

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 | <07/14/22> | Jayden Chavarria | Added brief summaries and design constraints |
| 1.1 | <07/26/22> | Jayden Chavarria | Research on the development requirements of MacOS, Linux, Windows, and mobile devices in Server and client capacity. |
| 1.2 | <08/02/22> | Jayden Chavarria | Provided reccomendations |

**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* has tasked us with creating a cross-platform web-based game based on a 1980s television game *Win, Lose or Draw*. They would like the game to be accessible on multiple platforms such as iOS, macOS, Linux, and Windows. The game they want to base off of is exclusively available on Android devices at this time. The game should include teams facing off to guess images that are being slowly drawn (30 seconds) from a random set of stock images. The teams will only have a set amount of time to guess (15 seconds) and if the team cannot guess within the allotted time, the other teams are allowed to guess and whoever guesses first gets the point. There will be four rounds each lasting about a minute.

## [Design Constraints](#_2et92p0)

* There needs to be a large library of stock images within the application to pull from to add randomness and replayability to the game. These images need to be able to be formatted to each individual device such as mobile phones, computers, tablets etc.. These images must also not be prone to copyright as to avoid any legal trouble.
* With the game already existing on an Android platform it needs to be almost identical but reformatted for other devices such as a laptop or a computer or other big tablet devices. The user interface should remain consistent through each iteration of the game no matter what platform, if need be a total redesign of the applications user interface should be kept in mind.
* With the game being on multiple platforms the Android app needs to be transferred to a web-based program so it can be streamed to any supported operating system to allow for cross platform playability.
* While designing the game there needs to be a system to check for multiple teams and multiple players per team making sure that each team member and team is identical as to discern them within the game. The server housing the program needs to be able to handle Access from multiple different platforms and have a large enough server capacity to hold the players.

## [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 Unified Modeling Language (UML) below allows us to see how The Gaming Room software will be developed. Encapsulated in the com.gamingroom package we have forms of inheritance shown mainly through the Entity class and abstraction through private data and variables. Encapsulation really only involves keeping the classes each with their own variables and methods all in one area to stay organized. The Game, Team, and Player Classes all inherit from the Parent Entity class which helps reduce redundant code being written in each class, which helps cleans up the code. The associations between GameService to Game, Game to Team, and Team to Player all have a 0…\* which just means that there can either be 0 instances of whichever class to many of them. This mean that there can be many games in the GameService or 0, there can be many teams in a game or 0, and there can be many players in a team or 0. The program only wants one instance of the game in memory, so we use the SingletonTester to test the ProgramDriver which has the main() function. This is shown in the UML as the ProgramDriver Class with a composition arrow pointing to the SingletonTester. Abstraction can be found in almost all of the classes including ID’s, names, and the game service. The reason why these variables and methods are private is simply that the user does not need to know this information and helps run the program without unnecessary information going out to the user.

