

Draw It or Lose It

# **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 | 10/17/2024 | Rebecca Shane | Executive summary, design constraints, UML diagram |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room is seeking to expand the reach of their existing Android game, *Draw It or Lose It*, by developing a web-based version that can operate on multiple platforms. This game allows teams to compete by guessing puzzles based on drawings rendered by the application. The goal is to design a solution that meets the client’s needs for a multi-platform, web-based distributed environment while preserving key features of the original game. To achieve this, the application will need to support multiple teams and players, ensure that game and team names are unique, and be designed in a way that only allows a single instance of the game to exist at any given time. The proposed solution will incorporate essential software design principles and patterns to ensure the application is scalable, efficient, and meets all client requirements.

## Requirements

The game application requires several key functionalities. First, the application must support multiple teams, with each team consisting of multiple players, ensuring scalability for larger games. The system must enforce unique names for both games and teams, allowing users to verify name availability during the setup process. Additionally, only one instance of the game should exist in memory at any given time, which can be accomplished using unique identifiers for each game, team, and player. This will help maintain the integrity of the system, especially in a distributed, web-based environment.

## [Design Constraints](#_2et92p0)

Developing a web-based version of *Draw It or Lose It* introduces several key design constraints that must be considered. First, the game must be compatible across multiple platforms, which requires using technologies that work well in a web-based, cross-platform environment. JavaScript will be essential for ensuring a smooth user interface, while a scalable backend using languages like Java or Python can handle server-side processing and database management.

Another important constraint is concurrency and scalability. Since the game will involve multiple teams and players interacting in real time, the application must be able to handle concurrent users effectively. Real-time communication will likely need to be facilitated through technologies that allow seamless interaction among players.

Additionally, the requirement for unique game, team, and player names introduces a constraint on data integrity. The system must ensure that names are checked and validated in real time to avoid conflicts. This will require a robust mechanism for querying the database to guarantee that no duplicate names exist.

Lastly, the need to ensure that only one instance of the game exists in memory at any given time requires the implementation of the Singleton design pattern. This pattern ensures that only one object of the *GameService* class is created and maintained throughout the lifecycle of the application, effectively managing the game’s lifecycle and preventing duplicate instances.

## [Domain Model](#_8h2ehzxfam4o)

The UML class diagram for the game application shows how the various classes relate to each other and illustrates important object-oriented programming (OOP) principles. The *GameService* class is central to the system, managing instances of games, teams, and players. It follows the Singleton pattern, ensuring that only one instance of *GameService* exists at any time, preventing duplicate data management and ensuring consistency. The *Game*, *Team*, and *Player* classes all inherit from the *Entity* base class, which provides common attributes such as *id* and *name*, demonstrating inheritance and promoting code reuse. The relationships between *Game* and *Team*, and between *Team* and *Player*, demonstrate composition, where games can have multiple teams, and teams can have multiple players, but these entities are dependent on each other.

OOP principles like inheritance, composition, and encapsulation are efficiently used to fulfill the software requirements. Inheritance allows shared behaviors across classes, reducing code duplication. Composition ensures the hierarchical structure needed for games, teams, and players. Encapsulation, by using private fields and public methods, protects data integrity. These principles together provide a clear, scalable, and efficient design for managing game-related entities, ensuring that game names are unique, and multiple teams and players can exist within a single game instance.

