

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 |
| --- | --- | --- | --- |
| 1.2 | 02/21/2021 | Joshua Kovacevich | This update finished and updated the recommendations portion with analyzation and recommendations for the operating platform (for the server) the architecture used, Storage and memory management, the network systems and security procedures to protect the server and client from potential security risks. |
| 1.1 | 02/07/2021 | Joshua Kovacevich | This version made updates to the Evaluation in particular weighing the pros and cons of different operating systems and development in all aspects of the game from server to client. |
| 1.0 | 01/24/2021 | Joshua Kovacevich | This version is refactored from the proof of concept, it uses a base Class to standardize the other classes and enables restriction of game, team and player classes names and ids. It ensures effective GameService utilization. |

**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)

The Gaming Room, henceforth referred to as the client, needs a multi-platform web-based Game Service developed for their game “Draw It or Lose It” which is currently an Android exclusive app. This game pits teams against each other to solve a puzzle(think phrase, title, or thing) that will be drawn over a period of 30 seconds from a library of stock drawings. The game has four rounds (each one minute long) where if the initial team fails to find the answer the other teams are granted a 15 second opportunity to submit one answer to solve the puzzle.

**Requirements:**

* Each Game needs at minimum one team.
* Each team can have multiple players.
* Game, team, and player names are unique and allow players to check if name is in use.
* Only one instance of each game can exist at a time.
* The game must be accessible to multiple platforms.
* The game requires time synchronization as it has strict time limits.
* The game is web-based.
* The game uses a prepopulated library of images that must work with all platforms.

## [Design Constraints](#_2et92p0)

While there are design constraints involved with using a web-based design the benefits may outweigh the associated costs. The First Design constraint will be game hosting, if the game is hosted on a server then we will need to ensure all platforms intended can properly interact with that server. On the other hand, if hosting is done by one of the players then the server can be effectively installed on each device, though this would require platform specific server configurations and would likely cause a significant amount of debugging as the interaction between multiple platforms directly may cause issues.

The next design constraint will be adapting the game in a way that will retain already present userbases on Android while encouraging other platform users to play the game. Users from different platforms interact with their devices differently, and an Android first approach may not be successful. A potential solution to this would be tailoring/tweaking the display to each platform to conform to their general preferences and adding cross platform functionality to the existing Android app.

A third design constraint will be the time issues, there is a potential for lag through individual internet connections and exploiting of lag reduction techniques could cause security issues or just reduce the overall quality of gameplay. A solution to this maybe in running using the Game Service itself to provide all timing with small (maximum of a couple second) cushions between times to account for lag. With one centralized timing mechanism there is less room for bugs and exploitation and centralization allows bugs to be isolated, found, and resolve quicker.

A large fourth design constraint is the mobile phone itself. While Screen resolution has come a long way there are only so many pixels you can cram into a 6in screen. In this respect I think it would be beneficial to track “difficult” problems with extremely low success rate and either manual review, automate review, or simply discard any problems/images that could potentially degrade the playing experience. While it may only be a couple outliers those outlying problem images could frustrate players and in extreme cases cause them to even stop using the app entirely. Therefore, I believe it is imperative to consider these ramifications and attack this issue head-on rather than attempting to recover from any damage it may cost.

## [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)

If you look to the UML diagram below you should be able to see an overview of the layout of the application. As the Player, Team, and Game classes all shared some common features and utility it made the most sense to have all those features and design points be centralized and inherited by those classes. This allows for an easier management and standardization of these features reducing the chance of a major game breaking design flaw. This also allows rapid development of new features that would be needed in all the other objects. An example would be a variable for time management or utility feature for retrieving server time. Currently, the Entity class passes on a standard variable id and name to each child class and centralizes the basic creation of these classes, this also enables a way to ensure unique instances of the games, teams, and players with relative ease.