**"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** | **Pros**   * MacOS is known for its heightened security * Can easily deploy a server that can run IOS for roughly 20$ * Vast technical support readily available either through people or products but this can cost quite a bit if used * There are MacOS server resources available through their Apples site.   **Cons**   * Apple doesn’t like to share so an Apple device is needed to run MacOS or iOS which leads to limited devices applicable * Any updates are made solely through Apple and any software updates need to be paid for * Very expensive due to proprietary systems | **Pros**   * Very open-Source software leading to a large community to consistently update against security threats * Linux can be used on many different devices which lends spending up to the client * No central controller of the software so there are no proprietary charges * Overall, the most cost-[effective](https://aws.amazon.com/free/compute/lightsail/?trk=6f75e631-3b71-4d44-a71d-d557bcd37563&sc_channel=ps&sc_campaign=acquisition&sc_medium=ACQ-P|PS-GO|Non-Brand|Desktop|SU|Websites|Solution|US|EN|Text&s_kwcid=AL!4422!3!531871356404!p!!g!!server%20for%20app&ef_id=Cj0KCQjwof6WBhD4ARIsAOi65agIVCb3ozoLibfq1YonruUgEffDTWMjAVH-OdBV7jv6dq7D09tbs9oaAkwVEALw_wcB:G:s&s_kwcid=AL!4422!3!531871356404!p!!g!!server%20for%20app) system since Linux License is free (server cost ranges from 3.50-160 for virtual cloud server)   **Cons**   * No large company backing the OS leads to a lack of software available and compatibility is only available when the users make it so | **Pros**   * A multitude of hardware options * Updates are more frequent due to a larger user base (compared to apple and Linux) * More software support again due to high user base   **Cons**   * Looking online the servers can be very [expensive](https://aws.amazon.com/free/compute/lightsail/?trk=6f75e631-3b71-4d44-a71d-d557bcd37563&sc_channel=ps&sc_campaign=acquisition&sc_medium=ACQ-P|PS-GO|Non-Brand|Desktop|SU|Websites|Solution|US|EN|Text&s_kwcid=AL!4422!3!531871356404!p!!g!!server%20for%20app&ef_id=Cj0KCQjwof6WBhD4ARIsAOi65agIVCb3ozoLibfq1YonruUgEffDTWMjAVH-OdBV7jv6dq7D09tbs9oaAkwVEALw_wcB:G:s&s_kwcid=AL!4422!3!531871356404!p!!g!!server%20for%20app) (around 8-240$ per month or around 6k$ to own your own decently stacked server) compared to Apple and Linux * Updates only come through Windows services * Known security flaws when compared to Apple and Linux | **Pros**   * Cost is relatively low but would depend on the traffic of users on your server * [Price](https://imaginovation.net/blog/importance-mobile-app-maintenance-cost/#:~:text=An%20app%20hosting%20server%20can,as%20a%20cloud%2Dbased%20environment.) can range from 70-320$ per month but cost can be covered by users making the price null * Servers can be easily expanded or minimized based on project size and user traffic * Code will not be seen by users * Mobile operating systems are readily used so finding support for applications is relatively simple   **Cons**   * Integrating physical or cloud servers is almost mandatory because mobile devices are not meant to carry high-capacity server loads |
| **Client Side** | **Pros**   * Development time and deployment is longer than Linux and longer than windows but isn’t long * Relatively easy to test software between multiple browsers * Web browsers available are dev friendly * Cost of developing on client side is also relatively low since MacOS is relatively big and has people specialized in it   **Cons**   * Again, an Apple product is needed to run MacOS * You need someone who specializes in MacOS if you want to outsource work which isn’t as hard as Linux but harder than Windows | **Pros**   * Open-source system allows for cost to stay low since it’s not controlled by a larger company * Also support web browsers with dev friendly tools * Dedicated developers can make development quick   **Cons**   * Has less users than Apple and Windows so finding a specialist that is both highly experienced and well versed with Linux can be difficult | **Pros**   * As the most used OS finding expertise is relatively simple * Cross platform testing is easy besides with MacOS (love apple *\*sarcasm*) * Again, A lot of browsers that support dev friendly tools which leads to quick development * Windows systems are relatively simple to learn for the user   **Cons**   * Costs are relatively high like apple because it’s not open source | **Pros**   * Client already versed in Android app development * Relatively low cost from such widely used and flexible operating systems (Android and iOS) * Almost all of the population uses a mobile device so clients can easily handle the system   **Cons**   * Harder to test cross platform capabilities and browsers are not user friendly * Long development time * There is more than just iOS and Android to develop in if they want a wider reach * Updates tend to be less frequent since they must go through Google and Apple play store |
| **Development Tools** | **Programming languages**   * Swift(free and open sourced) * Java * Python   **Programing IDE’s or tools**   * Visual Studio (Free) * PyCharm (Free) * Xcode (99$ per year if you want to deploy if not it’s free) * A MacOS device   **Requirements**   * A dedicated team that can use Swift (MacOS main coding language) and Xcode (The IDE recommended by apple to be used with Swift) efficiently. And if they can’t there are books available through apple to help teach. | **Programming languages**   * Python * Java   **Programing IDE’s or tools**   * Visual Studio (Free) * PyCharm (Free) * Eclipse (Free) * Any Laptop or comp that runs Linux   **Requirements**   * Should just need a single team well versed in python and java to work in which ever IDE they so please to work in as a team. | **Programming languages**   * PowerShell * Python * JavaScript * HTML   **Programing IDE’s or tools**   * GitHub (community support) * Stack Overflow (community support) * Visual Studio (Free) * PyCharm (Free) * Laptop or Computer running Linux   **Requirements**   * Same requirements as Linux just need a team unified under one IDE that can code in PowerShell, python, JavaScript, or HTML to complete the tasks efficiently | **Programming languages**   * Java(Android & iOS) * C++(Android) * Kotlin(Android) * Swift(iOS)   **Programing IDE’s or tools**   * Xcode(for development in iOS 99$ per Year) * Visual Studio Code (for Android or iOS -Free) * Android Studio (for Android, onetime fee of 25$ is needed if you want to publish anything to the Play Store) * A Laptop or computer running their respective IDEs for which ever OS they are developing in   **Requirements**   * Will need at least 2 teams one developing in the Apple Play store and one for the Android Play store as to keep their expertise in the respective areas |

## 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**:

By far the best recommendation for the ability to expand Draw It or Lose It towards other computing environments would be to migrate to a serverless or cloud-based environment running Linux OS. Running a serverless architecture allows for high scalability (or descalability if needed) and encourages effective solutions that require less maintenance. Cloud-based systems allow the client to transfer costs from maintenance and ownership (of the server) to the third-party vendors allowing for a pricing tree based on executions. Cloud-based services can be accessed through HTTP, APIs, etc. which provides for a wide range of other cross-platform devices to connect. Keeping the application on a server allows the programmer to focus on the front end of the application while keeping any responsibility for upkeep to the third-party vendor which allows for more dedication to the application.

