

Draw it or Lose it

# **CS 230 Project Software Design Template**

Version 1.1

## 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 | 05/27/25 | Zach Gutman | Version 1.0 Launch |
| 1.1 | 06/04/25 | Zach Gutman | Version 1.1 – updated requirements, constraints, and evaluations. |
| 1.2 | 6/20/25 | Zach Gutman | Version 1.2 – updated evaluations and recommendations for server OS |

**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)

Our client: The Gaming Room wants to port their existing game application *“Draw It or Lose It”* from its existing android platform to multiple different platforms- including mobile devices. The client has stressed that multi-platform support is critical to the development of the app. Players will need to be assigned a unique ID and then grouped into teams. Teams will have a limited amount of time to guess each other’s drawing and then the other team may guess- if the drawing isn’t guessed correctly in the allotted time. Games and teams must be unique.

## Requirements

* Develop a multi-platform application of the game *“Draw It or Lose It”*
  + Platforms compatible must include: Linux, Mac, & Windows
* Each game must consist of at least 2 teams
  + Each team must consist of at least 4 players
* Each game, team, and user must be unique
  + Only 1 instance of each game should exist at any given time
* Each player ID must be unique to ensure that only one instance of each player is active at any given time

## [Design Constraints](#_2et92p0)

* Browser-based games require a lot of testing due to many different browsers being used
  + Additional testing will need to be done for Linux, Mac, and Windows platforms
* The game will need to be optimized for desktops, tablets, and mobile users
* The game will need to be able to communicate between multiple different browsers and devices simultaneously
* Because nearly every device can connect to the internet, we might need to account for much greater server capacity
* Players will need to be able to register a user ID and submit a team name
* Players will need to be automatically assigned to a team and game
* Games will need to auto-populate on the server and automatically be assigned a unique ID
* Games will need to be terminated after the game has been completed to save memory

## [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 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.**

Above is our UML diagram for the *Draw it or Lose it* web-based game. The entity class serves as our superclass that is referenced by our other classes such as Game, Team, Player. Common attributes are shared among our classes such as ID and name. This helps to ensure that every game, team, and user have a unique ID and name. These subclasses will help us to develop of the skeleton of the game. These being the teams, the Game itself, and the players.

Off the Game subclass we have the GameService class. This has a compositional relationship with the Game class. This means that the GameService class helps to manage the Game class. We can see a similar type of relationship with the Team subclass and the Player subclass. Each modifier the other classes.

Our ProgramDriver class is where our actual game application will reside. Users when entering the site will be greeted with the ProgramDriver, from there the singleton instance is created. This ensures that there is only one instance of each game at any one time. Lastly the ProgramDriver has a dependency relationship with the SingletonTester class.

We can see several different object-oriented programming techniques used in the UML above. We have inheritance demonstrated from the Entity class and the Game, Team, and Player subclass. We use these subclasses to assign the attributes to the main Entity class. Using this method saves us memory on the server side because the subclasses will only have to referenced once. This should improve the game’s performance on the user’s side. Next we have encapsulation, which is contained in the GameService class. Many of the attributes of GameService are private data points- denominated by the – symbol. This keeps this unnecessary data from interfering with the User’s experience. This is showcases again in the SingletonTester class, having a – symbol ensuring that a single instance is created and cannot be edited by another constructor. Finally abstraction, which is shown in our UML diagram using the inheritance of common properties into our classes. Our Game, Team, and Player classes are all abstractions of the Entity class. Another abstraction example is that the SingletonTester access via getInstance(). The user won’t know or care how the instance is created- it only calls the method for access. The UML utilizes this in several places with much of the data being handled in the background processes for the user.

