

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 | 11/13/24 | Joe O’Brien | Initial implementation of Gaming Room web app. |

**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 ask of the Gaming Room is to is the create the web-based version of its existing Android game, Draw it or Lose it. This game allows players to participate in a team-based guessing game across multiple platforms. The game will select and display images as clues for the player to guess at a steady rate. To accomplish this the system should be able to manage game instances, ensure unique team and player names, and support multiple users.   
  
The solution CTS is offering involves a scalable web application that utilizes object-oriented principles and design patterns that meet the requirements provided. The systems should include classes for Game, Team and Player which all inherit from an Entity class for streamlined and maintainable code. We will utilize the singleton pattern in the GameService class to ensure efficient memory usage per the client’s request. The singleton pattern will guarantee a single active game instance. Also, the iterator pattern will manage unique IDs for games, teams and players which will prevent duplicates and better the user experience.   
  
This approach provides flexibility, scalability, and efficiency to provide the ability for “Draw it or Lose it” to handle the growing player base and add future enhancements. Adherence to best practices will not only meet the current requirements but allow for quick and easy additions to the game in the future. This document will outline the technical and design requirements of the Gaming Room that will guide the development process.

## Requirements

The Gaming Room has provided several requirements and expectations of the web-based version of their game “Draw it or Lose it”. The ask is for the game to be able to support multiple users and teams across various platforms while maintaining unique IDs and efficient resource management. The requirements being referred to are:  
  
1. **Supports Multiple Teams:** The game should allow multiple teams to participate in a single game instance.   
  
2. **Multiple players per team:** Each team should be able to have more than one player to allow for interaction and group gameplay.

3. **Unique game and team names:** The game will allow for unique game and team names. To ensure game and team names are unique, there will be a check for name availability when creating or joining a team.

4. Single game instance: Only one instance of a game should exists at any given time. This helps memory management and prevents conflict with other game instances.   
  
The requirements listed above will ensure the desired result from the client and the users of the application.

## [Design Constraints](#_2et92p0)

Several design constraints come to mind when considering development for the Draw it or Lost it game. These constraints must be considered to ensure proper functionality, scalability, and performance across different platforms.   
1. **Performance, Storage and Memory management:** The system must be able to support many users at once as the game grows in popularity. Application processing and data storage need to be considered to support this. This implies a cloud storage solution may need to be used for the backend as cloud solutions provide a scalability and affordability. This will also provide a smooth user experience.   
  
2. **Security Management:** Security is important to not only protect user information but to provide a fair playing ground. The application must secure data transmission. This may require implementing data encryption and authentication.

## [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 class diagram illustrates relationships and structure between classes within Draw it or Lose it. The diagram highlights important classes like Entity, Game, Team, Player, and GameService and how the interact.   
  
The Entity class serves as the base of the Game, Team, and Player classes, providing unique identifiers and names. The attributes ensure the uniqueness of games, teams, and players which directly addresses the requirement for unique game and team names.   
  
The Game class represents a single game instance and holds a list of teams that are involved in the game. This class inherits the Entity class which ensures each game has a unique identifier and name. Encapsulation is implemented here which restricts direct access to the list of teams which limits modification to only the defined methods. This design satisfies a game will have one or more team’s requirement.

The Team class holds a list of players that are assigned to each team and the class inherits common attributes from the Entity class. This makes each team uniquely identifiable. Much like the Game class, the Team class encapsulates its list of players to provide data integrity. This design is to satisfy the requirement for each team to have multiple players.   
  
The Player class represents each individual player and inherits the id and name from the Entity class. This provides consistency with other entities. Encapsulation is also applied to protect player attributes and aligning with object-oriented design principles.  
  
GameService is responsible for game instances including creation of games, teams, and players. The singleton pattern is also implemented here to ensure only one instance of GameService exists in memory at any given time.   
  
Finally, the Program driver class contains the main method for running the program. This is where initial configurations are tested and verified to ensure the application runs as intended.

