

The Gaming Room

# **CS 230 Project Software Design Template**

Version 1.0

## Table of Contents

**[CS 230 Project Software Design Template](#_Toc115077317)** [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 | 09/17/2024 | Dylan Wienecke | Revised Executive Summary & Requirements |
| 1.1 | 09/19/2024 | Dylan Wienecke | Added Constraint 1 & 2 |
| 1.2 | 09/20/2024 | Dylan Wienecke | Added Constraint 3 & 4 |
| 1.3 | 09/21/2024 | Dylan Wienecke | Added Domain Model Explanation |
| 1.4 | 09/30/2024 | Dylan Wienecke | Updated Server-Side Evaluations |
| 1.5 | 10/01/2024 | Dylan Wienecke | Updated Client-Side Evaluations |
| 1.6 | 10/04/2024 | Dylan Wienecke | Updated Development Tools |
| 1.7 | 10/05/2024 | Dylan Wienecke | Added Detailed Explanation of Bullet Points |
| 1.8 | 10/15/2024 | Dylan Wienecke | Outlined Platform and Architecture Recommendation |
| 1.9 | 10/16/2024 | Dylan Wienecke | Outline Storage and Memory Management |
| 2.0 | 10/18/2024 | Dylan Wienecke | Outlined Systems and Networks & Security |
| 2.1 | 10/20/2024 | Dylan Wienecke | Added Nginx Example to Networks Section |

**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 wishes to build a web-based application for their Draw It or Lose It game. The Draw It or Lose It application will address several key requirements requested by the client. The framework will be a web-based environment capable of maintaining consistent instances of games. Each instance will host one or more teams, comprised of multiple players. Each game, team, and player will have unique names to avoid conflicts. The environment will also possess a user friendly interface to manage gameplay where players guess images rendered from a stock library of drawings. The system must be able to fully render images at the 30 second mark during gameplay.

Our solution will focus on building a distributed system that supports the functionality of Draw It or Lose It. Considering a suitable architecture while prioritizing maintainability and efficiency will support future expansions and updates.

## Requirements

*<* Please note: While this section is not being assessed, it will support your outline of the design constraints below. *In your summary, identify each of the client’s business and technical requirements in a clear and concise manner.>*

* *Support for multiple teams*
* *Support for multiple players*
* *Maintain a single instance of a game in memory using unique identifiers*
* *Render stock images from a library within 30 seconds*
* *Implement a turn-based structure to allow non-active teams to guess after a 15 second window*

## [Design Constraints](#_2et92p0)

1. **Web-based Environment**
   1. A server-client architecture will be required to facilitate the distributed system that meets the functionality requirements specified. Real-time communication between server and players must be maintained during gameplay. This will ensure players can interact while data remains synced across each instance.
   2. This constraint means latency must be minimized during gameplay and the server is optimized for scalability to handle multiple games or teams concurrently.
2. **Unique Identifiers**
   1. Unique names and identifiers for each game, teams, and players will require a database constraint and strict memory management. The system will need to generate and verify unique identifiers in real time.
   2. This constraint will require careful design of the data model and key constraint management to ensure integrity is maintained to avoid conflicts. A relational database or a NoSQL system will be suitable depending on performance requirements.
3. **Single Game Instance in Memory**
   1. A single instance of a game can be active in memory at a time. This constraint implies the need for singleton design patterns to be implemented to prevent duplicate instances.
   2. Strict memory management and monitoring of game state is required to avoid concurrency issues. This will properly manage game states across multiple players and teams.
4. **Security and Data Integrity**
   1. Due to the fact the Draw It or Lose It game incorporates multiple users in a distributed environment, security and data integrity is crucial to functionality. Unauthorized users and tampering with a games state have to be carefully monitored. This constraint will prevent intercepted communication between the server and the client or cheating.
   2. This constraint will require implementing secure communications protocols such as HTTPS or WebSockets with encryption. Data integrity measures will be implemented to ensure the game state is correctly synchronized to prevent tampering.

