

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.0

## Table of Contents

[**CS 230 Project Software Design Template**](#_l6ti7uoag22u)1

[**Table of Contents**](#_30j0zll)2

[**Document Revision History**](#_grjogdjh5fi8)2

[**Executive Summary**](#_sbfa50wo7nsh)3

[**Design Constraints**](#_2et92p0)3

[**System Architecture View**](#_ilbxbyevv6b6)3

[**Domain Model**](#_8h2ehzxfam4o)3

[**Evaluation**](#_2o15spng8stw)3

[**Recommendations**](#_m8aleynsvzvc)5

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 07/15/21 | Anthony See | Starting creation of the ‘Draw it or Lose it’ environment game |
| 1.1 | 07/31/2021 | Anthony See | Adding recommendations for OS, memory, network, and security |
| 1.2 | 08/11/2021 | Anthony See | Revision of the OS, memory, network, and security recommendations |

## [Executive Summary](#_sbfa50wo7nsh)

The problem that this document covers is the creation of a ‘Draw it or Lose it’ game. The client already has an android version of the game but requires it to be compatible with a web-based application. The client does not know how to create the environment. The environment will be created by Creative Technology Solutions (CTS). To create the environment, variables and a simple backbone classes will be needed. This is to create the environment as close to the android version as possible. No code is needed for this. Other required information includes the max number of teams and the max amount of players per team.

## [Design Constraints](#_2et92p0)

Needs to be web-based so multiple platforms can run it.

Client side cannot be hardware intensive.

Needs to be created within the limitations of the web application.

Needs to allow users and teams to choose different players, teams, games.

Will not allow entities with the same unique ID to be created.

## [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 classes *Game, Team,* and *Player* have ‘is a’ relationship with *Entity.* Therefore, each of the three classes are *Entities.* They inherit from the *Entity* class as they all have common variables. This, also, means that the class *Entity* is a super class.

The relationship between *Game* and *Team* classes is the same for *Team* and *Player* classes. The *Game* ‘has’ *Team(s)* and *Team* ‘has’ *Player(s).* The *GameService* class, also, ‘has’ a *Game.*

Therefore, the class *GameService* can have multiple *Game(s). Game* can have multiple *Team(s). Team* can have multiple *Player(s).*

The classes *ProgramDriver* and *SingletonTester* are separate from the game classes. They both of a ‘usage’ on each other. They are used to test whether a game already exists or not.

****

## [Evaluation](#_2o15spng8stw)

**Server Side:**

There are server capabilities for every operating platform for the web-based service. Most operating platforms that have a web-browser offer a connection to the internet. The internet is the only feature needed, aside from the browser itself, to make the game work. Only a single server is needed for multiple platforms as it can talk to all of them. There’s no need for a server per platform.   
 The only costs to the client would be the operating system chosen to run the server, plus any hardware for the server. One choice is Linux because the open software it uses, which in turn, makes it free. However, it does have less features than a Windows based server. Windows allows more security, more support, and is compatible with most software. This part of the budget is up to client, but the recommendation here would be a Windows server.

**Client Side:**

To make sure the application is compatible with all web browser platforms is to use a language that all of them understand. Or, to use the most common languages and build the application to suite the browser better. The first way will take the shortest amount of time, but it may be difficult to get it to work on every browser. The second way takes longer, but makes sure that each browser can run the program. After the application has been built, the next step is to test it. Developers never know if anything works if they don’t test it. One, two, or three computers would be needed as well as an iOS device and an Android device to test the application. Each computer would be running a different OS(Windows, Mac, Linux). Multiple browsers would be tested, specifically the top performing browsers, for each OS. The reason why one computer is an option is one might opt for virtual machines.

**Development Tools:**

*Windows and WebBrowsers: Visual Studio: JavaScript*

Windows and Visual Studio are both made by Microsoft, so they both work well together. This pairing is among the easiest to deal with. This is the recommended option as almost every application built for any operating system or browser can be made on Windows in Visual Studio. JavaScript is chosen here because it is widely used in web applications.

