

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 | 01/17/2025 | Christian Clark | Created document |

**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)

Draw it or Lose it is a game application that will be designed for web browsers. This is a game that will allow for multiple different game sessions to happen simultaneously, while allowing for a number of teams for each game and a number of players for each team.

## Requirements

Draw is or Lose it will be a web application. It will be a game application that will allow for a variable number of games, with a variable number of teams per game, and a variable number of players assigned to each team. Each game, team, or player must have only one instance of it at any given time. It must have the ability to host a website and the server for the website. It must have the ability to allow account creation and for users to login to said accounts. It must have the security to support a web application. It must have a team of developers that can develop in Java and JavaScript. It must have the ability to maintain the web application to patch any security flaws that may pop up.

## [Design Constraints](#_2et92p0)

Since Draw it or Lose it must be hosted from a server, the connection speed must be taken into consideration. Without a sufficient connection speed, players will have disconnects or a very bad experience with the game, thus a good connection is one of the most important aspects of a web-hosted application.

Having unique instances of each game, team, and player is essential for keeping track of what is going where, but also for statistics for how many players are playing at any one time. Doing it this way keeps our codebase clean as we will not have to worry about losing track of certain games, teams, or players. Additionally, in the future, we can run statistics on how many players were playing and adjust server costs and upgrade as necessary.

Account creation and logging into a user’s account is very important. As is keeping accounts secure. There will be legal trouble if user accounts are not secure. A properly made database with encrypted user data is extremely important.

Security for the application itself also must be considered. Insecure applications can put both your server and the user’s computers at risk, especially if a remote code execution exploit is discovered. This can lead to legal trouble.

Having a security team for the code base to patch any potential security flaws before they’re a problem is important. Preventative work is better than having to deal with legal fallout from exploits.

Since we’re dealing with a web application, JavaScript developers must be hired to write the web portion of the code. The game will be written in java, but there must be a way for the user to load the game in their browser.

## [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)

Since we’re trying to keep a single instance of any game, player, or team at any time, we must have a counter to keep track of each instance. We use a singleton of the GameService class to keep track of each id of each instance. It is especially important we use a singleton in this way, as we can reference the singleton from anywhere, thus keeping our memory footprint for each Entity minimal, since we do not have to have a reference to GameService so we can get the current id for any specific instance type. A singleton is also useful as we can add to and read from it from anywhere in the code, meaning we can pass around data without needing to pass explicit references to an instance of that class.

To make the code base cleaner, and to make additions to all derived classes at the same time, we make the Entity class as a super class for Game, Team, and Player. This allows us to have the same id and name variables across all three derived classes and makes referencing it easier and simpler. This also allows for making toString easier, as we can define the portion that is identical across the three classes, while making the adjustments for the specific parts.

We have the main class in ProgramDriver, that has an instance of SingletonTester. This is a tester for our program to make sure the singleton works properly.

We have a singleton of GameService. GameService can hold any number of Game, and is the holder for the current id for each Game, Player, and Team. Each Game can hold any number of Team. Each Team can hold any number of Player. All three of Game, Team, and Player are subclasses of Entity.

