

**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 | 7/18/2024 | Reginald Cooper | Test Creation and completion of the document |

## [Executive Summary](#_sbfa50wo7nsh)

This presents the design for a game application for our client, The Gaming Room. The game, "Draw It or Lose It," will be developed as a web-based application. This outlines the critical aspects of the design, including constraints, the domain model, and an evaluation of potential operating platforms. The goal is to make sure that the game operates efficiently and securely across various environments while meeting all business and technical requirements

## Requirements

The game application must support multiplayer functionality, making sure the players can interact with the game from different devices and locations. It must provide a smooth user experience with minimal disruption. Security is primary, protecting user data and making sure communication is secure between clients and the server.

## [Design Constraints](#_2et92p0)

1. Network disruptions: The application must handle varying network speeds and ensure a smooth user experience.
2. Scalability: The design should allow easy scaling to handle increased user load.
3. Security: The application must protect user data and make sure there is secure communication.
4. Cross-Platform Compatibility: The application should function consistently across different operating systems and devices.

## [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)

GameService

Attributes: games, nextGameId, instance

Methods: getInstance(), addGame(), getGame(), getGameCount()

Here we manage the collection of games and provide methods to add and retrieve games. It uses the Singleton pattern to ensure only one instance exists.

Game

Attributes: id, name

Methods: getId(), getName(), toString()

This is an individual game with an ID and name.

ProgramDriver

Methods: main()

The entry point for the application.

SingletonTester

Methods: method(type)

This tests the Singleton implementation.

The relationships are as follows:

GameService contains a list of Game objects, indicating a one-to-many relationship.

ProgramDriver uses GameService.

SingletonTester is used to test the Singleton pattern in GameService.

Object-oriented programming principles demonstrated include:

Encapsulation: Each class encapsulates its attributes and provides public methods for interaction.

Inheritance: This is implied because common methods are shared.

Polymorphism: Methods like getGame() demonstrate polymorphism by being overloaded to accept different parameters.

Singleton Pattern: Ensures only one instance of GameService is created, providing a global access point.

**"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** | Reliable but less common for server hosting.  UPDATE:  **Characteristics**: Reliable but less common for server hosting.  **Advantages**: Strong security, seamless integration with other Apple products, robust development tools.  **Weaknesses**: Higher costs for hardware, less common in server environments, requires Mac-specific expertise.  **Costs**: Higher hardware costs | Highly reliable and widely used for server hosting.  UPDATE:  **Characteristics**: Highly reliable and widely used for server hosting.  **Advantages**: Open-source, cost-effective, highly secure, scalable, and supports a wide range of server configurations.  **Weaknesses**: Requires Linux-specific expertise for management and troubleshooting.  **Costs**: Generally free (open source), enterprise support may costs. | Common for enterprise applications but can be more expensive  UPDATE:  **Characteristics**: Common for enterprise applications.  **Advantages**: Strong enterprise support, widespread use in business environments, extensive documentation and support.  **Weaknesses**: Higher licensing costs, more vulnerable to security threats compared to Linux.  **Costs**: License fees for Windows Server, possibly additional costs for other Microsoft services. | |  | | --- | | Not enough space  for hosting,  usually used  as clients.  UPDATE: |  |  | | --- | |  |   **Characteristics**: Not typically used for hosting but crucial for client access.  **Advantages**: High demand for mobile access, large user base, supports modern, responsive design.  **Weaknesses**: Not designed for server hosting, additional development required for mobile interfaces.  **Costs**: Development tools may have associated costs (Apple Developer Program, Android SDK). |
| **Client Side** | Requires Mac-specific expertise, higher development costs.  UPDATE:  **Development Things to consider:** Requires expertise in macOS and iOS development. Tools like Xcode are essential.  **Cost**: Higher development costs due to specialized expertise and hardware requirements.  **Compatibility**: Making sure the application is compatible with Safari and other Mac-specific browsers. | |  | | --- | | Requires Linux-  specific expertise.  lower  development  costs.  UPDATE: |  |  | | --- | |  |   **Development Things to consider:** Requires Linux-expertise. Development tools like GCC, G++, and IDEs like Eclipse and VS Code are used.  **Cost**: Lower development costs, primarily due to open-source nature.  **Compatibility**: Making sure compatibility with various Linux distributions and browsers like Firefox and Chromium. | |  | | --- | | Requires  Windows-specific  expertise.  moderate  development  costs.  UPDATE: |  |  | | --- | |  |   **Development Things to consider:** Requires Windows Expertise. Tools like Visual Studio and .NET are common.  **Cost**: Somewhat expensive development costs.  **Compatibility**: Making sure compatibility with Internet Explorer/Edge and other Windows browsers. | |  | | --- | | High demand for  mobile-friendly  interfaces and  touch support.  UPDATE: |  |  | | --- | |  |   **Development Things to consider:**  Expertise in Swift and Objective-C for iOS; Java and Kotlin for Android. Cross-platform tools like React Native can be used.  **Cost**: Additional costs for maintaining multiple codebases or investing in cross-platform development tools.  **Compatibility**: Making sure responsive design and compatibility across various mobile devices and screen sizes. |
| **Development Tools** | |  | | --- | | Swift,  Objective-C  for iOS apps. |  |  | | --- | |  |   UPDATE:  **Tools**: Xcode  **Impact on Development Team:** Requires specialized expertise, potential need for separate teams for macOS/iOS development.  **Costs**: Apple Developer Program fees. | |  | | --- | | GCC, G++,  IDEs like  (Eclipse, VS Code)  for development. |  |  | | --- | |  |   UPDATE:  **Tools**: The same.  **Impact on Development Team:** Open-source tools reduce costs but may need Linux-specific knowledge.  **Costs**: Usually free. | |  | | --- | | Visual Studio,  .NET,  various IDEs  for development. |  |  | | --- | |  |   UPDATE:  **Languages**: C#, .NET, JavaScript  **Tools**: Visual Studio  **Impact on Development Team**: Requires Windows expertise, potentially separate teams for Windows development.  **Costs**: Costs for Visual Studio and Windows licenses. | Android Studio, Xcode, React Native for cross-platform solutions.  UPDATE:  **Languages**: Swift, Objective-C (iOS); Java, Kotlin (Android); JavaScript/TypeScript (React Native)  **Tools**: Android Studio, Xcode, React Native  **Impact on Development Team:** May need multiple teams or cross-platform development tools.  **Costs**: Apple Developer Program, Android development tools generally free. |

