

**Draw it or Lose it**

# **CS 230 Project Software Design Template**

Version 1.2

## 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 4**](#_Toc115077324)

[**Evaluation 5**](#_Toc115077325)

[**Recommendations 6**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 11/17/2024 | Shah Ali Omor | Initial version of the CS 230 project design document. |
| 1.1 | 12/01/2024 | Shah Ali Omor | Updated the Evaluation section. |
| 1.2 | 12/15/2024 | Shah Ali Omor | Added Recommendations section and finalized document. |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room, a client of Creative Technology Solutions (CTS), aims to develop a web-based distributed platform for their game, Draw It or Lose It. The objective is to extend the functionality of the existing Android-based game to support multiple platforms while ensuring robust and efficient management of game data, teams, and players. This design will incorporate industry-standard software design patterns such as Singleton and Iterator to fulfill the technical requirements, while ensuring scalability, uniqueness of names, and ease of management.

## Requirements

**Business Requirements**:

* Support for multiple teams and players per game.
* Ensure team and game names are unique to avoid conflicts.
* Manage game sessions efficiently to serve a distributed environment.

**Technical Requirements**:

* Implement a Singleton design pattern to ensure only one instance of the GameService exists.
* Incorporate the Iterator pattern for efficient management of entities like games, teams, and players.
* Develop a robust Entity class to standardize common attributes such as id and name for all entities.

## [Design Constraints](#_2et92p0)

1. **Singleton Pattern**: The GameService class must follow the Singleton pattern to ensure centralized management of all game-related data. This approach restricts the class from having multiple instances, reducing memory usage and improving data consistency. This ensures thread safety and prevents conflicts in game data access across platforms.
2. **Unique Names**: Names of games, teams, and players must be validated for uniqueness using the Iterator pattern before creating new entities. This prevents redundancy and user confusion while maintaining the integrity of data across distributed systems.
3. **Modularity and Scalability**: The system should use an Entity class as a base for shared attributes and behaviors, promoting modularity and adherence to object-oriented principles. This simplifies the addition of new features or entities while reducing code duplication.

## [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 gives a clear picture of how the classes in this application are structured and how they work together to meet the requirements. At the core of the design is the Entity class, which acts as a foundation for other classes like Game, Team, and Player. By using inheritance, we’ve been able to centralize common attributes like id and name, as well as shared methods like getId, getName, and toString. This makes the code cleaner and ensures that each class only needs to focus on what makes it unique.

The GameService class is the backbone of the application. It uses the Singleton pattern, which ensures there’s only one instance of the service at any time. This is a really efficient way to manage shared data, especially in a distributed environment where multiple users might be accessing the game at once. This approach keeps things organized and aligns perfectly with the client’s requirement to only allow one active game instance.

The relationship between Game, Team, and Player is set up in a way that makes it easy to scale. For example, each game can have multiple teams, and each team can have multiple players. This uses composition, where lists (List<Team> and List<Player>) allow us to dynamically add or retrieve teams and players. At the same time, we’ve ensured data integrity by using the iterator pattern in methods like addGame, addTeam, and addPlayer. This means we check for duplicate names before adding new entities, which keeps things consistent.

Another key principle here is polymorphism. Each class overrides the toString method to provide a custom string representation. This keeps the code flexible while ensuring a consistent way to represent different entities. Encapsulation also plays a big role, as each class manages its own attributes and behavior, making it easier to maintain and debug.

Finally, the ProgramDriver and SingletonTester classes help us tie everything together. The ProgramDriver serves as the starting point for testing the application, while SingletonTester ensures that the Singleton pattern in GameService is working as expected. Overall, the design reflects principles like modularity and scalability, making it efficient, easy to extend, and well-suited for the client’s needs.

**"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)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Mac is known for its reliability and security, making it a good choice for hosting web-based applications. However, its higher cost and limited server tools compared to Linux can be a drawback. Macs are generally used in development rather than deployment environments. | Linux is a widely preferred choice for servers due to its open-source nature, scalability, and vast community support. It offers unmatched performance, but it may require more expertise to set up and maintain effectively. | Windows is a versatile server platform with excellent support for enterprise applications, especially those built with .NET. However, licensing costs and potential vulnerabilities are its primary weaknesses. | Mobile devices are not typically used as server environments. However, hosting lightweight services or local applications can be achieved through devices like Android servers. Battery life and processing power are major limitations. |
| **Client Side** | Developing for Mac users can be expensive due to hardware costs. However, the ecosystem ensures seamless performance and a refined user experience. Expertise in Swift and Objective-C is essential for native macOS development. | Linux provides flexibility and is ideal for users who prioritize customization. However, development on Linux often requires expertise in various distributions and tools like GTK or Qt. It is also less common for end-user clients. | Windows is user-friendly and supports a broad range of software, making it a top choice for general client-side use. Development costs are moderate, and expertise in C#, .NET, or Windows frameworks is helpful for building native applications. | Mobile development requires expertise in native languages like Java/Kotlin for Android or Swift for iOS. Cross-platform tools like Flutter or React Native can reduce costs and development time while maintaining performance. |
| **Development Tools** | macOS development often relies on tools like Xcode for native app development. Swift and Objective-C are the primary languages, while IDEs like IntelliJ IDEA support cross-platform work. | Development on Linux is heavily reliant on open-source tools. IDEs like Eclipse, NetBeans, and text editors like Vim are commonly used. Developers often use C/C++, Python, or Java for Linux-based systems. | Windows development is supported by tools like Visual Studio, which is highly versatile and powerful. C#, .NET, and other Microsoft-centric frameworks dominate this environment. | Mobile development uses a mix of tools. Android Studio is the go-to IDE for Android, while Xcode serves iOS development. For cross-platform, tools like Flutter or React Native streamline the process. |

## Recommendations

1. **Operating Platform**: Linux is recommended as the operating platform due to its cost-effectiveness, scalability, and robust performance. It offers a secure and efficient environment for hosting "Draw It or Lose It" and supports distributed systems, making it ideal for expansion to various platforms.
2. **Operating Systems Architectures**: Linux’s modular monolithic architecture integrates core functionalities directly within the kernel, resulting in high performance and low latency. Features such as multithreading, process scheduling, and system calls ensure efficiency and reliability. This is particularly beneficial for handling multiplayer games that require quick responses and seamless coordination between players.
3. **Storage Management**: Amazon S3 is recommended for scalable and redundant object storage. It ensures high availability of game data, including user profiles and game states, which are critical for real-time gaming. For centralized file storage within the server environment, NFS (Network File System) can also be employed, enabling efficient data sharing across multiple servers.
4. **Memory Management**: Linux's dynamic memory allocation and virtual memory techniques ensure efficient resource utilization. Shared memory can optimize inter-process communication, reducing latency during multiplayer sessions. This approach improves server performance and provides a smoother gaming experience for users.
5. **Distributed Systems and Networks**: A microservices architecture is recommended for flexibility and scalability. Docker containers can be used to deploy and manage game components across multiple environments. Load balancers should be implemented to distribute network traffic evenly, while CDNs (Content Delivery Networks) can cache static assets to minimize latency. Redundancy mechanisms should also be in place to mitigate risks of outages and maintain seamless gameplay.
6. **Security**: To protect user data and maintain trust across platforms, implement SSL/TLS encryption for secure communication, OAuth for user authentication, and firewalls to prevent unauthorized access. Regular security audits and adherence to privacy regulations like GDPR will further enhance security and compliance.