

Draw It or Lose It

# CS 230 Software Design Document

Version 1.0

## Table of Contents

[CS 230 Software Design Document 1](#_Toc140830442)

[Table of Contents 1](#_Toc1087513792)

[Document Revision History 2](#_Toc1043098394)

[Executive Summary 2](#_Toc938275793)

[Requirements 3](#_Toc245729860)

[Design Constraints 3](#_Toc2128517166)

[Rationale 3](#_Toc1961155738)

[Recommendations 8](#_Toc550674612)

## Document Revision History

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 09/14/2022 | Zach Behrensmeyer | Initial write up of software design document for draw it or lose it |
| 1.1 | 10/2/2022 | Zach Behrensmeyer | Added evaluation and recommendations |
| 1.2 | 10/10/2022 | Zach Behrensmeyer | Revision of evaluation |
| 1.3 | 10/13/2022 | Zach Behrensmeyer | Revision of Executive Summary |
| 1.4 | 10/15/2022 | Zach Behrensmeyer | Added missing Domain Model and System Architecture sections and final revisions on the file |

## Executive Summary

Our customer, The Gaming Room, has an Android based app, “Draw it or lose it,” that is based on the 1980’s tv show *Win, Lose or Draw.* They would like to make a web-based version of this app that is available on multiple platforms to increase its popularity and user base. The premise of the game is to guess what an image is as it renders within a given 30 second time limit over the course of four rounds. There are multiple teams and if the team currently guessing fails, the other teams have an opportunity to guess and steal the point within a 15 second time limit. TGR is unsure of how to set up the environment and has reached out to us to help them streamline this process.

In order to assist them in reaching their goal we will use a distributed application architecture to allow multiple players to connect from different platforms such as Windows, Mac, and Linux. We first will authenticate a player and if it succeeds allow them to connect to a single game session that is secured and managed on one of The Gaming Room’s servers.

## Requirements

* A game 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 any given time. This can be accomplished by creating unique identifiers for each instance of a game, team, or player.

## Design Constraints

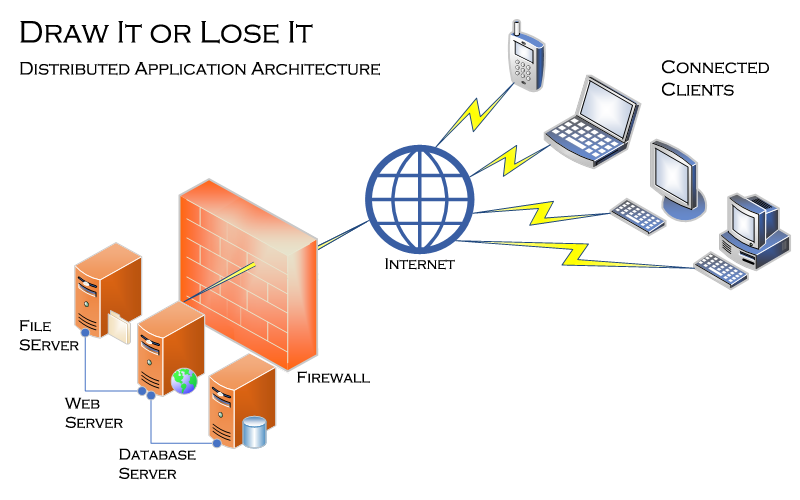
* The client wants to develop a web-based game app in order to increase the player audience. A web application must be built, and a server system will need to be invested in. It could be serverless technology with AWS or a set of purchased servers to run the software. We will need a server OS to host the site on as well.
* The game must have at least two teams.
* Game and Team names cannot be the same to avoid redundancy, this means we need to keep track of names for users to see if the in-game name can be used
* Each game instance is unique, we will need to make sure we can differentiate between the different games playing on our servers.
* We need to make sure we can effectively at the same time render the image for the current round for all players.
* We need to implement a firewall for security, a web server for the app, a file server for holding files, and a database server to host our database.