*MacOS/iOS: xCode: JavaScript*

xCode is chosen as the IDE for MacOS and iOS systems, as xCode is made by Apple, so they do work well together. JavaScript is the chosen language as there won’t be any difference in code between Windows and Mac, aside from file paths. JavaScript can, also, run on iOS as an application, aside from a few minor adjustments to the code, paths, and user interface. It will certainly save time using JavaScript, rather than build an entirely new application with different code.

*Android: Visual Studios: JavaScript*

Visual Studios is chosen for Android because of its uses across other platforms. Since Android can run the language as well, JavaScript will be chosen as the language. The code will be easy to create as the developer for Chrome, the most used web browser, develops the Android operating system.

*Linux: Visual Studio: JavaScript*

Visual Studio is chosen again as the recommended IDE because of the previous choices. JavaScript is chosen again because of the wide uses it has with the web browsers. If a pure Linux application were to be created, C would be preferred.

Visual Studios does have licensing fees for creation of software deemed for public sale and release. The cost is $45 per month or $250 per month for enterprise subscription. However, the licenses do carry over to other operating systems, but only per account/person. xCode is free only until the developer needs to release the app, then it is $99 a year. xCode is not transferable between systems as there is no way to run xCode on Windows or other systems unless a virtual machine is used. It is, also, recommended to not create a purely Linux based application, as Linux cannot easily run JavaScript without prior software installation. It is not ideal to recreate the environment in a different language for a small portion of the audience, unless the time was given for this. The application would have to be tested thoroughly as well, as JavaScript and C based languages run differently.

Two teams of 5 or more with 20 weeks would be ideal to create the application. Two weeks to create the environment and one week to test it on a single platform. Then a week would be needed to port the game to other operating systems and browsers, each needing about a week for testing to make sure the application works. The rest of the time would be needed to create the server program to run the game, however, this would be created alongside the game environment, with a different team, as the game needs the server to run.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Terminal commands are flexible. This allows settings of the server to be changed easily | Terminal commands are flexible. This allows settings of the server to be changed easily. It is more cost effective. | More compatibility between software. More applications are available as well. | Not as good as other OS’s. It would be ideal to have the server in a single location without it moving. |
| **Client Side** | Need to know what you are doing. It is a little difficult to get started. Cost is moderate, equal to Windows. | Need extensive knowledge on how to use it. It is difficult to get started. Lowest cost amongst the four. | Do not need to know much. Easy to get started. Cost is moderate, equal to Mac. | Is easy to update. Is malleable for clients and developers. Is a bit more difficult to get started. |
| **Development Tools** | Visual Studios, Eclipse, Xcode, PyCharm, NetBeans.  Java, JavaScript, SQL, C, Python | Visual Studios, Eclipse, Code::Blocks, NetBeans, PyCharm  Java, C, Python, Google GO | Visual Studios, Eclipse, Code::Blocks, NetBeans, PyCharm  Java, JavaScript, Python, C, HTML, Ruby | Visual Studios, Xcode, Android Studio, WebStorm.  Java, Python, C, Kotlin |

## Recommendations

1. **Operating Platform**: Windows is suggested here because of the wide range of support, as well as the ease of using it.
2. **Operating Systems Architectures**: The Windows x86 architecture is recommended because it is easy to use and is guaranteed to work along with any computer, windows or not, and any architecture.
3. **Storage Management**: There’s two ways to decide secondary storage for this game. If the game and the pictures collectively don’t go over one or two terabytes of total data, at least four, one terabyte solid state drives in RIAD 10 would be recommended. The RAID helps prevent the loss of data if one drive were to fail. The RAID system would handle all the communication between the drives and knows where the data needs to go, so there’s no need to worry about that.

