

Draw it or lose it

# **CS 230 Project Software Design**

Version 1.2

## Table of Contents

[**CS 230 Project Software Design**  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 | 01/25/2025 | Jacob Griggs | Design team game environment of Draw it or lose it |
| 1.1 | 02/05/2025 | Jacob Griggs | Second Draft |
| 1.2 | 02/19/2025 | Jacob Griggs | Third Draft(Recommendations) |

## [Executive Summary](#_sbfa50wo7nsh)

We have been asked to develop a web version of the Android app. Our help is needed to streamline the games development to ensure it meets all software requirements.

## [Design Constraints](#_2et92p0)

The Gaming Room has an existing Android-based deployment of the Draw It or Lose It. We have been asked to extend this to the web. The code of the Android version needs to be updated or rewritten to adapt it to a web version

**System Architecture View**

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 below. GamService is the parent, and Game, Team, and Player inherit Entity. The application consists of a main driver class that will be used to initiate the creation of the games, teams, and players. The actual creation is through the GameService class and follows a singleton design pattern so that only a single GameService class may exist at any time in memory.

**"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** | **Characteristics**: Less common for server use, more often used for development.  **Advantages**: Strong performance and integration with Apple ecosystem.  **Weaknesses**: Higher hardware and maintenance costs, limited server software options.  **Server-Based Deployment**: Limited, typically not used for large-scale web hosting.  **Licensing** **Costs**: Higher due to hardware and macOS Server costs. | **Characteristics**: Open-source, highly customizable, and widely used for web servers.  **Advantages**: Cost-effective (no licensing fees), robust security, and extensive community support.  **Weaknesses**: Requires technical expertise to manage and configure.  **Server-Based** **Deployment**: Yes, supports various web servers like Apache, Nginx.  **Licensing** **Costs**: Generally free, but enterprise support may incur costs. | **Characteristics**: User-friendly with extensive enterprise support.  **Advantages**: Seamless integration with other Microsoft products, strong support for .NET applications.  **Weaknesses**: Higher licensing costs, potentially more vulnerable to security threats.  **Server**-**Based** **Deployment**: Yes, supports IIS (Internet Information Services).  **Licensing** **Costs**: Windows Server Standard Edition costs around $1,176, and Datacenter Edition costs around $6,771 | Mobile devices can be used as a personal webserver or file server, but they are not equipped for multi-user serving. The hardware is typically more limited, e.g., RAM, and they are not scalable like blade servers. Costs is unknown as the hosting tools would probably need to be designed and built in-house. |
| **Client Side** | To develop for Macs, you need a Mac computer running the latest version of XCode. Moreover, the macOS SDK is in Objective-C or SWIFT which are lesser-known languages. Lastly, Windows usage is 75% of the market vs. macOS’s 16%. This presents a smaller market opportunity. | Development in Linux should be straightforward as Java or C/C++ or Python could be the language of choice – which are all commonly used. | Windows is typically developed using C# or .NET which are both common. There would be no barrier to entry to development of a Windows client application. | Mobile devices are not designed to be multi-user. However, design a client application for Android or iOS is straightforward. Android SDK is Java based so code developed for Windows and Linux might be able to act as a jumping off point. iOS is SWIFT based so the same requirements for Mac apply, including the hardware needs. |
| **Development Tools** | Mac use Objective-C and SWIFT for development languages. XCode is the common IDE used for Mac development. XCode is listed as $99 USD per year per developer | Linux development may take the form of C/C++, Java, or Python. Python IDEs are often free. PyCharm is popular Python IDE. C/C++ IDEs are numerous – but not all are available for Linux. Eclipse can do all of these and is free | Windows is primarily developed using C# and primarily .NET. Microsoft’s Visual Studio is an immensely popular IDE. Visual Studio ranges from $45 – $250 USD per user, depending on features, per year | Android SDK is Java based and the most widely used Android IDE is Android Studio which is developed by Google as the official development tool. Android Studio is free to download. iOS’s Objective-C and SWIFT languages are almost exclusively developed in XCode. XCode is listed as $99 USD per year per developer |

## Recommendations

1. **Operating Platform**: The most cost effective and versatile OS would be Linux. It reduces cost and does not limit access to the servers. Linux offers good security and operability. There are many tools available including security software. Since Linux can have an agnostic front end it can be transfer to different platforms.
2. **Operating Systems Architectures**: The suggested architecture would be for a backend server that controls the game environment and client-based rendering. All interactions could be done asynchronously. Since Draw It or Lose It gameplay is a client-server relationship there should be low latency between the frontend and backend. A more modern backend running containerized microservices like Kubernetes or Docker would allow for scalability. Choosing to use the frontend for rendering allows the server to offload some of the more resource intense parts of the application which would reduce the monthly data center costs. Moreover, client-side rendering should also insulate gameplay from network issues as framerate is important to gameplay. The client could cache some number of subsequent images ahead of active gameplay ensuring a smooth rendering.
3. **Storage Management**: There is no need to worry about storage management as any storage should be adequate for the games performance. HDDs or SSDs should provide the performance needed. On the server-side using cloud-native tools would help remove any other storage burdens
4. **Memory Management**: Linux uses the concept of page cache for data stored in main memory virtual memory for any pages allocated. Linux uses demand paging which allows for lower memory usage because ages not actively being used will not be loaded into memory. Page replacement is based on the Least Recently Used(LRU) algorithm.
5. **Distributed Systems and Networks**: Uptime considerations and outage prevention are the reasons so many applications are being built in cloud native architectures. Many cloud providers can replicate and shift services amongst different deployments to prevent large outages. The frontend and backend will communicate through RESTful APIs asynchronously. RESTful API usage allows the client/server communication to be transparent to the deployed frontend.
6. **Security**: Having a firewall implemented would help to increase security of the platform. The main security of the application should consist of Role-based authorization. This means that an entitlements interface will need to be created so effective administration of the roles and accounts is possible. The idea of least-privilege should be employed which should limit users in their scope to game controls, i.e., game creation, team name creation, team enrollment. If there is a need, the user scope could be extended into a team-captain/member hierarchy to allow limited users to edit a team or add/remove players from a team. A firewall should also be added as part of the server using industry-standard best practices for the default settings