecture but also demand specific strategies for performance optimization, ensuring cross-platform compatibility, and maintaining a consistent user experience across all environments.

## [System Architecture View](#_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](#_2jxsxqh)

The UML diagram for *The Gaming Room* provides a clear representation of the relationships between the different entities that make up the game’s architecture. At the core of the design is the Entity class, which serves as a base class from which Game, Team, and Player inherit. This use of inheritance demonstrates the key object-oriented programming principle of **reusability**, as common attributes like id and name are centralized in the base class, avoiding redundancy across subclasses.

* **Game Class**: This class holds a list of Team objects and inherits the common id and name attributes from the Entity class. The relationship between the Game and Team classes is a one-to-many relationship, where a game can have multiple teams, but each team belongs to only one game. This ensures proper organization and encapsulation of team-related information within each game instance.
* **Team Class**: Similarly, the Team class holds a list of Player objects, showcasing another one-to-many relationship. This reflects the structure of the game, where each team is made up of multiple players, but each player belongs to a specific team. By inheriting from Entity, the Team class efficiently reuses the id and name attributes, simplifying the model and avoiding the need for repeated code.
* **Player Class**: The Player class inherits from Entity as well, meaning each player has a unique id and name. The design ensures that players are uniquely identifiable, which is critical for team management and game interactions.
* **GameService Class**: The GameService class plays a pivotal role as it manages the creation and retrieval of game, team, and player objects. Using the **Singleton** design pattern, GameService ensures that only one instance of the service exists, maintaining data consistency and avoiding conflicts in game state management. The **Iterator** pattern is employed in methods like addGame() and getGame() to traverse lists of games and teams, checking for uniqueness in names and ensuring efficient management of entities.

In conclusion, the UML diagram encapsulates essential object-oriented principles like **inheritance**, **encapsulation**, and **design patterns** (Singleton and Iterator), ensuring that the game’s structure is scalable, maintainable, and efficient.

**"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](#_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** | **Characteristics**: Mac servers are built on Unix, making them stable and secure for hosting web-based applications. Mac systems can run web servers like Apache or Nginx.  **Advantages**: The Unix-based foundation provides a solid, reliable platform. Mac OS also has strong integration with other Apple services, such as iCloud.  **Weaknesses**: Limited enterprise support compared to Linux. The hardware is more expensive, and there’s a smaller developer community for server use. Licensing costs can be high since you need Apple hardware to run macOS. | **Characteristics**: Linux is an open-source platform that is highly customizable and commonly used for hosting web-based applications due to its stability, security, and support for a wide range of technologies (e.g., Apache, Nginx, Docker).  **Advantages**: Free and open-source, with strong community support. It offers flexibility and is commonly used for enterprise-level servers. Linux is highly efficient and scalable.  **Weaknesses**: Requires more technical expertise to set up and maintain. The variety of Linux distributions can lead to compatibility issues if not carefully managed. | **Characteristics**: Windows Server is a popular platform for hosting web applications, especially those built with Microsoft technologies like .NET.  **Advantages**: Easy to integrate with Microsoft’s development stack (e.g., .NET, ASP.NET), and has enterprise-level support. Widely used in many industries.  **Weaknesses**: Licensing costs can be very high, and it’s more resource-intensive compared to Linux. Less commonly used for large-scale, open-source web applications. | **Characteristics**: Mobile platforms like iOS and Android are not typically used for hosting servers, but mobile clients interact with the server to fetch data.  **Advantages**: The server doesn’t run on mobile devices but must be accessible by mobile apps through REST APIs.  **Weaknesses**: Mobile devices have limited resources and are not suited for server hosting; instead, the focus is on client-server communication. |
| **Client Side** | **Development Considerations**: Applications must be compatible with Safari and other macOS browsers. Developers should focus on making the application responsive and optimized for web usage on Mac desktops and laptops.  **Cost/Time/Expertise**: Moderate costs, as web development tools for Mac are readily available. Expertise in macOS and browser compatibility is essential. | **Development Considerations**: Ensure compatibility with web browsers such as Firefox and Chrome on Linux-based systems. Linux desktop users are more tech-savvy, so web apps should perform well across different distributions.  **Cost/Time/Expertise**: Minimal cost for development since Linux is open-source. Expertise is required to ensure that web applications run smoothly across different Linux distributions. | **Development Considerations**: The application must be compatible with Internet Explorer, Edge, Chrome, and Firefox. Ensure proper functioning across various Windows versions (7, 10, 11).  **Cost/Time/Expertise**: Moderate to high cost, depending on the tools used. Expertise in Windows web browsers is required for compatibility. | **Development Considerations**: The application must be responsive for both mobile web browsers (e.g., Chrome, Safari) and native mobile app development (iOS and Android). This requires building a responsive UI that adapts to screen sizes.  **Cost/Time/Expertise**: Higher cost and time due to the need to develop for multiple platforms (iOS and Android). Expertise in mobile frameworks (like React Native or Flutter) may be necessary. |
| **Development Tools** | **Programming Languages and Tools**: Development tools include Xcode, Swift, and web technologies (HTML, CSS, JavaScript). For cross-platform apps, tools like React or Electron can be used.  **Impact on Team**: Developers may need familiarity with Mac-specific tools, but many web technologies are cross-platform.  **Licensing Costs**: Xcode is free, but Apple hardware can be expensive | **Programming Languages and Tools**: Common tools include open-source IDEs like Eclipse, and languages like Java, Python, and JavaScript. Docker and Kubernetes are popular for Linux-based deployments.  **Impact on Team**: Linux development is flexible and open-source, with many free tools available. Developers need to be comfortable with the Linux command line.  **Licensing Costs**: Minimal, as Linux and most of its tools are free and open-source. | **Programming Languages and Tools**: Windows developers often use Visual Studio, .NET, and other proprietary tools. For web development, JavaScript and C# are common.  **Impact on Team**: Expertise in Microsoft technologies is required, which may lead to the need for specific teams focused on Windows tools.  **Licensing Costs**: Windows licenses can be expensive, and many proprietary tools require paid licenses. | **Programming Languages and Tools**: Tools for mobile development include Xcode (for iOS), Android Studio (for Android), and cross-platform frameworks like React Native and Flutter.  **Impact on Team**: Mobile app development requires specialized expertise, particularly in managing platform differences between Android and iOS.  **Licensing Costs**: Android Studio is free, but Xcode development requires an Apple Developer license ($99/year). |

