

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.0

## Table of Contents

[**CS 230 Project Software Design Template**](#_l6ti7uoag22u)1

[**Table of Contents**](#_30j0zll)2

[**Document Revision History**](#_grjogdjh5fi8)2

[**Executive Summary**](#_sbfa50wo7nsh)3

[**Design Constraints**](#_2et92p0)3

[**System Architecture View**](#_ilbxbyevv6b6)3

[**Domain Model**](#_8h2ehzxfam4o)3

[**Evaluation**](#_2o15spng8stw)3

[**Recommendations**](#_m8aleynsvzvc)5

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | Mar 18, 2025 | Malachi Okongwu | First Draft |
| 1.1 | Mar 29, 2025 | Malachi Okongwu | Second Draft |
| 1.2 | Apr 16, 2025 | Malachi Okongwu | Third Draft with recommendation |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room has tasked CTS with designing a web-based version of their Android game, Draw It or Lose It. The game must support multiple teams, each with multiple players, while ensuring that each game instance, team, and player is unique.

To achieve this, a singleton pattern is used for object creation, preventing multiple instances of the game. Additionally, an iterator pattern is implemented to manage teams and players efficiently, avoiding conflicts and duplication.

## [Design Constraints](#_2et92p0)

The Gaming Room currently has an Android-based deployment of Draw It or Lose It and has tasked CTS with extending it to the web. To ensure compatibility with web deployment, Java has been selected as the primary technology. Since Java is the native language for Android development, this choice should streamline the transition.

Additionally, any existing APIs supporting the Android platform will need to be reviewed or extended to accommodate web usage.

## [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 UML for the proposed design is shown below.

**"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.**

The application features a main driver class responsible for initiating the creation of games, teams, and players. The actual creation process is managed by the GameService class, which follows the singleton design pattern to ensure only one instance of GameService exists in memory at any given time.

To enforce this singleton behavior, GameService has a private constructor, preventing direct instantiation. Instead, it can only be accessed through the getInstance() method, which checks whether an instance already exists before creating a new one.

Once GameService is running, the driver class can invoke the addGame() method. This method uses the iterator pattern to ensure that duplicate game names are not created. If a new game is valid, it is added to the games list.

Similarly, after a game is created, a team can be added using the addTeam() method, which also employs the iterator pattern to prevent duplicate team names. The new team is then added to the teams list.

Players can be added to a team using the addPlayer() method, which follows the same iterator-based approach to prevent duplicate player names. Valid players are then stored in the players list.

The Game, Team, and Player classes all inherit from the Entity class, which contains two protected attributes: id and name. The default constructor is protected, preventing null object creation, ensuring that only overloaded constructors are used.

The UML design incorporates several object-oriented programming principles. Polymorphism and inheritance are demonstrated through the extension of the Entity class and the overloading of constructors. Encapsulation and abstraction are evident in the addTeam() method—users cannot directly instantiate a Team object due to constructor restrictions, yet they can create teams seamlessly through the provided method without needing to understand the underlying implementation.

## [Evaluation](#_2o15spng8stw)

There are several potential development targets, including macOS, Windows, Linux, and mobile platforms, for both hosting/server roles and client applications. The table below outlines the strengths and weaknesses of each platform.

It's important to note that the choice of server and client platforms are independent of each other. For example, even if Linux is selected as the server operating system, Windows could still be the preferred choice for the client.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | macOS Server is an option for Mac, but unless the client plans to invest in their own hardware, finding hosting services can be challenging and costly.  Pricing for macOS Server is $499 USD for up to 10 clients or $999 for unlimited clients. | Linux is the most widely used operating system for web hosting. Its open-source nature makes maintenance and licensing costs more affordable compared to proprietary operating systems like Windows.  Additionally, major cloud providers such as Google and Amazon often prioritize Linux over Windows for hosting services. | Windows servers offer a user-friendly GUI and seamless compatibility with many office applications, making them a familiar choice for businesses.  However, licensing costs can be significantly higher than Linux, as they are typically charged per user. Windows Server licenses range from $6,200 (for up to 16 core licenses) to $500 (for up to 50 clients) per installation per year.  Additionally, hosting options for Windows servers may be more limited compared to Linux-based solutions. | Mobile devices can function as personal web or file servers, but they lack the capability for multi-user hosting.  Their hardware is generally more constrained, such as limited RAM, and they do not offer the scalability of blade servers.  The cost is uncertain, as hosting tools would likely need to be custom-developed in-house. |
| **Client Side** | Developing for Macs requires a Mac computer running the latest version of Xcode.  Additionally, macOS development relies on Objective-C or Swift, which are less commonly used programming languages.  Furthermore, with Windows holding 75% of the market share compared to macOS at 16%, the potential user base is significantly smaller. | Development on Linux is relatively straightforward, as commonly used languages like Java, C/C++, and Python are all viable options.  Additionally, the GNU/Linux platform supports multi-user environments. However, the value of Linux development may be limited due to its lack of widespread adoption. | Windows development typically utilizes C# or .NET, both of which are widely used, making entry into Windows client application development seamless.  Since Windows XP, the operating system has supported multi-user functionality. Additionally, with 75% of computer users preferring Windows, it presents a strong business case for development. | Mobile devices are primarily designed for single-user experiences rather than multi-user functionality. However, developing a client application for Android or iOS is relatively straightforward.  The Android SDK is Java-based, allowing for potential code reuse from Windows or Linux development as a starting point. In contrast, iOS development requires Swift, similar to Mac applications, which also comes with specific hardware requirements. |
| **Development Tools** | Mac development primarily utilizes Objective-C and Swift as programming languages.  The standard integrated development environment (IDE) for Mac applications is XCode.  XCode requires a subscription fee of $99 USD per year per developer. | Linux development can involve languages like C/C++, Java, or Python.  For Python development, free IDEs like Notepad++ are available, with PyCharm being another popular choice.  There are many C/C++ IDEs, although not all are compatible with Linux. Eclipse is a versatile and free option that supports all these languages. | Windows development is mainly done using C# and .NET.  Microsoft's Visual Studio is a highly popular IDE, offering a wide range of plugins and integration options such as Jenkins, TestComplete, and more.  The cost of Visual Studio ranges from $45 to $250 USD per user per year, depending on the features included. | The Android SDK is Java-based, and the most popular IDE for Android development is Android Studio, which is officially developed by Google. Android Studio is free to download.  For iOS, Objective-C and SWIFT are primarily developed using XCode.  XCode costs $99 USD per year per developer. |