## [Evaluation](#_2o15spng8stw)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Hardware for Mac can be expensive but is very friendly for developers and tends to be secure. Scalability could be an issue with hardware. Mac’s operating systems also tend to be very consistent and updated regularly with small updates that don’t leave user’s feeling lost in the OS. Another con worth considering is hardware scalability. Server’s that can run a Mac ecosystem come directly from apple and can be very costly when compared to Linux or Windows systems. Additionally, the hardware has almost no capability for customization. Windows and Linux machines can freely swap parts at will, but Apple products tend to be set in stone after production. Lastly to consider is the lifespan of a apple product could much shorter. | A lot of customization because of the open source nature of this OS. This also let’s developers develop an OS that can be custom made for the program they are designs. Linux also tends to be the most secure OS of any on the list. Outside some very niche circumstances Linux boasts so great protections primarily because of it’s popularity with super-users and it’s dislike by users who might not be familiar. That being said, Linux is the most cost effective system with a price tag of generally $0. Linux also tends to be a lite and runs generally efficient servers. Lastly Linux dominates the cloud computing space- a development space that is becoming more popular everyday. | Perhaps the most popular OS system in the world, with machines spread to nearly every corner of the world. Due to this popularity machines for windows can often be found secondhand, or pieces for them. Windows al-so boasts many helpful tools that might scare certain developers away from Linux such as a GUI included, Server Manager tool, PowerShell ISE and Microsoft Visual Studio. All these programs developed by Windows themselves, with regular updates. Windows also tends to be the go-to OS for secure servers and user’s permissions. Lastly windows has great compatibility and additional tools developed by other users. | Hosting from a mobile device is usually not recommended but absolutely can be done. If we aren’t anticipating a lot of traffic on the server and run a very lite and efficient Linux system it could be possible for us. Using a mobile device could give us quite a lot of flexibility in our hardware and the language that we use to run the server. Many of the most popular servers run on mobile devices, like Python, Java, typescript, ruby, and C#. If we were interested in this route we might want to run very rudimentary texts off the server and use something like AWS to run other tasks that might be to taxing for our device. |
| **Client Side** | One of the most intuitive system, often used by people who dislike technology in general. Tends to pay a lot of attentions to the UI of it’s products and goes the extra mile to make them approachable. Apple’s OS though tends to be very popular in the west but not found much in the rest of the world. If we want to develop for a global user-base it may not be wise to use this system. Additionally like we mentioned above the apple ecosystem is a walled garden and only accessible with apple devices. | Similar to the server-side Linux has a lot of the same strengths. Virtually unlimited amounts of customization and community support that rivals that of the other big develops- though with a price tag of $0. Linux also tends to make maintenance very efficient and pushing updates can be done quickly and efficiently. Lastly security, because of the customization it can be extremely secure or entirely open to attack. Extra care must be take when choosing our flavor of this OS. | Windows boasting all of it’s great developer supported tools can be a huge boon for developers, but from a business perspective can be very expensive. Thankfully we have options when it comes to price points. Though because Window’s operates on a subscription model this is a cost that will reoccur on us as long as the server runs. Lastly like the server we Windows can have some of the strongest security. | Mobile devices tend to be very different between the different devices and so from a client side it can be difficult to anticipate what kind of machine our user’s will run.  Additionally mobile devices don’t tend to have a ton of customization and lack upgradability. All of that being said they tend to be the most accessible machines in the world and the vast majority of the world uses mobile devices to connect to the internet. Mobile users do require special UI for their devices though meaning we might need multiple UIs prepared for the site. |
| **Development Tools** | Mac is widely uses Java based systems. Meaning that many junior develops can begin coding on Mac platforms being a very popular language. This means we could keep our costs down with potentially junior developers. Mac also has the benefit of using the IntelliJ IDE, VScode and Replit development environments. Along with these tools MacOS developers have even more tools like Xcode, developed by apple themselves and used by teams of developers to make teamwork even easier. | The preferred system of many developers. Used by many developers for various applications. Many developers have experience with Linux systems. Linux rewards it’s developers with a colossal amount of tools for them. These IDEs also tend to run very lite and don’t require particularly powerful machines. We could definitely use some inexpensive machines and create a great product with them using Linux. | Windows environments use some of the classic languages such as C# and C++. Also utilizes Javascript, Rust, and Python, some of the most popular languages. Many community supported IDEs such as IntelliJ. Also has a developer created IDE with Window’s Visual studio. Visual studio has the potential to work with almost any project with lots of powerful built in tools. | Kotlin, Javascript, Android Studio are just a few of the languages that can be used when developing for a mobile environment. These being some of the most popular languages used today. Mobile application developers have a wealth of tools to work with. As I mentioned above, mobile development is a huge and growing sector of new projects. Mobile applications also can make great use of cloud computing to bolster their strength. |