1. **Operating Systems Architectures**:

Both serverless architectures and Linux OS are created in a modular fashion. Serverless architectures build off of the middle layer of a three-tier solution that executes logic and forwards data between the user’s UI and a database. This middle layer also provides different services such as content delivery and security services and is akin to a microservice architecture which allows for scalability. This larger scale is why going serverless allows for the gaming room to only pay for what they use (or executions). In Linux OS the Linux kernel is modular meaning its capabilities can be expanded through dynamically loaded kernel modules. This modularization provides better efficiency and processing while occupying less disk space providing a cleaner and more stable UI for users. The primary benefit of these systems being modular is allowing for a single element or application within the system to fail or update without crashing the whole system which leads to higher uptime.

1. **Storage Management**:

Staying in the serverless environment leaves us access to a server farm that leverages its storage management layer as a means to store media and data. This storage management layer creates a resource pool that promotes virtualization and redundancy as a way to save space on their end. Virtualization allows for the nullification of concerns from the programmer about memory capacity within their application. To support virtualization direct access storage should be used with indexed allocation. Indexed allocation allows fragmentation stemming from contiguous allocation to be null which leads to fewer performance issues and higher uptime. The Linux OS storage structure uses multitiered indexed allocation. This allocation stores all pointers on a single indexed block which then points to other blocks containing files or directories. Linux tries to keep blocks near the initial pointer block to reduce seek time and memory effective access time which together provide a better user experience.

1. **Memory Management**:

The Linux operating system uses partial memory execution through virtual memory in demand paging. Partial memory means that the CPU is executing the application using only part of it in the memory; Doing this allows for more space on the memory and can lead to multiple programs running at the same time. Partial memory can also lead to faster loading time and processing speeds which improves the user experience. Linux uses tables for each app process executed these tables are used to track pages that are in memory. Pages or logical segments are located on a disk and are only pulled into physical memory when they are needed for execution. If these pages are not properly managed, they can result in page faults which can inhibit performance. The main way page faults can be inhibited is through a well-organized swap algorithm that just swaps information from the virtual to physical space. Linux uses the LRU algorithm or least recently used algorithm; this algorithm uses relatively old or dormant pages as the first candidate for being swapped. This in turn promotes organized swap algorithms which lead to a better demand paging system which leads to an improved user experience.

1. **Distributed Systems and Networks**:

As The Gaming Room has stated they would like for this game to expand to multiple platforms, to allow for the sudden growth and possible future growth of the game, serverless architecture is very advantageous for its scalability. To accomplish heavier loads the application will need to be run on a distributed system using multiple servers. To be successful in a distributed system, both replication and load balancing are essential. Load balancing is accomplished by sending client requests across multiple servers to lighten the load on one server, doing this can improve availability and the speed of requests. Load balancing can also come with its own security, providing DDoS prevention and more secure firewalls. Replication should be used as a master to slave strategy; this should be implemented by having a single write node and multiple read nodes. We will use this method because reading information is much more frequent than writing, so when data is modified it only needs to be modified once on the write node (master node) but can be pulled from any of the read nodes (slave nodes). Having the distributed system contain load balancing allows for reduced connectivity issues. If a server were to go down or if an instance becomes unavailable all traffic would be directed to other good servers which effectively negates downtime but might come at a slight increase of load to these other servers. This is also the case for the deployment of new code; the developer should be able to create new code and send it to individual servers for testing without affecting other servers which leads to a zero-downtime experience for users.

Communication between clients and the app server will be handled by an HTTP protocol over the internet. Each connection will contain a unique identifier to aid the statelessness constraints of the HTTP protocol but must also ensure performance isn’t sacrificed in its management. While supporting multiple clients a single API that can support each client at once should be used. REST API communicates via HTTP to clients and servers in a stateless manner and can maintain unique identifying data. The client should send messages via GET, PUT, POST, and DELETE methods and should only carry out the functions desired to specific resources. These REST resources are better mapped to functions on which serverless architecture builds its solutions on.

1. **Security**:

The Gaming Room wants to handle authorization and authentication through login credentials using a username and password. The best course of action using login credentials is to use a role-based access control method that practices least privilege. Splitting credentials into players and admins can allow the users only what they are intended to do. This should include the player only being able to create an account, make and name teams, and obviously play the game. Administrators can be broken down into subsections to allow access for only certain files or methods or can just be an overall privilege that can get into anything. Using a REST architecture would require each client side to have identifiers; in this case, the games are relatively small which means identifiers are more than likely going to be completely unique. User data such as passwords and usernames should be stored as a hash to protect user information from being read through plaintext. The Client-side should be protected through browser security which prevents a third party from snooping around while on a browser, and transmission encryption which is basically just sending information through secure channels. These methods should always be kept up to date on the client side. As talked about earlier Linux uses a modular method of process independence. This entails that each process or application runs by itself and doesn’t affect any other application unless explicitly allowed. Linux also uses least privilege for each of its users which only allows them to open files that they are allowed access to. Linux uses DAC or discretionary access control so the ones who own the file (in this case The Gaming Room) can grant access to only those who need it.