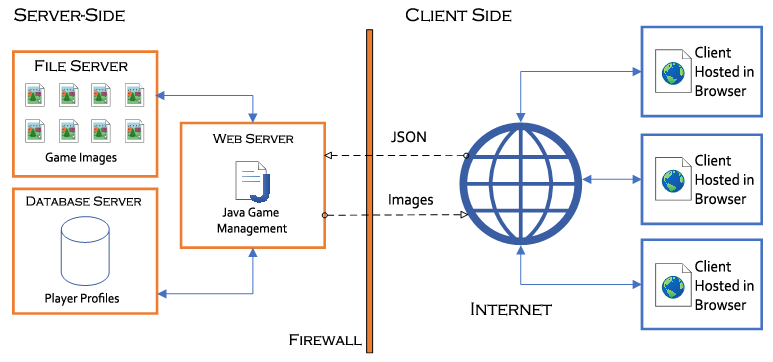
**Rational**

* The first constraint is the game needs to run on multiple platforms, outside of the general design constraints for the game there are also hardware constraints in play.
* This game needs a bare minimum number of players, there should be at least one team but that implies more than one player. We will need to test game lobbies of varied sizes and different operating systems to make sure it works across the board.
* The team and game names being unique does not necessarily make the code harder to design but we must keep in mind that there cannot be any duplicates, or we will encounter issues in the code.
* (Covers bottom 4 design constraints) We are going to be supporting many different operating systems, so this poses the question of what languages are we going to use that is compatible amongst all Operating Systems, we need to find out what type of server and hosting software we want to use that can allow clients from any OS to connect and play seamlessly, and make sure that there are not any OS related bugs with the finished product. We need to make sure it functions across the board. When I was reading on the internet about different hosting software Mac did not pop up as much as Windows or Linux, there appears to be more support and more hosting options available for the game. On top of this I learned that Linux and Windows communicate more easily than Mac and other Operating Systems.

## **System Architecture View**

The architectural diagrams provided below show how we can achieve *The Gaming Room’s* goal. We will have an array of servers, one file server to hold images and other files, a web server that manages the sessions, and a Database server that can be used to hold the data from the players. They can then connect to our server over the internet that is protected by a firewall and other security measures. This allows them to authenticate themselves and get into a game safely and securely. (Images provided by SNHU)





**Domain Model**

In the UML Diagram below we can see the Entity class is inherited from the Game, Team, and Player classes. This means Entity is acting as a super class to them. Entity contains a private id of type long and a private string called name. We can see it also contains a private empty constructor, a public constructor with an id and name. The Entity class also has a public getter in order to retrieve the name or id for whatever is being requested. There is also a to string method in order to concatenate the name and id into a string and be returned. Game Service references game, which references team, which references player. These references are aggregation. Game Service has a private list of games, The next game, team, and player ids and a private static variable which will be set to itself through the getInstance method since it is a singleton. There will only be one game service and it manages all games. It has an add game method via a name, it has 2 get game methods one that takes an id and another that takes a name. It can get the current count of games and has getter methods for the next game and team ids. Game has a private list of teams, a public constructor that takes an id and name, you can add a team to a game, and it offers a toString method like Entity. The team class allows the code to add a player to the team, Team has a constructor that also takes an id and name, it has a method to add a player, and finally Team has a toString method to return a string of both the team id and name. Finally, the Player class has a constructor that takes an id and name and a toString method to return player name and Id as a string. The last classes to mention are ProgramDriver, and Singleton tester. Program Driver is where the main method is located. It is the entry point of the Java program. It uses the singleton tester class to test our singleton GameService. The Singleton tester class has a uses relationship with ProgramDriver. We set these classes up this way so we can allow for more than one game to be running with multiple teams consisting of multiple players at the same 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 

