

<Draw it or Lose it>

# **CS 230 Project Software Design Template**

Version 1.0

## Table of Contents

[**CS 230 Project Software Design Template** 1](#_heading=h.gjdgxs)

[**Table of Contents 2**](#_heading=h.30j0zll)

[**Document Revision History 2**](#_heading=h.3znysh7)

[**Executive Summary 3**](#_heading=h.2et92p0)

[**Requirements 3**](#_heading=h.tyjcwt)

[**Design Constraints 3**](#_heading=h.1t3h5sf)

[**System Architecture View 3**](#_heading=h.4d34og8)

[**Domain Model 3**](#_heading=h.2s8eyo1)

[**Evaluation 4**](#_heading=h.17dp8vu)

[**Recommendations 5**](#_heading=h.26in1rg)

## [Document Revision History](#_heading=h.lnxbz9)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | <9/26/24> | <Wilson Walker> | <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](#_heading=h.35nkun2)

The Gaming Room is seeking to expand their Android-based game "Draw It or Lose It" into a web-based format that serves multiple platforms. The game, which is similar to "Win, Lose, or Draw," involves teams competing to guess images rendered from a library of stock drawings. The objective is to adapt the game to a distributed web environment while maintaining its core functionality, such as team and player management, puzzle solving, and game timing. The solution will involve using object-oriented principles, software patterns like Singleton, and efficient resource management to ensure scalability and cross-platform compatibility.

## Requirements

**Business Requirements:**

1. **Multi-Platform Availability:** The game must be available across multiple platforms, including desktop (Windows, Mac, and Linux) and mobile devices. This will help increase the game's user base and expand its reach beyond Android users.

2. **Team-Based Gameplay:** The game should support multiple teams competing against each other, maintaining the original game's core functionality.

3. **Unique Game Instances:** The system should manage multiple instances of the game running simultaneously, ensuring that each team plays its own unique game without interference from other teams.

4. **Brand Consistency:** The look and feel of the web-based version should remain consistent with the current Android app to retain the brand's identity.

**Technical Requirements:**

1. **Concurrency and Scalability:** The system must handle multiple concurrent users, teams, and games. Efficient management of resources and instances will be required to ensure smooth gameplay without server overloads.

2. **Security:** The platform must safeguard sensitive data, such as usernames and passwords, and prevent unauthorized access to game sessions.

3. **Session Management:** Each game session must be tracked separately, and users should be able to rejoin a session if they get disconnected.

4. **Performance Optimization:** The web-based version should run efficiently, regardless of the user's platform, ensuring fast load times and responsive interactions.

5. **Persistent Data:** Game and player data must be stored securely, allowing users to return to previous sessions or check their game history.

## [Design Constraints](#_heading=h.1ksv4uv)

Developing the game in a web-based environment introduces several design constraints:

1. **Platform Independence:** The game needs to run seamlessly across multiple platforms, including desktop and mobile environments (e.g., Windows, Mac, Linux, iOS, and Android). This requires using web technologies that are cross-compatible.

2. **Concurrency:** The system should support multiple teams and players interacting with the game simultaneously, which may introduce challenges in managing sessions and maintaining game state.

3. **Scalability:** The application must scale efficiently as more teams and players participate. This involves managing game instances, maintaining session data, and ensuring that server resources are optimized.

4. **Security:** Protecting user data and preventing unauthorized access is critical. Secure communication between users and the server, as well as encrypted storage of sensitive information, will be necessary.

## [System Architecture View](#_heading=h.44sinio)

1. **Client-Side (Presentation Tier):**

- The user interface (UI) will be delivered via web browsers and mobile devices, allowing players to interact with the game. This tier will handle all user inputs, display game data, and provide access to game functionality such as starting a game, joining teams, and guessing puzzle answers. The client-side will be built using standard web technologies such as HTML5, CSS, and JavaScript.

2. **Application Layer (Logic Tier):**

- This tier is responsible for managing the game logic, including player interactions, game state, team management, and puzzle rendering. The logic tier will use server-side technologies (e.g., Java or Node.js) to communicate between the client and the database. This tier handles all requests and enforces business rules and gameplay mechanisms.

3. **Data Layer (Storage Tier):**

- The data layer will store and manage persistent information, including player data, team details, game state, and puzzle libraries. A relational database (e.g., MySQL, PostgreSQL) or a NoSQL database (e.g., MongoDB) will be used to ensure that all game data is securely stored and can be retrieved efficiently. Data integrity and consistency are critical to ensure that the game state is maintained even in cases of concurrent users or disconnected sessions.

4. **Communication and Networking:**

- Communication between the client-side and the server will happen over a secure HTTP/S protocol. Real-time gameplay updates may be facilitated using WebSockets or server-sent events (SSE) to ensure low-latency updates for players as they guess puzzle answers or interact with team members. Data synchronization between teams and the server will also be key for maintaining a consistent game state across all platforms.

5. **Security Considerations:**

- User authentication and authorization will be implemented to protect sensitive user data and ensure that players only access their respective game sessions. Encryption mechanisms, such as HTTPS and secure data storage techniques, will be essential for preventing unauthorized access and ensuring a secure gaming environment.

## [Domain Model](#_heading=h.2jxsxqh)

**- ProgramDriver:** Responsible for launching the main application.

**- SingletonTester:** Used for testing the Singleton pattern within the application.

**- Entity Class:** Serves as the base class, containing common attributes like `id` and `name`, and methods such as `getId()`, `getName()`, and `toString()`. It is inherited by other key classes such as Game, Team, and Player, ensuring code reusability and adherence to object-oriented principles like inheritance.

