

Draw it or Lose it

# **CS 230 Project Software Design Template**

Version 1.0

Caelum Tobon

## 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 | 1/26/25 | Caelum Tobon | Initial software design prototype |
| 1.1 | 2/9/25 | Caelum Tobon | Updated software design evaluation |
| 1.2 | 2/23/25 | Caelum Tobon | Updated software design recommendations |

**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 wants to develop a web-based game version of their app, to serve multiple platforms, which is currently only available on Android in an app called Draw It or Lose It. The game consists of various games that host numerous teams with multiple players. In order for the game to function properly, each game and team name needs to be unique. Draw It or Lose It contains a large library of stock drawings and the staff at The Gaming Room do not know how to set up the proper environment.

## Requirements

Create a web-based version of an already established Android app to host numerous teams and players with unique IDs.

## [Design Constraints](#_2et92p0)

Android and the Web have different software development kits. The API needs to be able to allow 1 or more teams from either of the platforms. Game and team names must be unique. Must have the ability to alert the team that a team name already exists and let them choose again. Use unique IDs for each instance of a game, team, and player to limit instances to one.

## [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 ProgramDriver Class holds the main method. ProgramDriver uses Directed Association with SingletonTester to test whether there is already an instance of GameService or not. Entity class is the parent class to Game, Team, and Player classes. Game, Team, and Player all inherit Entity’s required attributes. A Player can’t have a Team, but a Team can have a Player. A Team can’t have a Game, but a Game can have a Team. A Game can’t have a GameService but a GameService can have a game. Game Service must only have one instance of each game running. Each Game can only have one unique Team at any time. Lastly, Teams can only have one of each individual Player at one time.

**"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** | Macs can be used with OS X servers, but it can be expensive if they either want to find a host or purchase their own hardware. | Linux is a practical choice for web-based hosting. It is open-source so maintenance and licensing will be cheaper than a closed OS like Windows. Cloud providers like Google and Amazon have a preference to offer Linux over windows. | Windows is a very secure choice, as they are GUI-based, and many applications used in the workplace will also be running on the server, creating more familiarity with it and therefore making it easier to work with. However, licensing costs can be high compared to Linux and hosting platforms can also be more limited. | It can be used as servers but isn’t the best at doing so as they aren’t capable of hosting multiple users. Hardware is also typically more restrictive in terms of things like RAM and are not scalable. Hosting tools would also likely need to be designed and created in-house so costs are not solidified and could potentially be over the desired budget. |
| **Client Side** | In order to develop, a Mac with Xcode is required. A good knowledge of SWIFT is also required as it is what will be used to develop, it is however a lesser-known language. Using Mac also limits the potential market when compared to Windows. | The development should be simple as Java, C++, and Python could be used, all of which are common languages. It can also support multiple users. However, its use isn’t very widespread, which could decrease the project’s overall value. | It is usually developed using C# or .NET which are both fairly common. Windows is also very capable of hosting multiple users and is the preferred OS for many computer users which could increase the project’s value. | Again, they aren’t made to host multiple users, but the design would be rather straightforward for a client application. Android is Java-based while IOS is swift-based so it may require more time and resources than other options, possibly leading to a decreased profit. |
| **Development Tools** | Mac uses SWIFT to develop, for this the Xcode IDE can be used. It will cost 99$ a year for each developer. | Linux can be developed with C++, Java, or Python. PyCharm is a popular free IDE for Python. C++ IDEs aren’t all available for Linux, however, the Eclipse IDE is capable of using all of them for free as well. | It is primarily developed using C# and .NET, these can be done with Microsoft’s Visual Studio which offers many plugin and integration options, this is a little pricier though, with a subscription fee of 45$-250$ yearly per developer. | Android is Java-based, the IDE most suitable to develop would be Android Studio which is free. iOS will however use the same IDE as Mac and therefore also cost the same. |

