

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.1

## 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 4**](#_Toc115077323)

[**Domain Model 4**](#_Toc115077324)

[**Evaluation 6**](#_Toc115077325)

[**Recommendations 10**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | <07/19/24> | Tishana | Initial draft of the software design document |
| 1.1 | <08/04/24> | Tishana | Revision of Evaluation page |
| 1.2 | <08/18/24> | Tishana | Revision of Recommendations page |

**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 excited to announce the expansion of our popular game, "Draw It or Lose It," from an Android app to a web-based version that supports multiple platforms. This expansion will not only broaden our audience but also provide a seamless gaming experience across various devices, opening up new possibilities for our game. The challenge of adapting the game for a web-based environment is one we're eager to tackle, ensuring compatibility and preserving core mechanics.

The software design problem involves creating a web-based architecture that supports multiple teams, ensures unique games and team names, and maintains only one instance of the game in memory at any given time. Additionally, the game must handle real-time interactions and provide a responsive user experience.

Proposed Solution:

Web-Based Architecture: Develop a robust and scalable web-based architecture that can handle multiple platforms.

Unique Identifiers: Implement a system for creating unique identifiers for games, teams, and players to ensure no name conflicts and maintain a single game instance in memory.

Real-Time Interaction: Use efficient communication protocols to manage real-time interactions between players and teams.

User Experience: Ensure a responsive, user-friendly interface that mirrors the original Android app's experience.

By following this approach, The Gaming Room is confident that we can successfully transition "Draw It or Lose It" into a web-based game. This expansion will not only broaden our reach but also enhance the game experience for users across different platforms. This document provides a comprehensive plan to address the software requirements, design constraints, and development steps necessary to achieve this goal.

## Requirements

Software Requirements:

1. Team Involvement: The game must support the participation of one or more teams.
2. Multiple Players per Team: Each team should have the capability to include multiple players.
3. Unique Game and Team Names: Game and team names must be unique, allowing users to verify the availability of a name when selecting a team name.
4. Single Instance Management: Only one instance of the game can exist in memory at any given time. This will be achieved by creating unique identifiers for each instance of a game, team, or player.

## [Design Constraints](#_2et92p0)

Scalability:

* Constraint: The application must support an increasing number of users and teams efficiently.
  + Implication: The architecture must scale horizontally by adding servers or instances as needed.

Real-Time Interaction:

* Constraint: Players and teams need to interact in real-time.
  + Implication: Implementing WebSocket or similar technologies to facilitate real-time communication and updates.

Data Consistency and Uniqueness:

* Constraint: Game, team, and player names must be unique, and only one instance of the game should exist at any time.
  + Implication: Use a central database to enforce unique constraints and manage game instances using a singleton pattern.

Cross-Platform Compatibility:

* Constraint: The game must be accessible from various devices and platforms (desktop, mobile, etc.)
  + Implication: Develop a responsive web interface using modern web technologies like Mac, Linux, Windows, and mobile devices.

Performance:

* Constraint: The application must perform well under load, rendering images and processing user input quickly.
  + Implication: Optimize server responses and client-side rendering to maintain a responsive user interface.

Maintainability:

* Constraint: The codebase must be maintainable and easy to update.
  + Implication: Use modular coding practices, proper documentation, and version control to ensure the application is maintainable.

## [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)

1. Entity Class:
   1. Attributes: ‘id’ (long), ‘name’ (String)
   2. Methods: ‘Entity()’, ‘Entity(id: long, name: String)’, ‘getId(): long’, ‘getName(): String’, ‘toString(): String’
   3. Description: This is a base class that holds common attributes (‘id’ and ‘name’) and behaviors for other classes.
2. GameService Class:
   1. Attributes: ‘games’ (List<Game>), ‘nextGameId’ (long), ‘nextPlayerId’ (long), ‘nextTeamId’ (long), 'service’ (GameService)
   2. Methods: ‘GameService()’, ‘getInstance(): GameService’, ‘addGame(name: String): Game’, ‘getGame(id: long): Game’, ‘getGame(name: String): Game’, ‘getGameCount() : int’, ‘getNextPlayerId(): long’, ‘getNextTeamId(): long’
   3. Description: This class manages the game instances. It follows the singleton pattern, ensuring only one instance of GameService exists at any given time.
3. Game Class:
   1. Attributes: ‘teams’ (List<Team>)
   2. Methods: ‘Game(id: long, name: String)’, ‘addTeam(name: String): Team’, ‘toString(): String’
   3. Description: Represents a game, which contains multiple teams. Inherits from the Entity class.
4. Team Class:
   1. Attributes: ‘players’ (List<Player>)
   2. Methods: ‘Team(id: long, name: String)’, ‘addPlayer(name: String): Player’, ‘toString(): String’
   3. Description: Represents a team, which contains multiple players. Inherits from the Entity class.
5. Player Class:
   1. Attributes: ‘Player(id: long, name: String)’
   2. Methods: ‘toString(): String’
   3. Description: Represent a player in the game. Inherits from the Entity class.
6. ProgramDriver Class:
   1. Methods: ‘main()’
   2. Description: The entry point of the application, which uses the GameService class.
7. SingletonTester Class:
   1. Methods: ‘testSingleton()’
   2. Description: Used to test the singleton implementation of the GameService class.