As the lines indicate the GameService while being unique itself, it can handle multiple unique games and ensure no games can have the same name, also allowing for unique and independent id numbers for each instance and their sub instances. The Game class can have multiple teams per instance and likewise the Team class can have multiple players per team. All over these separate instances are contained within a unique game allowing efficient handling of game creation.

The ProgramDriver class contains the main logic of the application and is used to initiate a Game service. From there the driver class ensures single instantiation with the SingletonTester. While this likely will not be in any releases of the GameService the SingletonTester is extremely useful for testing and debugging.

Overall, the current design of this program is meant to utilize the benefits of Objected Oriented Design to the maximum intent possible. This enables quick and reliable scaling to future needs, with relatively centralized and isolatable code. In mass deployment, bugs will come up and this Object-Oriented Design will allow for fast and efficient resolution to the issues.

****

## [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** | **Advantages:**   * Can run Mac code natively. * Uses Mac Hardware * LDAP capable.   **Disadvantages:**   * Few hosting platforms. * Would require extensive costs to host a sizeable website. * Extremely expensive hardware. * Few if any developers host Mac servers for large scale webhosting so finding experienced developers would be a challenge. * Closed source. * Server software is effectively an expensive Linux clone. | **Advantages:**   * Most widely used server technologies. * Open Source. * Most servers run Linux in some form (40%+). * Free Software like Ubuntu Server reduces costs. * LDAP Support * Compatible with MAD * Wide Variety of scripting language support (Perl, PHP, Python, Ruby) * MySQL * Most Developed server technologies * Few security problems   **Disadvantages:**   * Can become complex for average administrator. * Potential software/hardware limitations. * No set support/long term support * Limited software support | **Advantages:**   * Supported by Microsoft. * Beginner friendly IE (GUI) * Supports most Third-party Applications. * System Recovery * Driver support for hardware * Easy Updates   **Disadvantages:**   * Licensing Costs and per user costs. * Most attacked server OS * Significant and usable security exploits. * Limited Scripting language support * Web servers less common/prevalent. * Closed source. * Easy for inexperienced user to make mistakes. * Not a lightweight operating system. * Malware support. | **Advantages:**   * Cheapest option. * Puts resource demand on users. * Android/Apple Support LDAP * Almost no server space requirements for company.   **Disadvantages:**   * Kills User Battery * Server code is in user possession. * Exploitable by malicious users. * Server would need to be made for each platform and distribution. * Likely requires peer to peer direct link which exposes users to security threats. * Cross platform would be extremely difficult to build and maintain. * Limited direct control of servers, bad apples/un-updated services could harm company and following. |
| **Client Side** | * Mac specific software requires Swift/Objective C development * Mac is selective in what they (opening) allow users to user. * Mac has its own browser (Safari) that a fair number of users use. * Mac = Money, Apple charges licensing fees for inclusion and usage in the Apple store. * Exceptional user interface options, this is a double edges sword as users expect very polished designs. * Language Support limits the developers available to build and maintain software on Mac. * Mac can remove your application from Apple stores at any time for almost any reason they choose. | * Linux is completely open source and has few if any real licensing