

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 3.0

## 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 |
| --- | --- | --- | --- |
| 3.0 | 06/17/2025 | Brendan Arseneau | Completed full software design document part 3 |

**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)

This phase of the project focuses on evaluating and recommending the best platforms and tools to expand “Draw It or Lose It” beyond Android. The client wants to support desktop (Windows/macOS), mobile (iOS), and potentially web platforms. This means choosing a reliable server operating system, planning for multi-platform client support, and selecting the right development tools. The main goal is to make sure the application remains secure, scalable, and easy to maintain while supporting future growth.

## Requirements

The Gaming Room wants to scale the game to more platforms without starting over for each version. The codebase needs to work across devices like phones, desktops, and possibly browsers. Multiplayer features, secure client-server communication, and performance are top priorities. The system should be lightweight, flexible, and support updates without disrupting gameplay. Code reuse is especially important to avoid high maintenance and development costs.

## [Design Constraints](#_2et92p0)

Developing for multiple platforms introduces several limitations. Each operating system has different requirements, UI expectations, and tools. For example, Android uses Java/Kotlin, iOS uses Swift, and desktops have their own compatibility layers. To avoid building multiple separate codebases, the design should use shared logic and platform-neutral tools where possible. Also, since the original app was written in Java, the system needs to remain compatible with that base unless a clear cross-platform alternative is chosen.

## [System Architecture View](#_ilbxbyevv6b6)

A client-server architecture is ideal for this type of multiplayer game. The server should handle all business logic, session data, and multiplayer synchronization. Clients whether mobile, desktop, or web would simply connect to the server, send/receive updates, and display the UI. Hosting the server on a Linux environment is a smart move, since it’s cost-effective, reliable, and widely used for online applications. This separation also makes it easier to scale, secure, and update each part of the system independently.

## [Domain Model](#_8h2ehzxfam4o)

The same object-oriented model from the earlier version of the game still works well for a cross-platform environment. The base Entity class supports shared properties like ID and name, and the Game, Team, and Player classes extend it cleanly. These classes are managed by the GameService layer, which controls the logic behind creating and connecting all game elements. For cross-platform use, storing and transferring data in formats like JSON ensures everything remains consistent between platforms. The model is already lightweight and avoids duplication, which helps with performance and future changes.

**"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)

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** | macOS can technically run web servers like Apache or NGINX, but it's rarely used for production hosting. Licensing costs and limited support make it less suitable for scaling a multiplayer app. Better suited for local development or internal testing environments. | Linux is the most recommended platform for hosting. It’s lightweight, secure, open-source, and supports nearly all major server technologies (Apache, NGINX, MySQL). It also has strong community and enterprise support, with lower operating costs | Windows can host with IIS or third-party tools, and may integrate well with Azure if using Microsoft-based services. However, it tends to consume more system resources, requires licensing, and is less common for hosting online games. | Mobile devices aren’t designed to run game servers. They lack the necessary network stack, power, and reliability. Hosting a server on mobile is not supported or secure for this type of application. |
| **Client Side** | |  | | --- | |  |   Mac is a solid client platform. It supports Java and most dev tools, but some Java-based apps require signing to run. UI responsiveness is good, and performance is stable, but distribution outside the App Store is more limited. | Linux works for testing or advanced users, but isn't a common platform for players. Some UI libraries may require extra configuration. However, for developers, it works well and offers full control. | Windows is the most compatible platform for client use. Java runs out of the box, and the platform supports a wide range of tools, graphics libraries, and device configurations. It's ideal for desktop play and dev testing. | Mobile is the core focus. Android and iOS support will require native UI optimizations, responsive layouts, and touch input handling. Mobile clients must prioritize performance, battery usage, and data sync. |
| **Development Tools** | Eclipse and IntelliJ work on Mac, but sometimes Java SDKs or tools need extra configuration. It’s doable, just takes more effort. | Linux is great for developers who are used to it. Tools like VS Code and Eclipse run well, and Linux gives more control overall. | Windows is the easiest setup for development. Java, Eclipse, and everything else works right away with minimal setup. | For mobile, Android Studio is the main tool. It’s more complex than desktop tools, but it’s necessary if you want to build and test for phones. |

Based on the comparison above, Linux is clearly the best choice for server-side deployment due to its flexibility, stability, and cost-effectiveness. Windows and macOS can work for development and testing, but aren't ideal for production hosting. On the client side, Windows offers the smoothest experience for desktop users, while mobile platforms demand special attention to UI and performance. Each platform comes with its own development trade-offs, so choosing tools that support code reuse (like cross-platform frameworks) will be key.

## 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**

For the server side of Draw It or Lose It, Linux is the best option. It’s reliable, secure, and used all the time for hosting web-based apps like this. It runs well without using a ton of resources and works with a lot of open-source tools that can keep things fast and affordable. I also like that it’s already a common setup for multiplayer games, so it makes sense to stick with something proven.

**2. Operating System Architectures**

The app should follow a basic client-server structure. The main server handles all the logic, game state, and image files. Every client, whether it’s on Android, Windows, Mac, or a browser, just connects to that server. This setup makes it easier to add new platforms later without needing to rebuild the whole thing. It also helps keep everything organized and easier to update.

**3. Storage Management**  
To store everything, I’d use a file system to handle all the images and a small database like MySQL or SQLite for the game data. The images could be sorted into folders and maybe even delivered through a CDN to make sure they load fast when a bunch of people are playing at once. That keeps the app smooth and helps avoid delays. The database would handle stuff like scores, game sessions, or user info.

**4. Memory Management**  
Memory needs to be managed smart so the app doesn’t crash or slow down, especially with high-quality images. The game should only load what it needs for the current round. After that, it should clear it out so memory doesn’t get clogged up. Using smaller image formats and letting Java’s garbage collector clean things up will also help. The goal is to keep it running smoothly without using more memory than it has to.

**5. Distributed Systems and Networks**  
To make sure the game can grow and handle players from different places, it should use a system that connects all the clients to one central server. That server can manage everything, and if the app gets big, multiple servers can be set up behind a load balancer to split the work. That way if one goes down, the others keep it running. Using services like AWS or Azure could make this easier to set up and manage.

**6. Security**  
The app needs to keep players and their data safe. All traffic should be encrypted so nobody can see what’s being sent. Players should also have to log in or verify who they are before joining. On top of that, any input from players should be checked so nobody can try to cheat or break the app. Nothing sensitive should ever be saved as plain text, and any outside libraries being used should always be kept up to date to avoid security issues.