## 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**: After reviewing the changing needs of the client and where the development team has come in the work on the app, I still believe that a Linux based operating system for our server would be best. Linux can run a very lite server that will keep the game feeling snappy and responsive for user. Additionally the backend code can be written to be OS agnostic and will play well with potentially any platform. Linux also makes use of a lot of popular languages like python, java, swift or .net. This means that we could write the program in a language our team is familiar with to cut down on dev time. Lastly Linux is completely free! While windows and Mac platforms can come with some nice tools to help us out, it is really tough to beat the price tag of $0. Finally, security for Linux servers is naturally high, and there are several security tools we can use to buff of defense up even further.
2. **Operating Systems Architectures**: Ideally I think our perfect system would be a backend Linux server that handles the abstract processes of the game, such as setting teams and pulling images from our libraries for the players. We could pair this with a front-end process that renders the game and UI in all in glory on the user’s machine. Hopefully for this process we would use the client’s own hardware to render this to keep our costs low. Additionally, we break our processes into compartments. This has a few advantages, such as easy updates, increased security, and better processing. This also has the added benefit of keeping the cost to run our server down. As far as I can tell electricity isn’t getting cheaper, the less we can use the better the profit margins could be for our clients. We can sort of double dip, here using the client’s device to buffer the images and render the UI also saves on electric cost. Finally, because of this we could cache data in background and prepare the next image to load during the current round.
3. **Storage Management**: I think using a hybrid system would be most effective here depending on the client’s wishes. A primary database to store user’s information and maybe game statistics could be stored. Personally I think that a single 1 TB SSD would be perfect for our game here. We could spring for other options like HDDs, and NVMes. With cost in mind, the SSDs should be plenty big to support our application and fast enough to keep the game enjoyable for users. In tandem with the user’s front-end rendering using temporary cache memory (RAM) for each game will keep the application snappy and responsive. If *The Game Room* wanted to, we could also create another drive for storing user’s drawings to potentially sell back to user with a T-shirt or picture. Other popular browser-based games like Jackbox utilize similar methods. Lastly we should strongly consider using some form of cloud computing to potentially offload all of these concerns onto a company that will make sure our game runs smoothly from their end. The cost of the service would need to be researched by business development, but cloud computing is growing more popular everyday and seems to be the future of a lot of computing.
4. **Memory Management**: Our memory management would primarily be handled on the client’s own device. This could mean that game performance will vary between users, but it also could save us a lot of money. We could store a small amount of data about each game running in case the user is disconnected additionally if the client deems it necessary.

Additionally, with Linux we have the advantage of paging and swapping. Paging is what we would typically think of when it comes to RAM memory usage, but swapping is something else. Swapping is a potentially powerful tool that transfer the whole process to memory that is not being used if the system needs. For us if we had excess memory this could ensure that spikes could be handled by the server and keep processes running smoothly. On top of that Linux uses Page caching to cache data- like images for potential use by users. This is like preloading images. Finally, something we hopefully will never need to use, but if needed Linux also has the OOM killer. Which stands for Out of Memory (killer), basically this terminates processes to free up memory if none is available.

1. **Distributed Systems and Networks**: Here I think we could make a very strong argument for a cloud-based system. If our app is going to run 24/7 around the world, we could have the app hosted by cloud servers. This comes with a ton of benefits, but the biggest might be that we might become effectively immune to localized power outages. Depending on the cloud service we go with they could have servers in many different countries scattered across the globe. Meaning if one server goes down, they can shift our app to another server. Effectively keeping us online unless all server centers for our provider go down at once. For front-end and back-end communication we can use APIs to asynchronously work with each other. These also have the advantage of being OS agnostic and could be read/run on any device.
2. **Security**: Security will be crucial for us if we plan to store user’s credentials and possibly payment information. First, we will implement a password system for users with an outside 2 factor authentication system. This means that even if we have a security breach the other company, we use to keep data safe will need to be hacked at the same time. While this could be an annoyance to some users it will increase security greatly. Additionally, for us our framework gives players no access to the backend running of the game. While this is a fairly simple and life measure it will matter. Next, we will implement admin accounts who can make changes to the server and parse though the code as they see fit- to push updates or other maintenance. Lastly, another point for the cloud computing option, many cloud servers are already protected and come with some degree of security built in. Finally we can employ our own firewall to ensure that the developers themselves don’t become targets of attacks.