costs. * Linux has one of the smallest Desktop user bases. * Linux development is double edged, you will likely never get a return on investment for Linux specific implementations. * Less Linux specific developers than Mac, Windows, or most mobile platforms. * Not user friendly in terms of GUI, its more of an afterthought. * Limited Hardware support for Linux. * Clients should be relatively cheap; user base even may build it for you. | * Largest User base by far, offers a wide potential audience. * Microsoft support is a huge plus. * Microsoft allows users to download almost anything they want. * Most developers have/can develop for windows. * Most likely operating system to be attacked, someone will exploit any client you make for windows. * If the client costs money, it will be pirated. * Wide browser support means a website will likely work on any Windows PC, but you will need to consider someone is using edge… Somewhere. | * Mobile Based software is platform specific so you will need iOS developers, Android developers and maybe even Windows mobile developers. * All Platforms have browser support, which makes a browser based/hybrid application possible, and likely preferable. * Getting on App stores can be a process (particularly for apple). * Each platform has unique licensing and distribution costs associated with it. * Any action that requires a permission will need to be approved by owner, that means the more resources you need, the harder getting your application to work will be. |
| **Development Tools** | * While Mac has language support for most languages, Swift and Objective C are its main languages that are shipped with it. * Mac has access to some platform specific IDEs, and other IDEs like Eclipse, Visual Studio and Visual Studio Code. Visual Studio Code and Eclipse have Swift support and are completely cross platform. * There are costs associated with other IDEs and it is likely not worth the cost for any benefits given for those IDEs. * For graphics development Godot is likely the best open-source choice with an effectively free use license. | * Linux will support most any language with relative ease. * VS Code and Eclipse are available for Linux. * Unity and Godot can be used to build graphics/GUIs on Linux. * If its on Linux its likely freeware or opensource, if not you will be made fully aware. * Unity for example has licensing costs associated at certain thresholds of income but offers exceptional support and utility. * Linux development will likely require extra time as resources are less readily available for overcoming issues. | * If you want to build an application windows will support, it. You can write it in almost anything and there is likely a license free route for the IDE. * Visual Studio and Visual Studio Code are by far the best options for development on windows. * Unity, OpenGL, Unreal Engine 3, etc.… There are several great utilities for windows GUI development. * License Costs for GUI development are like all other platforms. Possibly more free use Engines than all others combined. * Almost complete language and VM support for all development needs. | * Apple and Google both offer platform Specific development tools. * There are numerous licensing fees that are difficult to get around for mobile development. If you want it in an App store, you are extremely likely to have to pay. * Due to the unique natures of any mobile OS, specific design tools are almost required for smooth development. * Likely will need a dedicated team for each mobile platform. * If one team works entire project, expect significant slowdowns on any native application development as each platform will require spool up and familiarization time. * Developing a hybrid app focused on web technologies is likely the best route. Using the phones browser in a sense to handle most client specific needs is more efficient then developing multiple unique native applications. |