**"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 could be an option to host a web-service, however the cost for the hardware to run MacOS natively will be far more than the other options. The software will also have to be developed and compiled for MacOS and its proprietary CPUs. Scalability is also a huge issue, as most mac computers, aside from the mac pro, are unupgradable, as such should be avoided. | Linux will allow for completely free hosting, should we have the hardware available. Depending on if we need a GUI or not, we can run the server purely from the command line and possibly run the server headless.  This should be the cheapest option, as there is no cost for most Linux distributions. | Windows is the best option for future backwards compatibility. If something changes with the other operating systems and deems our web software outdated, we will have to rewrite it for those changes. Windows, however, will nearly always be compatible with our software. Windows will usually have a licensing fee for a more professional server operating system. | There should be no reason we should run a web-service on a mobile device or mobile operating system. The hardware will not be sufficient, nor will the scalability be either. |
| **Client Side** | All desktop based operating systems will be mostly similar, with key differences being testing to make sure the JavaScript for the game runs on all web browser engines for all operating systems. MacOS in particular can be difficult, as we have to keep in mind that Safari uses WebKit as its browser engine. Expertise with MacOS and WebKit is a must for making sure that MacOS will be able to run the game, in addition to the extra man hours required to test. | Linux shouldn’t be too much of a difference from the other operating systems as well, as all the code will be run in a web browser. Since this will be our smallest userbase of players, and that Linux web browsers shouldn’t be too different either, lighter testing and costs will be required, if at all. | Windows will be our most popular option for players, and such should be our focus for testing and costs. We only need to test to make sure the big three web browser engines can run our game and we should be set, in addition to any security tests as well. Any of our standard windows computers should be able to test and debug with minimal cost. | Mobile devices are different here. Most of the time, we want an application for mobile games, as web browsers can be heavy on resources and the browsers themselves can mess with the gameplay. We should already have an application developed, but changes to it might be required to transition it over to an online API driven game if it already is not. Developers familiar with Swift for iOS and Kotlin for android will be required. The cost should be moderate, as a lot of changes will have to be made to change the game over to a web API based game, and might take longer as there are two different apps being developed simultaneously. |
| **Development Tools** | Since we are dealing with JavaScript for web browsers, any standard IDE built for JavaScript will be needed. A mac computer will be necessary to test the application on mac hardware. JavaScript, HTML and CSS will be necessary to program a web application of this complexity. The JavaScript IDE will have a licensing cost attached to it, and the cost for a mac computer to test with will be an extra cost as well. | This is extremely similar to the previous and will be similar to the Windows requirement. An IDE dedicated for JavaScript will be required. Knowledge of JavaScript, CSS, and HTML will be required. Again, a licensing fee will be required for the IDE. A machine running Linux will be necessary to test the application on it's web browsers. However, since this userbase is so small, and the possibility of it just working out of the box is so high, deep testing might not be required. | As is the same of the previous, An IDE dedicated for JavaScript will be necessary. Knowledge of JavaScript, CSS, and HTML are required. A licensing fee is required for the IDE. A Windows machine will be required for testing. | A mac computer to compile and publish for iOS is required, as is knowledge of Swift and an IDE for it like XCode. Xcode will have a fee attached.  For Android, any windows or Linux computer will be sufficient, and an IDE for Kotlin will be required. This most likely will have a fee attached depending on which IDE is chosen.  Since we are dealing with 2 different apps on 2 different operating systems, we will need 2 different teams working for each app. |

## 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**: Linux is the best option for our purposes. With a Linux server, we can use the hardware we want, which will give us the freedom of spending, and allow for future expansion. The operating system is free, and we can write our server software in house lowering the costs further.
2. **Operating Systems Architectures**: Since Linux is a free operating system, we can use nearly any distribution we want and can be picky about which one will meet our needs. We can use any hardware we want and will allow for future expansion. The costs will be much lower than the other options, especially MacOS, where the hardware for the servers alone will be a huge portion of our budget. Expandable storage will also be a major boon, since we can expand as we grow, leading to lower initial costs, and being able to spread the costs of expansion over a longer period of time.
3. **Storage Management**: We shouldn’t need a massive amount of storage overall. Just enough to cover our user database and any miscellaneous storage. The most important parts we need to be aware of is storing the data for the website and the API that the clients will be interacting with. A simple raid of hard drives should be sufficient with a third-party storage management system for the server itself.
4. **Memory Management**: We’re going to have to be keeping track of a variable amount of games, with a variable amount of teams, and a variable amount of players on each team. If we’re using a modern programming language like Java or C# for our server software, we can take advantage of its garbage collection feature to automatically get rid of any memory we fail to clear after a game is finished. Additionally, the game is based on having the users draw an image, so we’ll have to be constantly sending and receiving either image data or drawing instruction data to be able to recreate the image on other user’s devices. This can create problems, but if we use memory management techniques like lazy loading where we’re only holding the latest frame of the image in memory, we can minimize the amount of memory we use for each running game.
5. **Distributed Systems and Networks**: We’re going to have to communicate between different applications from different operating systems, therefore we are going to have to make a unified server base that all platforms can communicate with. Making an API that can send back data in the same format for all platforms that those platforms have to decipher and use is much better than having to do all the work on the server’s end. The servers will have to communicate between the user database and the users connecting to it at the same time acting as a middleman between the users and the data. Outages are going to be an inevitability, so having an error page built into the user clients is a must. The users need to be aware if the service is down, and having an API option to display scheduled maintenance on the user’s devices is also necessary.
6. **Security**: The first line of defense is in the software the users will be interacting with. We need to make sure all exploits are patched, and any unknown exploits are discovered as soon as possible. If our front end is secure, having our employees aware of phishing and any social engineering attacks criminals might try is essential. After that, having our user database being encrypted and salted will prevent criminals from gaining access to our user’s data should they somehow get past our other lines of defense.