## [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 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.**

1. **Entity**
   1. The Entity class is a parent class of Game, Team, and Player. It is used to define common attributes and methods shared across all child classes. This relationship allows avoidance of code duplication and demonstrates the OOP principle of inheritance and polymorphism.
2. **Game**
   1. Each Game contains a list of teams demonstrating a composition relationship where teams are integral to the game. Game inherits from Entity.
3. **Team**
   1. Each Team contains a list of players demonstrating a composition relationship where players are integral to the Team. Team inherits from Entity.
4. **Player**
   1. Inherits from Entity and is associated with Team forming a team-player relationship. Players exist as part of Team objects.
5. **GameService**
   1. GameService manages all Game objects in the form of a list. As a singleton class, there can only be a single instance of GameService in memory at any time. This helps orchestrate the life cycle of Game objects and associated classes used to facilitate the desired functionality.
6. **ProgramDriver**
   1. The ProgramDriver class uses GameService class to execute game logic in the main() method. This is the entry point of the program.
7. **SingletonTester**
   1. This class is used to test for an accurate singleton behavior by the GameService class object.

**Object-Oriented Principles &**

The Game, Team, and Player classes inherit from the Entity super class, which defines common attributes such as id, and name. This demonstrates the inheritance principle by allowing the behaviors of the parent class to be accessed by child classes without repeating code and simplifies future extensions. Encapsulation is demonstrated in each classes private attributes and public access methods. This design promotes data integrity and safeguards the internal state of each object. The polymorphism principle is primarily represented in the GameService class via method overloading. The getGame() functions can be called with different parameters indicating compile-time polymorphism. The Entity class, while not explicitly defined as an abstract class, facilitates the abstraction principle by hiding the specific implementation details of the Game, Team, and Player classes by providing a common interface such as the getID() and getName() methods. This design also supports easier extension of the system with new entities definitions in the future.

All of these principles help meet the requirements outlined by The Gaming Room. By using shared Entity classes with an id field, the system ensures all derived objects have a unique identifier critical to the game application. The singleton pattern of GameService ensures that only one instance of this service manages the game objects, which aligns with the requirement of only one instance of the game can exist in memory at any time. The composition relationship between Game, Team, and Player efficiently supports the requirement that each game has multiple teams and each team has multiple players. The UML diagram represents well established object-oriented approaches to system design and meets the businesses requirements for the game application.

## [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.

Below the chart there is a more detailed breakdown of the bullet points for each operating platform.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Advantages:   * Compatible with Apache, Nginx, Node.js * Unix based * Stable and secure   Weaknesses:   * Higher hardware costs * Poor optimization * Lack of support for server-side focus * May require 3rd party support for hosting(MacStadium) | Advantages:   * Compatible with Apache, Nginx, Node.js, Lighttpd, Windows Server, and more * Compatible with MySQL, PostgreSQL, and other database software * Highly optimized for performance, security, and scalability(suitable for server hosting) * Free and open-source(no licensing) * Support for cloud hosting(AWS, Google Cloud, Azure, etc)   Weaknesses:   * High degree of technical knowledge required to operate | Advantages:   * Compatible with Apache, Nginx, Node.js, and Windows Server * Compatible with MySQL or Microsoft SQL Server * Wide range of compatible hardware * Integrates well with other Microsoft enterprise tools * Support for cloud hosting(AWS, Google Cloud, Azure, etc)   Weaknesses:   * Higher resource consumption * Requires licensing fees * Higher hardware requirements due to resource consumption | Advantages:   * Suitable for testing lightweight server environments(using Node.js)   Weaknesses:   * Limited hardware capabilities (power, processing, network capability) * High hardware costs |
| **Client Side** | * Stable and secure * Higher hardware costs * User friendly tools (Xcode, CLion) * Apple Developer Program(Required to distribute applications) * HTML, CSS, JavaScript compatible * Swift or Objective-C for native clients | * Compatible with nearly all development tools and a wide range of hardware * HTML, CSS, JavaScript compatible * May require command line functions for version control(Git) and package manager knowledge * May require administration skills to optimize development environment | * Less expensive than Mac * HTML, CSS, JavaScript compatible * C# or .NET framework for native clients * Licensing fees for Visual Studio Professional(optional) * Visual Studio supports a wide range of languages * Licensing Fees for each machine running Windows | * Requires Apple Developer Program to publish for iOS * Requires one time fee to publish on Google Play * Additional fees for testing on multiple devices(iPhone, iPad, Android phones and tablets) |
| **Development Tools** | Languages:   * CSS * HTML * JavaScript * MySQL or NoSQL * Objective-C * Swift   Tools:   * Xcode * CLion * React * Apache * Nginx * Node.js | Languages:   * CSS * HTML * JavaScript * MySQL or NoSQL   Tools:   * Nearly any IDE (VS Code, JetBrains products, etc) * React * Vue.js * Angular * Apache * Nginx * Node.js * Lighttpd * Windows Server * AWS, Google Cloud, Azure | Languages:   * CSS * HTML * JavaScript * MySQL or NoSQL * C#   Tools:   * Most IDEs(Visual Studio, JetBrains products, Eclipse, etc) * Apache * Nginx * Node.js * Windows Server * AWS, Google Cloud, Azure | Languages:   * CSS * HTML * JavaScript * MySQL or NoSQL   Tools:   * Android Studio * Xcode |