**Evaluation**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| **Server Side** | Pros:  Mac offers a user-friendly GUI which can make it easy to use. It has a file system that is easy to navigate. It also offers a terminal that you can use to configure the server and make changes. You can easily make upgrades.  It runs on the same BSD system that Linux does so it should be stable.  Cons:  It is expensive, and not a popular option for web hosting, I was not able to find much about it compared to Windows or Linux. Does not have full normal access to Docker for creating containers.  Additional context:  Java is already available and can be used to write rest APIs to receive communication and send updates to the client using an IDE and JDK.  We could use Maven and Dropwizard to help build a web app. For this section we can verify users and let them in to games.  We can also include the use of JSON if we want to include payloads of data | Pros:  Linux is free and open source; it also is relatively well known for its stability for hosting websites. You can use Apache or Nginx for web hosting as well. It is also nice that the admin can go in and turn off certain software so the main app you want to run can run better. Is a UNIX derivative, meaning it is a continuation of UNIX design. Docker probably works best for Linux over Mac and Windows for creating containers.  Cons:  There is not LTS on all versions, and it is not very beginner friendly. You must use a command shell to configure, meaning it does not provide a user-friendly GUI.  Additional context:  Java is already available and can be used to write rest APIs to receive communication and send updates to the client using an IDE and JDK.  We could use Maven and Dropwizard to help build a web app. For this section we can verify users and let them in to games.  We can also include the use of JSON if we want to include payloads of data | Pros:  Windows Server is very similar to the OS on most people’s home computer making it very easy to use and offers IIS for hosting. It offers more software to run. Has access to active directory and offers Windows Defender out of the box. Has access to Docker for creating containers  Cons:  Windows does not appear to be as stable as Linux has proved to be for web hosting, with a bad history of blue screens and bugs, however it has improved dramatically over the years. This OS is not as secure making Docker not a viable option for creating containers. Windows Server also costs money.  Additional context:  Java is already available and can be used to write rest APIs to receive communication and send updates to the client using an IDE and JDK.  We could use Maven and Dropwizard to help build a web app. For this section we can verify users and let them in to games.  We can also include the use of JSON if we want to include payloads of data | Pros:  Many people have smart phones, this approach offers a wider range of portability, and can be cost effective as phones are much cheaper than servers.  Cons:  It’s not a great idea to have a mobile host. You should have it in one place so it can be tracked and secured easily. This approach is easily not as great as an option as the other 3 OS’ offer a better hosting option, and it has the poorest security option on the list of options here.  For ios we could use swift and JSON to communicate with the other operating systems. We can include a payload of data in a post request so the server can then handle that data and respond.  Android has access to an Android Maven library which could be put to use to follow a similar pattern to Windows, Mac, and Linux. |
| **Client Side** | The cost for client-side development is comparable to windows but depending on the language chosen it may require a more specialized developer, costing more time and money. This will require a decent level of expertise with the OS.  We can use HTML and CSS to create UI. We would then need to make sure the UI looks good for Mac, all of the different OS will probably be a little different and have access to different libraries. We can use Javascript to call the Rest APIs with POST and GET methods. | This is probably the least commonly used OS of the big 3 this one may require more time and money to write the code and test it since it is the least common. The languages that can be used on here are similar to windows, but OS wise will cost next to nothing since it is free and can be installed on most machines.  Here we still use HTML, CSS, and Javascript to create UI. We would then need to make sure the UI looks good for Linux. Linux in specific has been an afterthought when it comes to access to libraries for decent UI. We will need to make sure that the UI still looks good and all components can communicate with the server. We can use Javascript to call the Rest APIs with POST and GET methods. | This is probably the most common operating system people are taught software development on, and would likely cost the least amount of time and money to use. It arguably has access to the largest amount of tools and languages. However, one I think would cost less than Mac, but the price would be relatively similar.  Like the other platforms we will use HTML, CSS, and Javascript to create UI. If we start on Windows, we just need to start off by making sure it looks good and everything functions. We can still use Javascript to call the Rest APIs with POST and GET methods. | This one would likely require the expertise of a mobile developer which if we don’t have would require more expertise and time. The benefit is that this can be used from almost anywhere since it would be on a mobile device.  Things we can do to make this web app better is to make it responsive so the app resizes based on the size of the browser. This makes it more friendly to mobile users on many different devices. We should also improve the UI to work better on mobile devices. If there is important information or buttons we should probably move them based on screen size since we don’t want the app to be overwhelming and hard to navigate on mobile. We should also plan on using larger text and buttons for people to use since the screen is smaller. |
| **Development Tools** | This game was made on Android, we can use Java to integrate Web APIs to send and receive updates between the host and clients. We can use IntelliJ or Netbeans to write Java in. For front end development we could use the standard HTTP, CSS, and Javascript plus supporting libraries like bootstrap to make it look nice. We unfortunately lose access to Notepad++ here, but we have access to Sublime, Atom and Komodo edit for text editors to write our web languages. Natively has access to MySQL Workbench for database storage. | To integrate with the Android app on Linux we have access to IDEs that allow for Java development like Netbeans, IntelliJ, and Eclipse. Once again, we can use this language to send and receive updates between the host and different clients.  For front end development we could use the standard HTTP, CSS, and Javascript plus supporting libraries like bootstrap to make it look nice like a game.  Notepad++ is available on Linux and is a very popular and commonly used text editor that has access to many languages.  Natively has access to MySQL Workbench for database storage. | Windows is easier to use than the other operating systems in my opinion. It arguably has access to most IDEs and support groups to help build the app. It has access to IDE’s for java development like Eclipse, Netbeans, and IntelliJ.  It would also use the standard, HTTP, CSS, and Javascript plus supporting libraries. It has access to notepad++, visual studio code, and Atom to aid in the writing of the front end and responses from users.  In terms of database options, in addition to MYSQL it has access to Microsoft SQL Server another very popular Database tool. | The mobile options are probably the most limited with iOS allowing the use of Java and Swift, and Android with Java. Android studio can be very useful as you can emulate mobile devices on your computer. This would allow us to go out to the website on an emulated device to test the website with different emulated devices.  Once again for front-end development you can use the standard, HTTP, CSS, and Javascript plus supporting libraries and depending on the OS this is being written on we can use xcode, notepad++, or sublime text editors to do the front-end work.  If for some reason, we were hosting the website on a mobile device we could use SQLLite for database storage. |

