

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 | 08/07/2024 | Happy Dooley | <Brief description of changes in this revision> |

**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 Gaming Room is seeking to develop a web-based version of their popular Android game, *Draw It or Lose It*. The game will have teams competing to guess what is being drawn from a library of stock images within a set time frame. The current game is limited to the Android platform, but the client desires to expand by making it available on multiple platforms using a web-based solution.

The solution involves creating a web-based game application that keeps to the client's software requirements like allowing multiple teams and players, ensuring the uniqueness of game and team names, and managing game instances efficiently to prevent duplication. This document outlines the design considerations and constraints that are needed to develop a scalable, responsive, and user-friendly web application.

## Requirements

**Team and Player Management:**

* **Multiple Teams:** The game must support multiple teams, with each team consisting of several players.
* **Unique Team Names:** Each team name must be unique to avoid conflicts. The system must validate the uniqueness of team names at the time of creation.
* **Player Assignment:** Players should be able to join teams, and the system must handle player assignments efficiently.

**Game Instance Management:**

* **Single Game Instance:** Only one instance of the game can exist in memory at any given time. This will be enforced by assigning unique identifiers to each game instance, ensuring that no duplicate instances are running simultaneously.
* **Unique Identifiers:** The system must generate unique identifiers for games, teams, and players to ensure that each entity is distinct and easily trackable.

**Gameplay Requirements:**

* **Game Rounds:** The game will consist of four rounds, each lasting one minute. During each round, images will be rendered as clues, with the drawing process completing by the 30-second mark.
* **Guessing Mechanism:** If a team fails to guess the puzzle within the allotted time, other teams should be given 15 seconds to offer one guess each.

**User Experience:**

* **Cross-Platform Accessibility:** The game must be accessible on various platforms, including desktops, tablets, and mobile devices. The user interface should be responsive and adapt to different screen sizes.
* **Real-Time Interaction:** The system must support real-time interaction between players and teams, ensuring that game updates (e.g., drawings, guesses) are reflected instantly for all participants.

**System Performance:**

* **Scalability:** The application must be designed to handle an increasing number of users without compromising performance. This includes efficient handling of simultaneous game sessions and user requests.
* **Low Latency:** The game must provide a smooth and responsive user experience with minimal latency, especially during gameplay.

**Security:**

* **Data Protection:** The system must ensure the protection of user data, employing encryption and secure communication protocols (e.g., HTTPS) to prevent unauthorized access.
* **Authentication:** Secure authentication mechanisms must be implemented to ensure that only authorized users can access the game.

## [Design Constraints](#_2et92p0)

**Cross-Platform :**   
The game needs to be accessible across multiple platforms such as desktop, tablet, and mobile. This requires using web technologies that support a responsive design and cross-platform compatibility like HTML5, CSS3, and JavaScript.

**Security:**   
Since the game will be web-based, it must have security measures to protect the data of users and prevent unauthorized access. This would implementing HTTPS, data encryption, and secure authentication methods.

**Latency:**   
For the user to have a seamless experience, the system must minimize latency, particularly in real-time game actions and updates. The architecture should prioritize low-latency communication methods between the client and server.

**Concurrency Management:**   
Only one instance of the game can exist in memory at any given time. This constraint will demand efficient concurrency management and unique identifiers for game instances, teams, and players. The backend system must be designed to handle simultaneous requests without conflict.

**Scalability:**   
The application must be designed to scale with the number of users. This would mean there is a need for a scalable architecture, potentially using cloud services for load balancing, data storage, and processing power to handle increased demand.

## [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)

**Entity Class:**

* The Entity class serves as a base class with common attributes such as id and name. It uses fundamental methods like getId(), getName(), and toString(), which are inherited by the Game, Team, and Player classes.
* **OOP Principle - Inheritance:** The Entity class shows inheritance, where common attributes and methods are defined in a superclass (Entity), and specialized classes (Game, Team, Player) inherit these features, promoting code reuse and reducing redundancy.

**Game Class:**

* The Game class contains a list of Team objects (teams: List<Team>) and provides methods to add teams and manage game-related information. Each Game can have zero or more Teams associated with it.
* **OOP Principle - Association:** The Game class is associated with the Team class, representing a one-to-many relationship where a single game can have multiple teams.

**Team Class:**

* The Team class holds a list of Player objects (players: List<Player>) and methods to add players to the team. A Team can have zero or more Players associated with it.
* **OOP Principle - Composition:** The Team class uses composition to manage its players, indicating that a team is composed of multiple players. This closely aligns with the game’s requirement to manage teams and players.

**Player Class:**

* The Player class, which extends the Entity class, represents individual players in the game. Each player is uniquely identified and named, ensuring the uniqueness of team and player names as required by the system.
* **OOP Principle - Encapsulation:** The Player class encapsulates the player’s attributes and behavior, allowing the class to control access to its data and ensure that player-related operations are handled within the class.

**GameService Class:**

* The GameService class is responsible for managing the collection of Game objects. It provides methods to add and retrieve games, generate unique IDs for games, teams, and players, and ensure that only one instance of the game exists at any time.
* **OOP Principle - Singleton:** The GameService class follows the Singleton design pattern, ensuring that only one instance of this service exists to manage all game-related operations. This aligns with the requirement to have only one instance of the game in memory at any given time.
* **OOP Principle - Aggregation:** The GameService class aggregates Game objects, emphasizing the relationship between the service and the games it manages without implying ownership.