**Mac:**

**Server Side:**

* **Capabilities:** MacOS is compatible with popular web server software - such as, Apache, Nginx, and Node.js. Mac also has a Unix foundation similar to Linux making it capable of hosting a distributed environment. Mac is also compatible with MySQL for database requirements. However, macOS is not optimized for large-scale hosting.
* **Costs:** MacOS requires Apple hardware to run. Apple machines are considerably more expensive than general PCs running other operating systems. While there is no direct licensing fee to utilize macOS, the cost of Apple specific hardware would present a significant barrier for hosting purposes.
* **Additional Components:** Since macOS is not specifically tailored for server use, additional enhancements may be required to meet specified functionality. This would require network and server management expertise on the development team and/or third-party licensing to cloud services like MacStadium, which would add to costs.

**Client Side:**

* **Costs:** Native macOS applications also require Apple hardware. This greatly increases the cost of developing a client on macOS. The Apple Developer Program required to publish in the Apple Store will also require an additional fee.
* **Language and Tools:** MacOS applications utilize Swift or Objective-C which can be developed in Xcode, Apple’s proprietary development environment, or JetBrains related products like CLion. However, web-based applications compatible with Apple machines can be based in HTML, CSS, or JavaScript.
* **Skills Required:** Developers will need proficiency in front-end web development technologies like HTML, CSS, JavaScript as well as macOS-specific tools like Swift and Xcode to meet native application requirements.

**Development Tools:**

Languages:

* CSS
* HTML
* JavaScript
* MySQL or NoSQL
* Objective-C
* Swift

Tools

* Xcode
* CLion
* React
* Apache
* Nginx
* Node.js

**Linux:**

**Server Side:**

* **Capabilities:** Linux virtually support all web server software (Apache, Nginx, Node.js, Lighttpd, Windows Server, etc.). Linux is compatible with MySQL, PostgreSQL, and other database systems ideal for large-scale web applications. Linux is highly optimized for performance, security, and scalability suitable for server hosting.
* **Costs:** Linux is free and open-source, making it a cost-effective solution for web hosting. The lack of licensing fees reduces overall server costs, but does require more technical expertise to reach the desired functionality. Third-party support tools for hosting like AWS, Google Cloud, and Azure offer numerous affordable hosting options for Linux.
* **Additional Components:** Linux may require additional components like database management systems (MySQL, PostgreSQL) or caching solutions (Redis, Memcached). However, given the number of Linux distributions such as Ubuntu, CentOS, or Debian systems, finding a distribution specific to your needs may cover most of the requirements without the need for additional components.

**Client Side:**

* **Costs:** Due to the open-source nature of Linux, almost all development tools and hardware are compatible with this platform. This compatibility also includes tools for web development making the cost for developing a client on Linux virtually free.
* **Language and Tools:** Client-side development on Linux will primarily utilize HTML, CSS, and JavaScript, along with front-end frameworks like React, Vue.js, or Angular.VS Code, Atom, and Sublime Text are commonly used IDEs compatible with Linux, which are free and open-source.
* **Skills Required:** Developers will need sufficient knowledge of web technologies (HTML, CSS, JavaScript) and have familiarity with Linux environments. This will include using command line functions for tasks like version control(Git) and package management. Additionally, due to the high level of customization in Linux, developers may require basic system administration skills to optimize their development environments.