**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**:

Given The Gaming Room’s intention to expand "Draw It or Lose It" to other computing environments, I recommend **Linux** as the most suitable operating (server) platform for hosting this web-based application.

**Rationale:**

* Linux is known for its stability, flexibility, and scalability, making it ideal for handling high traffic volumes and multi-instance game hosting.
* Its open-source nature ensures low licensing costs, reducing operational expenses for The Gaming Room. Moreover, Linux provides a wide variety of server software (such as Apache, Nginx, and Docker) that can facilitate scaling the application to thousands of users across multiple platforms, ensuring efficient resource management.
* Linux is also highly compatible with cloud platforms like AWS, Azure, and Google Cloud, allowing the company to choose from various infrastructure providers to ensure the best performance at minimal costs.

Choosing Linux gives The Gaming Room the ability to deploy on different hosting environments, from local servers to cloud-based infrastructures, without being locked into proprietary solutions.

1. **Operating Systems Architectures**:

The chosen operating platform, **Linux**, follows a **monolithic kernel architecture**, which offers several advantages for performance and scalability.

**Key Characteristics:**

* **Monolithic Kernel Architecture**: In a monolithic kernel system like Linux, all system services, including memory management, process management, and hardware drivers, run in kernel mode, providing faster execution. The modularity of Linux allows it to be configured and optimized for specific server environments, ensuring high performance even under heavy load.
* **Process Isolation**: Linux supports strong process isolation through the use of namespaces and control groups (cgroups). This allows each instance of the game (each player session) to run in its own isolated environment, preventing resource contention and security vulnerabilities between instances.
* **Scalability**: The modularity and flexibility of Linux make it ideal for horizontal scaling, allowing The Gaming Room to expand by adding more servers or virtual instances seamlessly as the player base grows.
* **Security**: Linux is well-known for its robust security features, such as SELinux (Security-Enhanced Linux) and AppArmor, which provide granular control over access permissions, ensuring that user data is protected even in a multi-instance environment.

This architecture will support the performance and security demands of a multi-platform, multi-user game like "Draw It or Lose It."

1. **Storage Management**:

For handling the large number of high-definition images and game data, I recommend implementing a **cloud-based storage solution** with **object storage systems** such as **Amazon S3** or **Google Cloud Storage**.