**"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** | Mac is reliable for hosting web-based software. It offers good performance and security. However, Mac is expensive compared to Linux or Windows. MacOS servers are less common in large-scale environments. It supports common development tools like Docker, Apache, and MYSQL. | Linux is a widely preferred operating system for hosting web applications due to its performance, security, and open-source nature. It offers a large range of server distributions, and is highly configurable and adaptable, working for both small and large-scale deployments. Its weakness lies in the need for more technical expertise to manage. | Windows Server editions are popular for web hosting, particularly in enterprises. It provides excellent support for Microsoft tools and services. It is more resource-heavy, can be costly for licenses, and may require more system updates compared to Linux. | Hosting directly on mobile devices isn’t a common practice due to hardware limitations, but they can act as development or testing platforms for mobile web applications. Mobile devices can emulate a server environment for local testing using tools, but it is not practical for production environments. |
| **Client Side** | Mac requires supporting multiple macOS versions and ensuring compatibility with Safari, the default browser. While Apple provides good development tools, it’s more expensive to maintain Apple hardware for testing. Time to develop for Mac clients may also increase due to Apple’s strict guidelines for integration and UI consistency. | Linux clients will need to support various distributions (Ubuntu, Fedora, etc), and ensuring compatibility across multiple desktop environments can be challenging. Linux clients offer flexibility in using open-source tools and libraries. The development process is low-cost but may require more time for testing across different environments. | Windows development involves supporting a large variety of hardware configurations and ensuring compatibility with different versions of Windows (10, 11, etc). Microsoft provides strong development tools like Visual Studio, but the cost of licensing Windows clients and supporting legacy systems may increase complexity and time for development. | Developing for mobile devices requires consideration of multiple platforms such as Android and iOS. Each mobile OS has unique hardware, software, and screen size requirements. Android development involves more fragmentation in terms of OS versions and devices, while iOS tends to have stricter app approval processes but more uniformity. Development time and cost can increase due to the need for platform-specific expertise and extensive testing. |
| **Development Tools** | Development for Mac typically involves using Xcode, which supports Swift and Objective-C for native applications. For web applications, you can use a variety of languages including Python, Ruby, and more. Mac also supports Docker for containerized environments. | Linux development typically relies on open-source tools and languages like Python, JavaScript, and more. Linux also supports powerful package managers, and tools like Docker, which make it highly efficient for server-side development. | Windows development is often done in languages like C#, Java, or JavaScript. Visual Studio is the most common IDE used, offering extensive support for .NET and other frameworks. Other options include Visual Studio Code for lighter web development and Eclipse for Java. Windows also supports WAMP stacks (Windows, Apache, MySQL, PHP) for development environments. | Development for mobile devices is primarily done using Android Studio (for Android development using Java or Kotlin) or Xcode (for iOS development using Swift or Objective-C). Cross-platform development tools are also popular, allowing developers to write code once and deploy it to both Android and iOS. Debugging and testing tools such as Android Emulator or iOS Simulator are essential for this environment. |

## 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**: I recommend using Linux as the primary operating platform for expanding *Draw It or Lose It* to other computing environments. Linux is an open-source, highly customizable operating system that supports a wide variety of hardware architectures. Its stability and efficiency make it an ideal platform for server environments, particularly when managing a multi-user game with high demands on performance and scalability. Linux’s strong support for distributed systems and containerization technologies, like Docker and Kubernetes, would allow for easy deployment and scaling across multiple platforms, including web, mobile, and console environments.
2. **Operating Systems Architectures**: The Linux operating system architecture is based on a monolithic kernel, which allows for efficient management of system calls, memory, file systems, and device drivers within the kernel space. This architecture provides high performance and flexibility. Additionally, Linux's modularity allows developers to fine-tune the system to meet specific requirements, which is particularly useful for scaling the game across multiple platforms. It also offers robust support for multi-threading and process scheduling, which ensures smooth operation even when managing multiple game instances concurrently.
3. **Storage Management**: For the storage management system, I recommend using a cloud-based storage solution like Amazon S3 or Google Cloud Storage in conjunction with a Linux file system such as Ext4. These systems are optimized for scalability, redundancy, and speed, which will be essential for managing the large library of high-definition images used in *Draw It or Lose It*. Cloud storage will ensure that the files are accessible from any platform, while Linux’s native file systems provide efficient handling of file operations, caching, and disk space allocation.
4. **Memory Management**: Linux’s memory management techniques are well-suited to handle the requirements of *Draw It or Lose It*. The platform efficiently manages memory through demand paging and virtual memory, allowing processes to use more memory than what is physically available. Additionally, Linux uses a memory allocator called the “buddy system” to allocate and deallocate memory blocks quickly. For high-performance applications like this game, the kernel’s ability to handle large amounts of memory for file caching and buffer management will ensure smooth rendering and fast performance, even under heavy load.
5. **Distributed Systems and Networks**: To support communication between different platforms, I suggest implementing a distributed system using RESTful APIs or WebSocket protocols for real-time interaction. By using Linux’s strong support for networking and distributed services, the game can run across different devices with seamless communication between clients. Cloud services or distributed databases such as MongoDB can manage player sessions and game states, ensuring data synchronization across platforms. The architecture will rely on robust connectivity protocols and load balancing to handle potential outages or connectivity issues, ensuring the game remains available and responsive.
6. **Security**: Security is critical for protecting user data and ensuring the integrity of the game application. Linux provides strong security features such as SELinux for access control and process isolation, ensuring that unauthorized access is prevented. Additionally, encryption protocols like SSL/TLS can secure data transmission between platforms, while multi-factor authentication and proper user role management can safeguard user accounts. Regular security patches and updates can further protect the platform from vulnerabilities. By incorporating these features, The Gaming Room can ensure that user information is protected across all platforms and during communication between different devices.