**- GameService Class:** Follows the Singleton pattern to manage game-related services, ensuring only one instance of the class exists. It holds a list of games, with methods to add, retrieve, and count games, as well as manage player and team IDs.

**- Game, Team, and Player Classes:** Represent the main game components, each managing their respective entities (games, teams, players) and using object-oriented practices like composition to ensure relationships between the objects. For example, a Game contains a list of Teams, and a Team contains a list of Players.

**"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](#_heading=h.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** | Mac provides a stable and secure environment for hosting web applications, though it is less commonly used for large-scale deployments. The server hardware is expensive, but macOS offers good integration with development tools. | Linux is one of the most popular server environments due to its stability, scalability, and cost-effectiveness. It supports a wide range of open-source tools and is highly customizable. Its community support is a major strength, but less experienced developers may find it harder to manage. | Windows is a reliable server option, especially in enterprise environments. Windows Server supports many development languages and frameworks, but licensing costs are high compared to Linux. It provides good support for .NET applications and integrations with Microsoft technologies. | Mobile devices, while not directly suitable for hosting server applications, are crucial in a mobile-first development approach. Server-side interactions must be optimized for mobile clients through REST APIs or WebSockets, focusing on latency and bandwidth management. |
| **Client Side** | macOS supports rich web development environments and tools like Xcode and Swift, but has limited cross-platform tools natively. Web apps built on Mac browsers need to ensure compatibility with other OS environments. | Linux offers flexibility with multiple distributions and development environments. Client-side development on Linux can be challenging if cross-platform support (especially for Windows and Mac) is not prioritized. However, it is a powerful platform for web-based applications. | Windows dominates the desktop client space. Windows browsers (Edge, Chrome, Firefox) require web applications to perform well across a variety of hardware configurations. Development is straightforward, but ensuring cross-browser compatibility is critical. | Mobile client development requires the use of responsive design and optimization for various mobile devices and screen sizes. iOS and Android have distinct development environments (Xcode for iOS, Android Studio for Android), and both must be tested to ensure performance. |
| **Development Tools** | macOS developers typically use Xcode, Swift, and other Apple-specific tools. macOS also supports cross-platform tools like Visual Studio Code and Eclipse for broader web development, making it versatile for multiple platform support. | Linux developers use open-source tools such as Eclipse, VS Code, and text editors like Vim or Emacs. Linux's flexibility allows developers to customize their environment, and it's commonly used for server-side programming. Node.js, Python, and Java development are popular on Linux. | Windows developers often use Visual Studio, which is a powerful IDE for building cross-platform applications. Windows supports a wide range of programming languages, including C#, Java, JavaScript, and Python, providing versatility for client and server development. | Mobile app development relies heavily on platform-specific tools: Xcode for iOS and Android Studio for Android. Cross-platform mobile development can use frameworks like React Native, Flutter, or Xamarin, providing a unified codebase for both iOS and Android 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**: The recommended operating platform for *Draw It or Lose It* is **Linux**. Linux offers stability, scalability, and cost-efficiency, making it ideal for hosting web applications. Its open-source nature reduces licensing fees, and it provides robust support for server-side frameworks such as Node.js, Python, and Java. Additionally, Linux's ability to handle high concurrency is critical for a game with multiple teams and users interacting simultaneously. Linux can also be integrated with cloud services, making it a strong candidate for scaling the application as the user base grows.
2. **Operating Systems Architectures**: The architecture should follow a **client-server model** where the client (mobile/desktop) handles the user interface and interactions, while the server manages the game logic, databases, and communication between players. Linux servers will host the game logic, databases, and web services, while the client-side application will be delivered via web browsers or mobile devices. This separation ensures scalability, security, and ease of maintenance across platforms.
3. **Storage Management**:For storage management, a **relational database** such as MySQL or PostgreSQL is recommended. These databases offer reliable data consistency, support for complex queries, and strong transactional features, which are critical for managing game sessions, player profiles, and team data. To ensure performance, database indexing and optimization techniques will be used to handle large volumes of data efficiently. For high scalability, cloud storage solutions like Amazon RDS or Google Cloud SQL could be considered to manage data backups, recovery, and expansion seamlessly.
4. **Memory Management**: Memory management in Linux can be handled using efficient techniques such as **caching** to reduce the load on the database and server. Tools like Redis or Memcached can be used to store frequently accessed data in memory, speeding up gameplay responses for users. Additionally, garbage collection mechanisms and optimized code practices will be implemented to minimize memory leaks and optimize the game's performance, especially in high-concurrency environments.
5. **Distributed Systems and Networks**: To support *Draw It or Lose It* across various platforms, a distributed system will be implemented using cloud services. This system will allow multiple instances of the game to run concurrently. Technologies like **WebSockets** will be used for real-time communication between the server and client, allowing players to receive immediate updates during gameplay. The system must be fault-tolerant, using load balancers to distribute traffic evenly across servers to prevent downtime and ensure a smooth gaming experience, even during high demand.
6. **Security**: Security is paramount for *Draw It or Lose It*. All communication between the client and server should be encrypted using **HTTPS** to prevent man-in-the-middle attacks. User data, such as login credentials, will be hashed using secure algorithms (e.g., bcrypt or SHA-256). Additionally, proper access control mechanisms will be enforced to ensure that only authenticated users can join games. Regular security audits, including vulnerability scanning and penetration testing, should be conducted to identify and patch potential weaknesses in the system.