## 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**: I would recommend running a Linux server and using the client’s browser for most client-side applications. This is by far the most cost-effective option and the easiest cross platform client-side option there is. Think back to the days of flash games, It did not necessarily matter what operating system you were on, if your computer/browser could run flash you could play.
2. **Operating Systems Architectures**: I recommend using a Client-Server architecture where much of the information for the application is held on the server side and the individual users hold specific client-side information (token IDs, Cookies, Game information, etc.) for their session. Making the user login, (preferably through a third party like Facebook) will allow the server to only handle background application operations, limit exposure of the server, and allow the server to focus on hosting games and executing processes for those games. The picture database can be held on the server then once an operation (server side) determines the next picture, that picture can be sent to the client, synced to the server(for time) then the client’s individual applications can manage drawing the image and sending/receiving user input through the server, allowing the server to filter and validate data. I also recommend using platform specific payment systems as developing a payment system on your own could be expensive and time consuming and will have less support and protection then other payment services. An Example in the case of “Login with Facebook” would be using their payment system for Desktop applications, and Android Pay and Apple Pay for mobile applications. These will take fees but those will still cost you less than a data breach.
3. **Storage Management**: I would recommend using ubuntu as a server on Linux as it contains built in storage management software. You could (and likely should) use RAID (Redundant Array of Inexpensive Disks) to combine storage space so that any individual storage disk failure will not down the entire system. Storage also involves the Physical storage disks so I will include here that I strongly recommend renting multiple dedicated servers (if you do not already own dedicated servers) across the regions you will be offering the game to ensure decentralized traffic, relatively unthrottled internet connections, and a fixed price regardless of user base. This will allow low latency, decentralized storage, redundancy, and firm costs rather than using Virtual Server hosting which may become extremely expensive. The average price to rent an Ubuntu server, the base price per year is $225 per physical server with additional support and package options peaking around $1,500 per year per server with 24/7 phone and ticket support through Ubuntu. Depending on the size of your user base you can scale up relatively quickly if we build a traffic management program on your initial server by creating a whitelist of other servers to forward traffic to.
4. **Memory Management**: Linux uses a relatively robust memory management system. First, the Linux kernel uses a “Virtual Memory Primer” to abstract the actual physical memory from the application software, this can be used to limit and control sharing of data. For best performance there should likely be a server-side process for n number of games running, while these games are running client requests would be able to send a request or information, the server could plug this information into one of these running processes/threads, return the requested information or updated value, then free the server process (while maintaining it running) for the next client request. In this way the requirement to spool up a process and cold start each time there is a request is almost entirely removed. Processes can be spooled up and prepared in anticipation of high traffic (possibly even running on outside servers for extremely high traffic times) and spooled down or transferred to other servers for maintenance as required. The major benefit of this is that there is little overhead once a process is started, each running process can be calculated for its memory space needed therefore it can be optimized for needs, server requests will be exceptionally fast, and finally, this will allow traffic to be transferred for server maintenance and or repair at a moment’s notice without incurring heavy slowdowns or time-outs. Memory management on the client side will be relatively easy as the client can effectively hold a handful of objects in memory while in use so their OS should be able to indefinitely handle operation. In the worst-case scenario Linux has something called “OOM killer” which if memory resources are exhausted will kill a task in hopes of freeing up enough memory to continue living. This means, in the example listed above, even if a pre-spooled task becomes inflated or has a memory leak of some kind(which we should still avoid) the Linux OS is capable of killing that process entirely and freeing up memory, unless in the midst of an operation when killed it is unlikely any user would notice, and with a proper design that user request could be restarted in another thread with limited slowdown for the user. An automated system could even be implemented to kill, or spool tasks as needed to ensure proper and efficient memory management.
5. **Distributed Systems and Networks**: As touched on above having multiple dedicated servers running the server-side application, likely with several servers coordinating traffic to all servers for each session(cached on client side for session) and each server running pre-spooled tasks awaiting user requests the likelihood of outages and other server problems should be mitigated to the smallest amount reasonably possible. Distributing the client-code on usage like a webpage does makes the most sense as a java applet or JavaScript program can be sent and ran in an exceedingly small amount of time. Furthermore, if that applet is cached on the client side with a tag such as “Version 1.01.34” then that tag can be requested by the server, compared to the most current version and if not updated can be updated on connection with the server. This limits the exposure time where apps are used but not updated for long periods of time, making it necessary for multiple versions to be supported and merging their data. This could even be used to guarantee mobile applications are kept up to date without having a pending install for the user. Finally, this means that the platform specific code becomes less prevalent as building a hybrid app (or even a full web app) is significantly less intensive as it generally uses an embedded browser within the app or the phone.
6. **Security**: While Linux servers generally have a good track record in security, I think it is better to limit exposure for the company. Providing Username/Password functionality is a must but if Single Login is used (through Facebook for example) and payments are routed through various ecommerce platforms (App stores, FB pay, PayPal) for client security and server liability. In fact, mobile applications on iOS and Android are required to use their payment methods and not use any other third-party payment processors). The idea is that you should want to have to protect as little data as possible, Username, password, email, maybe some basic location and advertising data but that is it. There is no reason you should be holding massive amounts of client data in either the server-side or the client-side application. That alone will reduce the likelihood you will be attacked as you have little payoff. Next using end-to-end encryption, hashed and salted passwords, and decentralized storage and authentication enables you to reduce your vulnerabilities. You should also not trust and validate all information sent from a client to the server, this means session tokens, API keys, another key that records states of variables, etc. This along with giving the user no direct privileges to read and write on the server will make attacks exceptionally hard, and relatively un-worth a cyber-criminal’s time. Additionally, running antivirus software, closing all unused ports, and analyzing traffic should be sufficient to reduce security risks to The Gaming Room.