Object Oriented Programming Principles

1. Encapsulation:
   1. Each class encapsulates its attributes and provides methods to interact with those attributes, ensuring data hiding and abstraction. For example, ‘Game’, ‘Team’, and ‘Player’ classes all encapsulate their properties and behaviors.
2. Inheritance:
   1. The ‘Game, ‘Team’, and ‘Player’ classes are inherited from the ‘Entity’ base class. This allows them to share common attributes (‘id’ and ‘name’) and methods (‘getId()’, ‘getName()’, and ‘toString()’), promoting code reuse and reducing redundancy.
3. Singleton Pattern:
   1. The ‘GameService’ class follows the singleton pattern. The ‘getInstance()’ method ensures that only one instance of ‘GameService’ exists at any time. This is critical for managing the game state globally across the application.
4. Composition:
   1. The ‘Game’ class contains a list of ‘Team’ objects, and the ‘Team’ class contains a list of ‘Player’ objects. This represents a “has-a” relationship, where a game has teams, and a team has players. This composition allows for building complex structures from simpler objects.
5. Abstraction:
   1. The ‘Entity’ class provides a level of abstraction by defining common properties and behaviors that are shared among ‘Game’, ‘Team’, and ‘Player’ classes. This allows focusing on more specific attributes and behaviors in the subclasses.

Fulfillment of Software Requirements

* Multiple Teams and Players: The ‘Game’ class can have multiple ‘Team’ objects, and each ‘Team’ can have multiple ‘Player’ objects, satisfying the requirement for supporting multiple teams and players.
* Unique Names and Single Instance: The ‘GameService’ class, with its singleton pattern, ensures only one instance of the game exists in memory. It also manages unique identifiers for games, teams, and players, ensuring name uniqueness.
* Real-Time Interaction: The diagram lays the groundwork for managing game state and interactions in real-time through the central ‘GameService’ class.
* User Experience: By organizing the structure clearly and promoting code reuse through inheritance and composition, the design supports a responsive and maintainable user interface.

**"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** | Macs are known for their stability and performance but are not commonly used as servers. They offer a user-friendly interface and robust security features, making them suitable for development environments rather than large-scale production servers. The high security and seamless integration with other Apple products are significant advantages. | Linux is a popular server choice due to its flexibility, stability, and open-source nature. It supports a wide range of server applications and services, which makes it highly customizable. The extensive community support and no licensing fees are significant advantages. | Windows servers are widely used and offer strong integration with other Microsoft products. They provide a familiar environment for many developers and administrators, which can facilitate ease of use and support. Microsoft's comprehensive documentation and support are a considerable advantage. | Mobile Devices typically act as clients rather than servers. However, developing mobile-friendly web applications requires consideration of performance limitations and varying device capabilities. The primary advantage is the high portability and accessibility, allowing users to engage with the application on the go. |
| **Client Side** | Developing client-side applications for Mac requires consideration of cost, time, and expertise. The high initial cost of Mac hardware and software can be a barrier, but the development tools are sophisticated and well-integrated. Developers must be familiar with macOS development environments and tools like Xcode. The seamless user experience and integration with other Apple devices are significant benefits, though the learning curve for development tools can add to development time. | Client-side development on Linux offers the advantage of being cost-effective due to its open-source nature. However, it requires strong technical skills and familiarity with the Linux environment. Development can be efficient with the right expertise, but the initial setup and configuration may take longer. The flexibility and customization options are strong points, though potential compatibility issues with specific software must be managed. | Windows provides a familiar environment for many developers, which can speed up development time. The integration with other Microsoft products and comprehensive support are advantages. However, the moderate to high Windows licenses and hardware costs must be considered. Developers should be proficient with Windows development tools and environments like Visual Studio to optimize development. | Development for mobile devices involves considerations for different platforms (iOS, Android) and the need for responsive design. The costs can vary depending on the target platform and devices. Expertise in mobile development and knowledge of platform-specific SDKs (Android SDK, iOS SDK) and tools like Android Studio or Xcode are essential. Testing across various devices and ensuring performance optimization are critical factors that add to the time and complexity of development. |
| **Development Tools** | Developing software for deployment on Mac typically involves using programming languages like Swift, Objective-C, JavaScript, Python, and Java. Essential development tools include Xcode, Visual Studio Code, and IntelliJ IDEA. The Cocoa framework, Docker, and Git are commonly used. Developers must be proficient in these languages and tool to ensure efficient development and deployment on Mac. The must be proficient in Mac-specific tools and languages. Mac development typically requires a good understanding of macOS environment and Xcode. May require a specialized team for macOS development. Xcode is free but requires Mac hardware which can be expensive. Some tools licensing fees, but many are open source. | Linux development often utilizes programming languages like Python, Java, C++, JavaScript, and PHP. Commonly used IDEs include Eclipse, Visual Studio Code, and IntelliJ IDEA. Tools like Docker, Git, Jenkins, and Kubernetes are integral to development. Familiarity with these languages and tools is crucial for effective Linux development and deployment. Requires a team with strong technical skills and familiarity with Linux environment. Development can be efficient with the right expertise. May necessitate a dedicated team for Linux-based application. Generally, Linux tools are open-source and free. No licensing fees for the operating system. | Languages like C#, .NET, JavaScript, Python, and Java are commonly used for Windows development. Popular development environments include Visual Studio, Visual Studio Code, Eclipse, and IntelliJ IDEA. Tools like Docker, Git, Jenkins, and Azure DevOps support development. Developers must be well-versed in these technologies to deploy software on Windows successfully. Development team must be familiar with Windows development tools and environments like Visual Studio. Often requires a team with experience in Windows-specific technologies. Windows has moderate to high licensing costs for both the OS and some development tools like Visual Studio (though there are free versions). | Mobile devices require knowledge of languages such as Swift (iOS), Kotlin (Android), JavaScript, HTML5, and CSS3. IDEs like Xcode (for iOS) and Android Studio (for Android) are essential, along with cross-platform tools like Xamarin, React Native, and Flutter. Proficiency in these languages and tools is necessary to build and deploy robust mobile applications. Requires expertise in mobile development and platform-specific SDKs (iOS SDK, Android SDK). May need separate teams for iOS and Android development or a cross-platform development team using tools like Xamarin or React Native. Xcode for iOS is free, but requires Mac hardware. Android Studio is free. Cross-platform tools like Xamarin may have licensing costs. |