**Why Cloud Object Storage?**

* **Scalability**: Object storage systems are ideal for handling large amounts of unstructured data (like game images). They automatically scale as more data is added, ensuring that storage can grow without manual intervention.
* **Cost Efficiency**: Cloud providers offer tiered storage, so The Gaming Room can move older game data (like previous game sessions) to cheaper, slower storage while keeping frequently accessed files (like current game images) in faster storage tiers.
* **Global Availability**: Using cloud storage ensures that players across different geographical regions experience low-latency access to the game’s data. Cloud platforms often replicate data across multiple regions, providing redundancy and high availability.
* **Performance**: With high-speed networks connecting the cloud storage to the game servers, the game will be able to retrieve and serve image files efficiently, ensuring smooth gameplay.

Additionally, **compression techniques** should be used to reduce the size of the image files, further optimizing storage costs and improving data transfer speeds.

1. **Memory Management**:

To ensure optimal memory management for "Draw It or Lose It," the Linux platform uses several advanced memory management techniques that can handle the game's real-time rendering and multi-instance environment.

**Memory Management Techniques in Linux:**

* **Demand Paging**: This technique ensures that only the portions of memory that are needed are loaded, reducing the overall memory footprint and ensuring efficient use of system resources. As images are loaded into memory during gameplay, demand paging will minimize unnecessary memory usage.
* **Swapping and Virtual Memory**: Linux’s swapping mechanism allows it to handle memory overflow efficiently. When physical memory is full, less critical data is swapped out to disk, freeing up RAM for more critical tasks, such as rendering high-definition images.
* **Caching**: Frequently used images and data can be cached in memory, allowing for faster access during gameplay. This reduces latency when switching between images in the game rounds.
* **Garbage Collection and Memory Pooling**: The game’s processes will periodically release unused memory through garbage collection, ensuring that memory is continuously recycled and made available for new tasks. Memory pooling can further optimize resource allocation by reusing memory blocks for similar tasks.

These techniques ensure that the game will run smoothly even on platforms with limited physical memory, like mobile devices or smaller desktop systems.

1. **Distributed Systems and Networks**:

To facilitate communication between the different platforms, "Draw It or Lose It" can be implemented as a **distributed system** using a **client-server architecture** with RESTful APIs.

**Communication Strategy:**

* **RESTful APIs**: The game server can expose RESTful APIs to handle game data requests, such as fetching images, updating game scores, and managing team interactions. REST APIs are platform-agnostic and can be easily integrated into web, mobile, and desktop applications.
* **Load Balancing**: A load balancer can distribute traffic evenly across multiple game servers to ensure that no single server is overwhelmed by player requests. This also ensures high availability and reduces the impact of any single server failure.
* **Fault Tolerance and Redundancy**: By replicating the game’s server instances across multiple regions or data centers, the game can maintain availability even in the event of network outages or server crashes.
* **Eventual Consistency**: Given that the game will involve multiple players interacting in real-time, eventual consistency techniques can be used to synchronize game states across platforms. This ensures that players see the same updates, regardless of whether they are playing on desktop, mobile, or another platform.

The distributed nature of the system allows for scalability while ensuring that game data and interactions remain consistent across all platforms.

1. **Security**:

Ensuring security across multiple platforms is critical, especially given the need to protect user data and prevent unauthorized access. For this, I recommend a combination of **encryption**, **role-based access control (RBAC)**, and **secure communication protocols**.

**Security Measures:**

* **Data Encryption**: All user data, including images and game states, should be encrypted both in transit (using SSL/TLS) and at rest. This ensures that even if data is intercepted or stolen, it remains unreadable.
* **Role-Based Access Control (RBAC)**: Implementing RBAC ensures that only authorized users have access to specific game functions. For example, administrators could have higher-level access to manage game settings, while regular users can only interact with the game content.
* **OAuth 2.0 Authentication**: Using OAuth 2.0 will provide a secure way to authenticate users across different platforms, ensuring that user credentials are never exposed to third parties.
* **Secure API Access**: APIs should be secured using API keys or tokens, preventing unauthorized access to the game’s backend systems.
* **Regular Security Audits**: Regular vulnerability scans and security audits should be conducted to ensure that the system remains secure as it scales and evolves.

These combined security practices will ensure that both user data and gameplay interactions remain secure, providing peace of mind to both players and the client.