## Recommendations

1. **Operating Platform**: Linux is recommended for its reliability, scalability, and cost-effectiveness for server hosting.
2. **Operating Systems Architectures**: Utilize a microservices architecture on Linux servers to make sure modularity and scalability.
3. **Storage Management**: Use a combination of relational databases (PostgreSQL) and NoSQL databases (MongoDB) for efficient data storage and retrieval
4. **Memory Management**: It has tool for efficient memory management techniques such as garbage collection and memory pooling to optimize performance
5. **Distributed Systems and Networks**: It has a robust API for communication between clients and servers, using WebSockets for real-time interactions and REST for standard requests.
6. **Security**: Uses encryption (TLS/SSL) for data transmission, secure authentication mechanisms (OAuth), and regular security audits to protect user data.

UPDATE:

Based on the update evaluation, here is the recommendations for The Gaming Room:

1. Server Hosting: Utilize Linux for its reliability, scalability, and cost-effectiveness.

2. Client Development: Invest in cross-platform development tools like React Native to streamline development for both mobile platforms and ensure responsive design for desktop browsers.

3. Development Tools: Leverage open-source tools where possible to reduce costs, and ensure teams have the necessary expertise for each platform.

4. Security and Scalability: Implement robust security measures (TLS/SSL, OAuth) and design for scalability to handle thousands of players efficiently.

My Final Update Evaluation:

I recommend using **Linux** as the operating platform due to its reliability, scalability, and cost-effectiveness. Linux is a widely adopted choice in server environments, particularly for applications that demand strong security and the ability to scale efficiently. Being open-source, Linux also offers a cost advantage and benefits from a robust community that supports its continuous improvement.

As for the operating systems, I think **Linux** is well-suited for **a microservices architecture**. This architecture enables different components of the "Draw It or Lose It" game, such as the game engine, user management, and chat service, to be developed, deployed, and scaled independently. Each service operates as a separate process, communicating with other services through lightweight protocols like HTTP/HTTPS or message queues. This approach is especially beneficial for a multiplayer game that needs to handle varying loads efficiently, guaranteeing a smooth and scalable experience.

For storage management, I recommend using a combination of databases like **PostgreSQL and NoSQL databases** such as MongoDB. PostgreSQL is ideal for managing structured data and handling complex queries, making sure data integrity and supporting ACID transactions, which are important for maintaining consistency in player scores, game states, and user profiles. On the other hand, MongoDB is perfect for handling unstructured data and provides the flexibility needed to store dynamic data, such as in-game messages or real-time game data.

When it comes to memory management, **Linux** offers advanced techniques like garbage collection and memory pooling, which are essential for optimizing performance. **Garbage collection** automatically reclaims memory occupied by objects that are no longer in use, which is critical in a gaming environment to prevent memory leaks. Memory pooling further enhances efficiency by reusing memory for objects that are frequently allocated and deallocated, reducing the overhead associated with memory management.

To make sure that "Draw It or Lose It" can communicate effectively between various platforms, I recommend using a distributed system architecture. This can be done by using **RESTful APIs** for standard client-server communication and **WebSocket’s** for real-time interactions between players. WebSocket’s provide a full-duplex communication channel over a single, long-lived connection, which is crucial for real-time gaming. The network connecting these systems should be designed to handle high availability, load balancing, and fault tolerance, making sure a seamless gaming experience even during peak usage times.

For security, I believe it should be a top priority for The Gaming Room. I recommend using encryption protocols like **TLS/SSL** to secure data transmission between clients and servers. Also, using **OAuth** for secure authentication and access control will make sure that only authorized users can access game data and services. It's also important to conduct regular security audits and vulnerability assessments to identify and mitigate potential threats. To protect the system even more, I suggest using firewall protection and **intrusion detection systems (IDS)** to guard against network-level attacks.

My recommendations are as follows:

1. **Operating Platform**: I suggest using Linux for server hosting due to its reliability, scalability, and cost-effectiveness.

2. **Operating Systems Architecture**: I recommend using a microservices architecture on Linux servers to make sure modularity and scalability.

3. **Storage Management:** I advise utilizing PostgreSQL and MongoDB for efficient and flexible data management.

4. **Memory Management:** I recommend leveraging Linux’s memory management techniques, such as garbage collection and memory pooling, to optimize performance.

5. **Distributed Systems and Networks:** I suggest using RESTful APIs and WebSocket’s for effective communication between platforms in a distributed system.

6. **Security:** I strongly recommend making sure robust security with TLS/SSL encryption, OAuth for authentication, and regular security audits.