**Development Tools:**

Languages:

* CSS
* HTML
* JavaScript
* MySQL or NoSQL

Tools:

* Nearly any IDE (VS Code, JetBrains products, etc)
* React
* Vue.js
* Angular
* Apache
* Nginx
* Node.js
* Lighttpd
* Windows Server
* AWS, Google Cloud, Azure

**Windows:**

**Server Side:**

* **Capabilities:** Windows is compatible with most popular web server software (Apache, Nginx).Windows supports multiple database systems like Microsoft SQL Server and MySQL but can be resource intense despite optimization which can impact performance at scale.
* **Costs:** Windows requires licensing fees which will drive developments costs up. Windows Server would be an additional licensing fee on top of the base platform making it a less desirable choice for database management. If a cloud based environment, such as Azure is considered, this can drive costs up exponentially as projects scale. Especially if multiple servers are required to manage data.
* **Additional Components:** Windows Server integrates well with other Microsoft enterprise tools(e.g., SQL Server or Active Directory), which may require additional licensing fees. Due to higher resource consumption during runtime, higher-end hardware may be required. Windows does provide extensive support for management tools and automating system management, which can potentially offset maintenance costs in the future.

**Client Side:**

* **Costs:** Client-side development would require Windows licensing for all relevant machines. This may extend to other Microsoft enterprise tools like Visual Studio professional version. However, Windows is generally less expensive than Mac with better compatibility across types of machines.
* **Language and Tools:** Web applications developed on Windows typically utilize HTML, CSS, and JavaScript. Windows-native clients would include languages like C# or VB.NET, which are part of the .NET framework. Visual Studio is the primary IDE used on Windows, with support for a wide range of languages and tools needed for native development and cross-platform.
* **Skill Required:** Proficiency in web technologies and, if developing Windows-native applications, must know C# or VB.NET. Familiarity with Visual Studio and debugging Windows applications will be crucial.

**Development Tools:**

Languages:

* CSS
* HTML
* JavaScript
* MySQL or NoSQL
* C#

Tools:

* Most IDEs(Visual Studio, JetBrains products, Eclipse, etc)
* Apache
* Nginx
* Node.js
* Windows Server
* AWS, Google Cloud, Azure

**Mobile Devices:**

**Server Side:**

* **Capabilities:** Mobile devices, namely iOS or Android, are not designed for server-side hosting. However, lightweight servers running Node.js are possible for testing and development purposes. The lack of processing power, memory, and network capacity would significantly impact the ability to host a web-based application at scale.
* **Costs:** Since mobile devices are not meant for server-side use, there are no direct costs related to licensing a mobile OS for hosting purposes. However, using mobile devices would be inefficient and prohibitively costly in terms of hardware, as they are not built for continuous operation.
* **Additional Components:** External hosting services would be required, considering mobile devices lack the required functionality. However, mobile devices can be utilized for testing responsiveness or mobile-specific services but may require integration with the server infrastructure to handle client-server communications.

**Client Side:**

* **Costs:** For iOS development, Apple Developer Program licensing fee is required. For Android, the tools are free, but to publish on Google Play requires a one-time fee. Additionally, devices for testing (iPhones, iPads, Android phones, and tablets) will add to expenses.
* **Language and Tools:** iOS development requires Swift and Objective-C within Xcode. Android development would require Kotlin/Java. Android Studio is the recommended IDE for Android development. For cross-platform development, tools like React Native or Flutter can build apps that run on both iOS and Android from a single codebase.
* **Skills Required:** Knowledge of Swift or Objective-C and Xcode will be required for iOS native applications. Kotlin and Java as well as Android Studio for Android applications. Cross-platform skills will include tools such as React or Flutter which utilize JavaScript for development.