**ProgramDriver and SingletonTester Classes:**

* The ProgramDriver class contains the main() method, acting as the entry point for the application. It utilizes the GameService to manage game instances.
* The SingletonTester class is used to test the Singleton pattern implemented by the GameService class.
* **OOP Principle - Testing and Use Cases:** These classes are responsible for initiating the application and testing the Singleton pattern, demonstrating how the system will be executed and validated.

**"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 a Unix-based operating system known for its stability, security, and user-friendly interface. It supports various web development tools, such as Apache, Nginx, and databases like MySQL and PostgreSQL, making it useful for server-side hosting. However, macOS is not as widely used in production environments as Linux, and the cost of Apple hardware is generally higher compared to others. | Linux is an open-source operating system widely regarded for its flexibility, reliability, and low cost, making it the preferred choice for server environments. It supports nearly all programming languages and frameworks, allowing for highly customizable server setups. Despite its strengths, Linux requires more technical expertise to manage, which can be a drawback for less experienced users. | Windows Server is a popular choice for hosting web-based applications due to its user-friendly interface and wide-ranging software support. It is particularly strong in environments that rely on Microsoft products, offering native support for ASP.NET. However, Windows has higher licensing costs and is generally considered less secure than Linux due to its susceptibility to malware. | Hosting on mobile devices will usually involve cloud-based solutions that is optimized for mobile access or dedicated mobile hosting environments. This approach allows for real-time access to services and apps, enhancing user experience. However, mobile hosting is typically limited in power and capabilities compared to traditional server setups, and it may have specific security vulnerabilities. |
| **Client Side** | Developing for macOS requires a higher initial investment due to the cost of Apple hardware, but it can be time-efficient thanks to tools like Xcode. However, developers need to ensure cross-platform compatibility, and expertise in macOS and iOS development is essential for native applications. | Linux offers a cost-effective solution as it is open-source, but development may take longer due to the need for ensuring cross-platform compatibility. Developers must be proficient in Linux-based systems and comfortable using multiple programming languages, making expertise a significant factor. | Windows development is often quicker due to the prevalence of tools like Visual Studio, but the associated software and licensing fees can increase costs. Developers need to be well-versed in Windows-specific technologies and potentially proprietary languages like C#. | Developing for mobile devices involves varying costs depending on the targeted platforms (iOS, Android). Development time can be extended due to the need for platform-specific adjustments. Expertise in mobile-specific development frameworks like Swift for iOS or Kotlin/Java for Android is essential. |
| **Development Tools** | Development on macOS typically involves using Xcode, Swift, and Objective-C, among other tools. Xcode provides a robust suite for macOS and iOS development, but its tools are not cross-platform, which can be a limitation when developing for Windows or Linux environments. | Linux supports a wide variety of development tools like GCC, Python, Java, Ruby, and Node.js. These open-source tools offer great flexibility and customization, though some commercial software may not be available or supported on Linux, which can limit certain development activities. | Windows development relies heavily on tools like Visual Studio, .NET, and languages like C# and C++. These tools are well-integrated with Windows-native applications and enterprise environments, but they often come with pricy licensing costs, which can be a drawback. | Mobile development tools include Swift and Objective-C for iOS, Java and Kotlin for Android, as well as cross-platform frameworks like React Native and Flutter. These tools are highly optimized for mobile app development, but each platform requires specific tools, which can increase the complexity and cost of developing for multiple platforms. |

## 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**:  
   For The Gaming Room to expand "Draw It or Lose It" beyond its current Android environment, I recommend using a cross-platform operating system such as Linux. Linux offers flexibility and compatibility with multiple platforms like macOS, Windows, and mobile devices. Its open-source nature also allows for customization and scalability, making it a solid foundation for the game’s expansion.
2. **Operating Systems Architectures**:   
   The Linux architecture is modular and supports various hardware configurations, making it versatile for different devices. Its architecture is based on a monolithic kernel that manages CPU, memory, and peripheral devices efficiently. The kernel handles multiple tasks like process management, networking, and device drivers, ensuring the game runs smoothly across various platforms.
3. **Storage Management**:   
   For storage management, I suggest using a cloud-based system like Amazon S3 or Google Cloud Storage. These systems offer scalable storage solutions that are reliable and secure. They allow The Gaming Room to store large amounts of game data and user information while providing easy access and quick retrieval across different platforms.
4. **Memory Management**:   
   Linux uses advanced memory management techniques, such as paging and segmentation, to ensure that applications like "Draw It or Lose It" run efficiently. The operating system allocates memory dynamically, meaning that the game will only use the memory it needs at any given time. This prevents memory leaks and ensures that the game performs well, even with limited resources.
5. **Distributed Systems and Networks**:   
   To enable "Draw It or Lose It" to communicate between various platforms, a distributed software system can be implemented using RESTful APIs over a secure network. This allows different instances of the game to share data and synchronize across devices. The system should be designed to handle potential issues like network outages by implementing redundancy and ensuring that data is consistently updated across all platforms.
6. **Security**:   
   Security is critical for protecting user information, especially when dealing with multiple platforms. I recommend using encryption protocols such as SSL/TLS to secure data transmitted between devices. Additionally, Linux offers built-in security features like SELinux (Security-Enhanced Linux), which can help enforce security policies and prevent unauthorized access. User data should be stored securely using encryption both in transit and at rest to protect against breaches.