If the game were to be expanded over the course of its lifetime by adding features and pictures, it would be recommended to have a mixture of solid state and hard disk drives. Two, two terabyte solid state drives would be used in RAID 6 to store used information for the day’s operation. Four or more eight or 16 terabyte hard drives would be added in a RAID 10, separate from the RAID 6 system to store data that’s not being used. Example: A collection of December pictures that’s not being used because it is July. Once the days operation is done, the system would pull new pictures for the next day, replacing them the ones in the RAID 6 system.

The reason for the large amount of data storage in both systems is because it is relatively cheap to purchase and maintain secondary storage. For a one terabyte solid state on Newegg, it is less than $100. For an eight-terabyte hard disk drive on Newegg, it is less than $190. This portion of the storage management is still flexible, but above is what is recommended for the game.

Both systems will use one of two directory structures. One system should only be used in the low storage configuration, as it is a purely, first level, random structure that names the pictures randomly and stores it. This is a valid system; however, it would not be ideal because of the time spent searching for said picture. It would, also, be almost impossible to create themed pictures with processing time in mind.

The second directory structure would consist of a tree architecture for the directory. Here, the pictures would be stored in sperate folders that are within folders. Example: December->Christmas->25th, or Holidays->December->Christmas. The system would not need to search as much as in the other architecture.

For the management of the system, a mixture of Python and C++, or just C++ would be used here. Python would be used for it’s fast processing time of finding and loading information, which would be passed off to the C++ portion. C++ would be used here because of its power over memory management. The whole system could be done in C++ as well, as the only feature of Python is the faster information processing.

1. **Memory Management**: A large amount of RAM would be recommended. The exact amount would be unknown until an adequate amount of data is received from the game. At least 128 to 256 gigabytes of memory per server should be used to start, depending on the initial picture size. The reason for the unknown amount is we simply do not know how well the game will do on launch and thereafter. Each game session would be held in memory for fast response time, as well as two pictures per session, one being drawn and the other queued up. Using an average of 2.4 megabytes per picture and 256 gigabytes of memory, would yield just over 100,000 available sessions per server. This does not include the workings of the game itself and the systems needed to process the data. Using 256 gigabytes per server, also, means that an upgrade to put more in would not be needed. This would create a large amount of down time or high latency for players.

Same to the storage management, a mixture of Python and C++, or just C++ would be used here for the same reasons. Python for its fast-processing time, and C++ for its power over memory management.

1. **Distributed Systems and Networks**: As the game will be web-based, it will need to be cloud orientated with the servers picking up the work and processing of the games. Therefore, it is recommended to have multiple servers to run the game. This is just incase a server crashes, a drives fail, they need to be upgraded or other events including natural disasters, another server can pick up the slack. Having multiple servers, also, improves latency and response for the users of the game. This is a big deal because having low ping can be the difference between winning a game or losing a game. If ping is high, players will become frustrated and won’t play.

Like stated in the other sections, hardware required to create the servers is strongly dependent on how well the game does. What was recommended is what we recommend for hardware for the servers. The central processing unit, CPU, is a different story because of the amount of users on each server, the encoding of the pictures and the amount of work it needs to do. Several CPU’s per server would be ideal as how fast response time to users is directly linked to how fast a CPU processes threads. A CPU with a large number of cores, 10 or more, and with high frequency, 2.2 Ghz+, would be ideal. Since there is such a large number of different CPUs to choose from with a wide range of cost, it should be looked at more closely with more information from the client. With that being said, a cost effective Intel Xeon processor, or an AMD Epyc CPU should be chosen for the job.

Java would be used to handle anything else the server needs. If the work is internal, as if data doesn’t leave the server, it can be worked in the existing systems with C++. Any other needs, like talking to other servers, sending data, or receiving data over networks and the internet should be done with Java, because of the security it features.

1. **Security**: Java will be used for the security portion of the environment because of its high security features. A login system will create a role hierarchy for the game. Admins are given different options than Developers; Players are given different options than teams; Users are given different options than current players; and so on. The number of roles and the attributes of the roles may change during development and in later revisions. Java will, also, be used for any over-network communication, as stated in section 5.