## Recommendations

**1. Operating Platform**  
For backend infrastructure, Linux-based servers are the recommended option regardless of the chosen frontend technology. Linux offers reduced licensing costs and broader hosting options, unlike Windows, which may impose limitations based on its licensing model. Since the frontend will interact with the backend via APIs, it can remain platform-independent.

Linux is a widely adopted server OS known for its security, stability, and extensive tool support. Its popularity ensures compatibility with a wide range of security and operational tools.

The frontend can be developed using the most suitable language for each target platform—for example, Swift for iOS, Java for Android, or .NET for Windows—ensuring a native and optimized user experience.

**2. Operating System Architecture**  
A recommended approach is to use a backend server to manage the core game logic while rendering is handled by the client application. Given that *Draw It or Lose It* is not reliant on fast reaction times, communication between the backend and frontend can be asynchronous, minimizing concerns over latency.

Implementing a modern, scalable backend using containerized microservices orchestrated with Kubernetes or Docker would allow for efficient scaling and maintenance. The specific architecture would ultimately depend on the selected cloud provider, as each offers unique toolchains and integrations.

Delegating rendering to the client reduces server load and operational costs, while also improving performance by insulating the game experience from network fluctuations. The client could cache upcoming images to ensure smooth visual rendering throughout gameplay.

It should also be determined whether the application will be browser-based or deployed as a standalone desktop application. Browser-based deployment, particularly through a Progressive Web App (PWA), could simplify distribution and cross-platform support.

**3. Storage Management**  
Unless The Gaming Room chooses to invest in dedicated hardware, the type of storage medium (HDD vs SSD) becomes less critical. Both storage types would be adequate, especially when combined with intelligent caching and client-side rendering.

Leveraging cloud-native storage solutions will provide flexibility, particularly in supporting scalability and regional deployment.

**4. Memory Management**  
On Linux servers, memory is managed using the page cache and virtual memory with a demand paging strategy. This ensures that only necessary memory pages are loaded, conserving RAM usage. Page replacement typically follows the Least Recently Used (LRU) method.

Android uses the Android Runtime (ART) and the Dalvik VM, both of which employ paging and memory mapping. This means once memory is modified or accessed, it remains in RAM and is not paged out.

In iOS, memory management is handled by Automatic Reference Counting (ARC), which reduces developer overhead by managing memory allocations at compile time via Xcode.

Client-side memory demands will be minimal. Only one or two images need to be stored in RAM during gameplay, along with the memory required to operate the rendering engine, such as a web browser. On the server, minimal RAM will also be sufficient due to the offloading of rendering tasks to the client. However, if microservices are used, memory requirements may grow in proportion to the number of active users.

**5. Distributed Systems and Networks**  
To maximize uptime and minimize the risk of service disruption, deploying the game in a cloud-native environment is recommended. Most cloud providers support failover capabilities and service replication across geographic regions, improving resilience.

Communication between the frontend and backend will occur over RESTful APIs in an asynchronous fashion. This decoupled design allows the frontend to remain agnostic of the server environment, enabling support for Android, iOS, Windows, and more.

**6. Security**  
The application will use role-based access control (RBAC), requiring the development of an entitlements management interface to support secure and efficient role administration.

The principle of least privilege will guide user permissions, limiting each user’s access to specific features such as creating games, naming teams, or enrolling players. A team-based hierarchy could optionally be implemented, enabling designated team leaders to manage their respective teams.

Administrative-level access will be explicitly disallowed for regular users to reduce security risks.

API endpoints will be secured with SHA-256 encryption using 128-bit keys, and all connections will require TLS 1.2 or higher. Digital certificates will be sourced from Entrust to ensure authenticity and trust.

In addition, a firewall will be configured on the server to enforce industry-standard security practices and provide an added layer of protection.