## Recommendations

The following recommendations address the necessary considerations for The Gaming Room to ensure a seamless expansion of "Draw It or Lose It" into a web-based version that supports multiple platforms.

1. **Operating Platform**: Linux emerges as the prime choice for the operating platform. Its widespread use in web-based applications is a testament to its flexibility, stability, and open-source nature. The absence of licensing fees and the ability to customize it to support various server applications and services make it a cost-effective solution. Its robust security features and extensive community support further solidify its position as the ideal platform for developing and hosting web-based applications.
2. **Operating Systems Architectures**: Linux operating systems use a modular architecture, allowing customization and optimization based on the application's specific needs. The Linux kernel is at the core, managing system resources and providing essential services. On top of the kernel are various distributions (distros) like Ubuntu, CentOS, and Debian, which offer different packages and tools to support development. The architecture is highly scalable, enabling it to handle varying loads and ensuring high availability.
3. **Storage Management**: Our recommendation for storage management is a comprehensive one. We propose a hybrid approach that combines the strengths of both relational and non-relational databases. For relational data storage, we suggest using MySQL or PostgreSQL, which ensure data integrity and support complex queries. For non-relational data storage, we recommend MongoDB or Redis, which provide scalability and flexibility. This approach efficiently handles data types and access patterns, ensuring optimal performance and reliability.
4. **Memory Management**: Linux uses several techniques, including paging, segmentation, and virtual memory. It efficiently allocates memory to processes and manages swapping to disk when physical memory is low. For "Draw It or Lose It," leveraging Linux's advanced memory management ensures the application can handle multiple concurrent users and maintain performance. Techniques like caching frequently accessed data in memory can also improve responsiveness and reduce server load.
5. **Distributed Systems and Networks**: Recommend using a microservices architecture with RESTful APIs to enable communication between various platforms. This approach allows different components of the game to interact seamlessly over the network. Using Docker for containerization and Kubernetes for orchestration can further enhance scalability and manageability. To ensure reliable connectivity, implement load balancers and redundancy. Address potential outages by designing a system for fault tolerance and implementing robust monitoring and alerting systems.
6. **Security**: Security is not just a consideration, it's a priority. To safeguard user information, we recommend implementing SSL/TLS encryption for data in transit between clients and servers and using encryption for data at rest in databases. Secure coding practices should be employed to prevent vulnerabilities like SQL injection and cross-site scripting (XSS). User authentication and authorization mechanisms using OAuth or similar protocols should be implemented. Regular updates and patches to the operating system and application components are crucial to mitigate security risks. Additionally, firewalls, intrusion detection systems, and regular security audits should be used to ensure comprehensive protection.