## **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**: My opinion is that the Gaming room should start on Windows over Linux and Mac. Linux does not provide the easy start that could come from Windows or Mac. Windows and Mac will generally cost roughly the same, but Windows requires the least amount of expertise, and it will cost less to get the project going. Windows offers most of the same software that Linux would have and more and its native to Windows. It also has access to a user interface for administration services, so it does not require you to edit any files. Everything is pretty much click and go thanks to this. Also, Windows is owned by one of the largest tech companies in the world, Microsoft, it is likely they will continue to make advancements and additions to their operating system software. They often provide free OS updates making your machine more secure. Another useful software, IIS, can be used to create a web server environment and will easily allow someone to create a new site.
2. **Operating Systems Architectures**: The benefit of starting with Windows is it is very easy to use and offers a user-friendly GUI, Although Mac does as well it is not as popular for hosting websites, although it still has a very large user base, and we should expect many players with this OS. Linux lacks a proper GUI and is not as easy to get started. You must configure your web app through a terminal and while it offers a lot of customization it still is not recommended for someone new. Windows also allows administrators to create accounts and to access system resources through the GUI. This includes file management, meaning you can view files, images, and even have access to a messaging system. It is also possible to give higher access to control the memory to best suit specific needs for what you are hosting. On the other hand, if someone wants to use windows but it used to Linux can use the command line to still set up the server making it more flexible for developers.
3. **Storage Management**: The nice thing about Windows is it comes with Storage Sense and OneDrive. These are file management and storage tools. OneDrive allows you to store files on the cloud. Storage Sense allows you to automatically free up drive space by getting rid of temp files and clearing out the Recycle Bin. On Windows you are also allowed to choose the save locations for your apps running on the server, making it easier to manually find files. These built-in features are super helpful for when you are working on a large project, so you know exactly where everything is. As previously mentioned, the files could be compressed or use a file type that allows the images to be smaller such as AVIF or JPEG. This is one of few ways we could be allowed to save space. Additionally, it's important to note that in a serverless implementation, storage could very easily be increased and decreased as needed with your vendor of choice. It would increase or decrease in price monthly depending on how much space you are using at the time. With the combination of an ICDN alongside proper storage such as an SSD or a cloud service we could very quickly load images for the users.
4. **Memory Management**: The Gaming Room has expressed interest in having images that render over time. This is going to require the use of a file system and some type of method to tell our users to load. Specifically, we need to worry about what is the best way to load images that are randomly selected. It may be best to render these images on the server side and send updates to the clients. Users can have any device, perhaps the newest smartphone, or a 10-year-old laptop that may struggle to render the images. We can do this on the server side. We can build an Image Content Delivery Network by creating a subdomain that hosts the image files we have to share with the game sessions. This should allow us to easily and quickly render our images for our clients. This is because it does not use as many bytes to transfer the files. So, when a round ends or someone guesses correctly, we can use the server to tell all clients to load a new image from the ICDN. We can also use browser caching. Another approach would be to preselect the next x number of rounds images at the start of the game. We can then tell the clients to cache the next x images, so it more easily loads. This factor is more dependent on the user devices and what browser they are using. Chrome for example allows the browser to use up to 80% of disc space on the user's device. Firefox allows up to 50%, and Safari allows 1 gig before prompting the user to increase in 200MB increments. If we combined the use of the two methods with image compression here, This should mitigate any issues for our clients rendering the images.
5. **Distributed Systems and Networks**: For Draw it or lose it we will be building a client-server application. All the users will have a client application that will communicate with and depend on the server they connect to (Depending on if there is one or more servers they can connect to). We will have a web server, a database server, and a file server that will all need to be able to communicate with each other over the network. We would need to make sure that our servers are up to par with the number of players we expect to see, if our servers can’t handle the players then we may experience outages and upset the player base. This could be a decent option for a serverless environment perhaps run through AWS so that way on the fly we can increase our Memory or Storage at the cost of a higher monthly rate. We could always lessen the amount if our player count dwindles. Then there is no need to pay for high costing servers, and maintenance on said servers. AWS is also very secure and will try its absolute hardest to make sure we do not have any outages.
6. **Security**: Another nice thing about Windows is that it comes with built-in protection software called Windows Defender which will Scan the machine for malware, viruses, and other security issues. It is recommended to pair this with something else. Perhaps some kind of firewall can be implemented and require users to log in so we can monitor what the users are doing. Overall, something needs to be done to secure sensitive data. We also need to make sure we regularly do Windows updates to battle against other vulnerabilities. Upon releasing it we should hire a third party to perform a penetration test against our app to make sure it is hard to get into. We do not want black hat hackers to use this site to get to our systems and do malicious things. We will also need to add encryption to our site so that way we can protect our user's information. Windows unfortunately has a higher risk of security breaches so having good security is a must.