

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.0

## 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 |
| --- | --- | --- | --- |
| 1.0 | 05/21/25 | Kyle Shaw | Worked on summary, requirements, design constraints sections. |
| 1.1 | 06/09/25 | Kyle Shaw | Added evaluation consideration section. |
| 1.2 | 06/29/25 | Kyle Shaw | Added operating platform recommendations to help make the app cross platform. |

**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 client for the Gaming Room company wants us to bring their mobile app to a web-based environment. The *game Draw It or Lose It* has game instances where each game can have one or more teams involved, and each team can have one or more players. Each round has a timer of 30 seconds before another team can answer and take the points within a 15 second time limit. We will be coding the backend in java to handle user requests from the web.

## Requirements

*Business Requirements:*

* *A game that will have the ability to have one or more teams involved*
* *Each team will have multiple players assigned to it*
* *Game and team names must be unique to allow users to check whether a name is in use when choosing a team name*
* *Only one instance of the game can exist in memory at a given time. This can be accomplished by creating unique identifies for each instance of a game, team, or player*

*Technical Requirements:*

- Ensure a timer runs for 30 seconds before other teams can guess

## [Design Constraints](#_2et92p0)

Users must be able to get game data from the server including the stock image library

When a user creates a team or username, they must validate it with the server before playing.

Should consider internet connection speed when loading images and what happens if a player loses connection.

Use Java as the server language for the web application.

## [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 client requires each game, team, and player to have a name and identifier. The UML diagram below solves this by having each Game, Team, and Player object inherit from the Entity class which has common attributes and methods. The Game, Team, and Player classes all override the toString() object they inherit from Entity to give their own descriptions.**

**The Game class has a method to create a new team and uses the iterator pattern to check if that team name exists already. If no team name exists, GameService’s GetNextTeamID function is called returning a static value that holds the next unique team ID value and incrementing that value by 1. This returned value and the team's name is then used to create a new Team instance and store it in a list of teams.**

**The Team class has a method to create a new player and uses the iterator pattern to check if that player’s name exists already. If no team name exists, GameService’s GetNextPlayerID function is called returning a static value that holds the next unique player ID value and incrementing that value by 1. This returned value and the player’s name are then used to create a new Team instance and store it in a list of players.**

**Another requirement is that there can only be one instance of the game that can exist at anytime. The GameService class has a method to check for unique names when generating a new game in the addGame function.**

**The GameService class implements the singleton pattern to ensure only one instance of the GameService can exist at any point in time. If someone tries to create more than one GameSerivce instance, it will instead return the current instance.**

"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** | Mac OS can run servers and even comes with Apache installed. Any servers run would have to be on main OS as Mac OS Server was discontinued. You can still use the software on Mac OS to run a server. Mac OS can only be used on Apple supported hardware. | Linux is open source and requires no licensing fees. Being open source means it has many distributions like Debian and Ubuntu Server. Many distros are built to be lightweight, long-term version support, and stable updates. | Windows can be reliable to host server software on but more expensive than Linux. If you wanted to spin up more servers of windows, each instance would need a license. Window itself has a lot of features that might slow it down for a server, so hosting it on Windows Server or Linux could provide more efficiency. | Running a server on a mobile device is not recommended. Mobile OS are built for user interaction and not a server environment. Limited ram and memory that cannot be easily expanded. |
| **Client Side** | You can develop a mac OS app using swift or objective c (swift is usually used in new applications). Mac OS only has 16% market share, so a mac app won't reach as many people as a windows app would. Mac is usually run on hardware apple produces so issues on different types of hardware is less likely. Cross platform languages like java could cut down development time but not have a native feel. A website that connects to the Api though a web-browser is another option. | Linux is generally used by people who are very knowledgably about computers. It is not as widely used as mac, windows, or mobile devices in general use. One main concern is how an application might work across multiple Linux distros. While a Linux app should be runnable on any distro, we should consider the distros that would be most popular with gaming. A website that connects to the Api though a web-browser is another option. | Window has a large pc market share at 71 percent.  Windows is wildly used by all types of people, including casual users and gamers.  There are many options for developing client apps on windows. Building a client app that is cross platform, or a website can cut down the need for separate apps on different platforms. | Native client apps would require separate software for IOS and Android meaning extra cost and time. There are many different mobile screen sizes so developing and testing for different sizes is needed. Client already has an android app, but we would still need to update it, so it works with the server API. |
| **Development Tools** | On mac you can use Xcode to develop swift mac applications for the client side. Other IDEs are available to use including visual studio and eclipse. We could compile java code to run our server code on the mac.  If you want to publish to the mac store, then you need to use Xcode to publish it. | Many languages work on Linux including C++, c#, java, etc.  A variety of IDEs and code editors also work on Linux including Visual Studio, Eclipse, JetBrains IDEs, etc.  Xcode is not supported on Linux. We could compile java code to run our server code on Linux. | Many languages work on windows including C++, c#, java, etc.  A variety of IDEs and code editors also work on windows including Visual Studio, Eclipse, JetBrains IDEs, etc.  Xcode is not supported on windows. We could compile java code to run our server code on windows. | For IOS, you can develop and publish apps on Xcode. IOS apps generally use swift language for native.  You can develop Android apps natively using Kotlin or Java. Android studio is a popular IDE to use for android. |