## Recommendations

Analyze the characteristics of and techniques specific to various systems architectures and make a recommendation to The Gaming Room. Specifically, address the following:

**Operating Platform**:

After analyzing the relevant information, using Linux-based servers would be the best option in this scenario. By using these servers, we can minimize the license costs while not limiting access to data centers in the same way that Windows servers can. The front-end may be agnostic to the backend and connect via APIs. Linux also boasts decent security and operability. Since it is one of the most common server platforms, there are many tools that are readily available including security software. Since the front-end will be agnostic it can be written in the preferred language for that platform, e.g., SWIFT for iOS, Java for Android, and .NET for Windows.

**Operating Systems Architectures**:

The recommended architecture would be for a back-end server that manages the game’s environment and front-end/client-based rendering. Also, there does not need to be low latency between the front-end and back-end. So, any transmission could be done asynchronously. A more modern back-end running containerized microservices with Kubernetes or Docker could allow for scalability. The cloud provider would need to be selected before an exact architecture was determined as most cloud providers have some kind of proprietary tooling. When choosing to use the front-end for rendering, it allows the server to offload some of the more resource-intense pieces of the application which could help reduce the monthly data center costs. Moreover, client-side rendering should also insulate gameplay from network issues because framerate is particularly important to gameplay. The client could cache a few select subsequent images ahead of active gameplay ensuring a smooth rendering process. However, it still needs to be decided if the application will be browser-based or some sort of Java app when run on PCs or Macs. As a recommendation, I believe it might be easier to choose to create a browser-based game and enable it with PWA.

**Storage Management**:

Unless the Game Room wants to purchase its own hardware for this project, no decision on a storage medium, HDD vs SSD, needs to be made. Either HDDs or SSDs should be able to provide what the application needs to perform the best, especially with some sort of caching behavior and client-side rendering. On the server side, using cloud-native tools will add flexibility, especially where scalability or localization is concerned.

**Memory Management**:

Linux uses the concept of page cache for data stored in main memory and virtual memory for any pages allocated. Linux also uses demand paging which allows for decreased memory usage because pages that are not actively being used will not be loaded into memory. Page replacement is based on the Least Recently Used (LRU) algorithm. The Android Runtime (ART) and Dalvik virtual machine both use paging and memory mapping to help manage memory. This means that any memory an app modifies, whether by allocating new objects or touching mapped pages, remains resident in RAM and cannot be paged out. Memory management in iOS was initially non-ARC (Automatic Reference Counting), where we need to retain and release the objects. Now, it supports ARC, and we do not have to keep and release the objects. Xcode can take care of this task automatically in compile time. Minimum RAM amounts on the server will be needed with client-side rendering; however, if we are using modern architecture with containers and microservices the cost will be scaled with the number of users. Client-side RAM should also be minimal, as only 1-2 images need to be stored in memory at any given time and then any RAM needed to drive the client application, i.e., the browser needs.

**Distributed Systems and Networks**:

Uptime considerations and outage prevention are the reasons that many applications are being built in cloud-native architectures. A lot of cloud providers can replicate and shift services amongst different deployments to prevent large-scale outages. The front-end and back-end will communicate through RESTful APIs asynchronously. Using RESTful API allows the client/server communication to be transparent to the deployed front-end, e.g., Android, Windows, and iOS.

**Security**:

Security will consist of Role-based authorization. This means that an entitlements interface must be made so effective administration of roles and accounts is possible. The idea of least privilege should be used, which should limit users in their scope to game controls, i.e., game creation, team name creation, and team enrollment. If needed, the user scope could become extended into a team-captain/member hierarchy to give permissions to limited users to edit a team or add/remove players from a team, however, no user will have ADMIN privileges on the system. APIs will be protected using encryption, SHA 256, with 128-bit keys, and TLS below 1.2 will not be allowed. Certificates can be purchased from Entrust. For additional protection, a firewall should also be added as part of the server using industry-standard best practices for the default settings.