**"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 typically not very common for server-side features as the server-side features are very limited and the hardware typically costs quite a bit more than other options. The OS lacks scalability compared to other systems making it suitable for smaller environments but not for a game that can potentially host many users. | Linux is a widely used because it is very cost efficient due to its open-source nature. Stability and scalability also contribute to its likeness making it an ideal choice for hosting web-based applications. Linux is highly customizable and offers great performance and security as well. | Windows Server is another widely used technology for server-side development but can be very costly due to licensing fees. Windows also requires a lot of maintenance like updates and patches which make it difficult to maintain stability of an application. | Mobile devices are more client-oriented and are rarely if ever used for server side requirements. Mobile devices lack stability, scalability and processing power making it a poor choice for server-side development. |
| **Client Side** | MacOS requires Mac specific frameworks and interfaces to develop applications to run on the system. This will typically increase costs of development due to the “Mac-Specific” hardware and software required. | Due to the many distributions of Linux, extensive knowledge of the operating system will be necessary for testing Which could be time consuming. The user base for Linux clients is much smaller therefore a web-based solution would make more sense to alleviate compatibility issues. | Windows is a cost effective and widely supported ecosystem with many experienced developers. Windows provides a lot of support for web based apps making it a very likeable operating system for development. | Mobile clients require a responsive interface that can adapt to many different screen sizes and operating systems. Mobile clients also require consideration for touch input and low power performance optimization for a battery powered mobile device. |
| **Development Tools** | Mac offers relevant tools like Xcode and Visual Studio code where you can develop for languages like Swift and JavaScript. Mac supports Unix based commands which is valuable for cross platform development. | Linux supports quite a few options for development tools like Eclipse, Visual Studio Code and IntelliJ. Linux is typically favored for server-side development due to its compatibility with many different environments. | Windows is also compatible with a wide range of development tools a few of which are VS Code, Eclipse, and NetBeans. Windows supports development for many languages as well including but not limited to C#, C++, Java, JavaScript and many more. | For native mobile development, Android Studio and Xcode are essential. Languages used are Java (Android) and Swift(iOS). React Native is a JavaScript framework used for cross-compatibility between iOS and Android. These development tools are optimized for responsive design and touch interfaces. |

## 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**: A Linux based cloud platform would be best for the Gaming Room to expand its game to other platforms. Some examples of these systems could be Amazon Web Services or Azure. Linux is scalable and cost effective making it a great candidate for web-based game applications. The open-source nature of Linux supports the cross-platform requirement from the Gaming room and supports flexibility and the possible high concurrency of the application.
2. **Operating Systems Architectures**: As recommended above, the architecture should be cloud-based and run on Linux servers. This architecture can divide the application into smaller services making it more manageable and scalable as needs grow. The services can all be responsible for a for different functions like authentication and game logic.
3. **Storage Management**: Cloud based storage, like Amazon S3, is recommended for Gaming Room. Cloud storage like S3 is easily scalable and is cost efficient and simple to maintain. Cloud storage can also provide a consistent experience to all users no matter their location.
4. **Memory Management**: The cloud-based Linux platform recommend effectively manages memory with memory allocation and deallocation techniques and garbage collection. Memory limits can also be customized for each service. This will make sure each part of the application will only use memory that is necessary. These techniques provide an expectation of reliability considering the possible size of the player base.
5. **Distributed Systems and Networks**: Distributed systems rely on network connectivity, so resilience is important. Load balancers can be used to help avoid outages. The game should us a RESTful API to communicate between the servers. Caching strategies can be used to reduce latency of these communications.
6. **Security**: Encryption like SSL/TLS can be used to secure data transmission between players and servers. User sessions can be secured with token-based authentication systems like JSON Web Tokens (JWT). Also, firewalls Intrusion detection systems (IDS) can be added to protect important information and ensure a trustworthy environment.