**One extra thing to consider is that as certain operating systems support ARM architecture now it might lead to some incompatibility on working on all devices that OS supports. I believe you can compile your applications for the arm architecture as an option for users to download.**

**If a website is used as the client to access our backend api we need to consider different browsers and what features they support and also different screen sizes.**

## 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**:

The operating system I would recommend for hosting Draw It or Lose It’s backend API is Linux. Linux has the advantage of being open-source meaning that many of its distributions do not have licensing fees. Windows Server on the other hand has a licensing fee, and if you need to expand past 16 cores or have more than one processor, you will need to buy additional licenses. Mac OS’s can only be licensed on apple supported devices which limit you to their hardware.

1. **Operating Systems Architectures**:

A article on *geeksforgeeks* mentions how the Linux kernel is monolithic meaning that more functionality is included in the kernel code like file systems (Monolithic Kernal and Key Differences From Microkernel, 2024). A monolithic Kernel has the advantage of being faster than a microkernel because less interprocess communication is needed. The Linux kernel is also modular, meaning that functionality can be added to it without needing a restart.

1. **Storage Management**:

The game Draw It or Lose It has 200 high-definition images it needs to store. Our backend should use an SSD instead of an HDD for quick access of data especially for commonly requested data like images. While SSD are more expensive than hard drives, they have gotten much cheaper over time. Images should be kept in a file system rather than a database as databases are not as efficient with large files. Another thing to note is that Linux generally uses EXT4 as its file system. Doing some research, I found that XFS can possibly be more performant when it comes to large files so partitioning an SSD with XFS might give a performance boost with the high-quality image sizes.

1. **Memory Management**:

Draw It or Lose It will need to constantly load images into memory in order to process them and send them to users. While storing images on an SSD can be fast, memory is quicker. This leads to the question of what should be stored on memory vs storage. At first, I thought maybe we could store the images in a static array to keep them in memory for quick use. On average each image is 8MB and we have 200 images. This is about 2GB which can easily fit on memory. A problem is that if we wanted to add more images down the road, this could dramatically increase in size.

In chapter 11 of *Operating System Concepts* Linux page cache is mentioned and it's where files requested from storage are shared in a memory cache that the operating system handles. This means that we can request images from storage and repeated requests of that image could hit the page cache instead. I was worried at first how common files would need to be reread from missed cache hits but when memory is not in demand, the page cache can be gigabytes in size. Linux prioritizes what stays in the cache by Least Recently Used (LRU), meaning that images that are commonly used should stay in cache. If the company wanted to expand and add more images in the future I think reading from the SSD and letting page cashing might be the best option as the commonly used images should stay in the cashe.

1. **Distributed Systems and Networks**:

In Draw It or Lose It the server and clients worked together in a client-server architecture as a distributed system. When a user wants to search for available games, join a game, get game updates, and take a guess, then they need to communicate with the server. The server can give the player updates via JSON files which the client program can use to update the new game information to the user. With the REST API the client is the one communicating with the server so to get game updates the client needs to make new requests.

One thing to consider is connectivity between the user and the server. With RESTful APIs connectivity is solely based on the client communicating with the server. If the client where to leave the game the client side should inform the server otherwise the server wouldn't know. Both the client and server should have a timeout function where after a set amount of time they determine that the other side has lost communication.

1. **Security**:

In Draw It or Lose It password and username authentication from the user for each request to confirm who they are. The role permission-based system checks the user's role to see what they are allowed to access. The biggest security concern is to make sure the user's basic login token is hard to steal otherwise any other user could act as that user. Basic tokens should be sent through https rather than http as https encrypts communication with the server and user.

Linux also has role-based authentication, which is important for protecting server code and data from unauthorized access. If someone in the organization wanted to change a server file if proper file permissions were set, they would need to be authenticated and authorized to do so.

References

GeeksforGeeks. (2024, July 29). *Monolithic kernel and key differences from microkernel*. GeeksforGeeks. <https://www.geeksforgeeks.org/operating-systems/monolithic-kernel-and-key-differences-from-microkernel/>

Wodehouse, C. (2025, June 4). *XFS vs. Ext4: Which Linux File System Is Better?* Pure Storage Blog. <https://blog.purestorage.com/purely-educational/xfs-vs-ext4-which-linux-file-system-is-better/>