**Development Tools:**

Languages:

* CSS
* HTML
* JavaScript
* MySQL or NoSQL

Tools:

* Android Studio
* Xcode

## 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**: Linux, specifically the Ubuntu Server distribution, is my recommended operating platform for The Gaming Room’s application Draw It or Lose It. Ubuntu’s stability and scalability make it the most suitable choice. Its compatibility with numerous server technologies allows for expansion to multiple computing environments. The open-source nature and compatibility with software like Docker will allow the application to be deployed across cloud environments or virtual machines, which provides the flexibility needed for future scaling and portability. These characteristics make it the strongest choice to facilitate both budgetary and functional requirements for the web-based application. Additionally, the user-friendly interface associated with this distribution can reduce the need for highly technical skills associated with operating a Linux-based environment, simplifying management and maintenance tasks for the team.
2. **Operating Systems Architectures**: The monolithic kernel architecture of Ubuntu Server means that all fundamental services like memory management, process scheduling, and drivers are handled in a single space. This design promotes efficient communication between processes, minimizing the overhead needed to handle game operations. For example, in a multiplayer game scenario, minimizing the overhead ensures that player interactions and game state updates occur in near real-time. Ubuntu also possesses a 64-bit architecture, making it highly compatible with most modern hardware, which is essential to achieve high performance and scalability. The 64-bit architecture allows the system to address more memory, thereby accommodating larger workloads and more simultaneous users as the game gains popularity.
3. **Storage Management**: Ubuntu Server can use Logical Volume Manager (LVM) or ZFS to provide a suitable storage management solution. LVM allows for easy resizing of storage volumes, which makes managing data as the player base grows simpler. For instance, as more players join, additional storage can be allocated without disrupting the system, ensuring a seamless experience. ZFS contains highly advanced features like data integrity verification; it employs checksums to verify data and metadata. If checksums do not match, ZFS will automatically attempt to fix the discrepancy, thus improving data reliability. ZFS also supports efficient data compression, making managing large amounts of data more seamless. For instance, compressing data related to player statistics can save significant storage space, which is particularly beneficial as the player base expands. Both of these systems provide flexibility and reliability, which are crucial for managing game data, such as profiles and game states.
4. **Memory Management**: Ubuntu Server includes advanced memory management features, such as paging and swapping, to optimize the use of physical memory. Paging ensures that the most frequently used parts of the program are kept in physical memory, while swapping moves less frequently used parts to disk, freeing up RAM for critical processes. These techniques help deliver smooth gameplay by efficiently allocating memory to processes like game logic, player interactions, and by reducing latency. Additionally, Ubuntu has built-in caching mechanisms that reduce disk access during peak server activity, thereby enhancing responsiveness. However, services like Redis or Memcached, as mentioned in the evaluation section, are compatible and may be preferable for optimizing performance in a distributed system. For example, Redis can cache player session data, which significantly reduces the need to query the database repeatedly, improving response times during high-traffic scenarios. These memory management techniques ensure the scalability and performance needed to support multiplayer interactions.
5. **Distributed Systems and Networks**: Seamless communication between platforms can be achieved by leveraging distributed software and network protocols. Ubuntu Server can use Redis (mentioned above) to distribute frequently accessed cached data across servers, reducing latency and improving performance. Additionally, a message broker like RabbitMQ can be leveraged to facilitate communication between services and support real-time data synchronization between clients, ensuring game states are consistent across all devices. WebSockets can be employed to maintain persistent connections between the server and client devices, which supports real-time interactions crucial to a multiplayer environment, whether users are on a desktop, mobile device, or gaming console. This is particularly beneficial for features like live drawing updates and chat functionalities. In conjunction with the software mentioned, Nginx can be used to distribute incoming traffic across multiple servers, ensuring no individual server becomes overwhelmed with requests. This reduces the occurrence of outages and increased latency, providing a better user experience. Combining these features and tools makes for a scalable and resilient distributed system that meets The Gaming Room’s requirements for Draw It or Lose It.
6. **Security**: Security is paramount for protecting user information on and between platforms. Ubuntu Server offers several security features, including a built-in firewall (UFW) to control incoming and outgoing traffic and AppArmor to enforce strict access control policies, which helps prevent unauthorized access to system resources. For network security, data can be encrypted using TLS to ensure secure transmission between clients and the server, safeguarding user credentials and game data from potential eavesdropping. Additionally, strong password policies, regular system updates, and secure API endpoints further enhance security. For example, enabling multi-factor authentication (MFA) for administrative access and securing API endpoints with OAuth can add layers of protection, ensuring that only authorized users can access sensitive information. These measures collectively make Ubuntu Server a secure platform for hosting the game, mitigating risks and protecting user data.