

Draw-it or Lose-It

# **CS 230 Project Software Design**

Version 3.0

## Table of Contents

[**CS 230 Project Software Design** 1](#_Toc190969300)

[Table of Contents 2](#_Toc190969301)

[Document Revision History 2](#_Toc190969302)

[Executive Summary 3](#_Toc190969303)

[Requirements 3](#_Toc190969304)

[Design Constraints 3](#_Toc190969305)

[System Architecture View 4](#_Toc190969306)

[Domain Model 4](#_Toc190969307)

[Evaluation 6](#_Toc190969308)

[Recommendations 8](#_Toc190969309)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 21-Jan-2025 | Dan Loranger | Initial Revision |
| 2.0 | 06-Feb-2025 | Dan Loranger | Completed evaluation section  Completed recommendations sections |
| 2.1 | 07-Feb-2025 | Dan Loranger | Added missing evaluations for the client-side development tools |
| 3.0 | 20-Feb-2025 | Dan Loranger | Revised to enhance and fulfill check items identified on the deliverables list. |

## [Executive Summary](#_sbfa50wo7nsh)

Creative Technology Solutions (“CTS”) has been engaged by “The Game Room”, a new client, to develop a web-based game as highlighted in the following brief.

*Draw It or Lose It is loosely similar to the 1980s television game Win, Lose or Draw, where teams compete to guess what is being drawn. Rather than a player drawing images on an easel to help team members guess the puzzle (a phrase, title, or thing), the application will render images from a large library of stock drawings as clues. A game consists of four rounds of play lasting one minute each. Drawings are rendered at a steady rate and are fully complete at the 30-second mark. If the team does not guess the puzzle before time expires, the remaining teams have an opportunity to offer one guess each to solve the puzzle with a 15-second time limit.*

## Requirements

* Game play consists of 4 rounds
* Each round lasts 30 seconds
* Opposing team opportunity lasts 15 seconds after each round
* Images render linearly across the 30-second round interval (presumed as scan lines)
* 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](#_2et92p0)

- UI will be web-based

- UI must support modern browsers, typical PC and mobile platforms. This may require separate UI per platform.

- Game server must be centralized to prevent multiple instances concurrently and allow teams / games to form dynamically.

- Given the design already exists in an Android app, existing code should be reviewed and leveraged as much as possible for re-use. Given that the existing Android APP utilizes JAVA, this will be the programming language used on the Android APP as well as the server-side code.

## [System Architecture View](#_ilbxbyevv6b6)

Not Required at this time.

<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)

Provided here is the existing game UML diagram as previously generated by the customer.

**"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.**

Behaviorally, the entry point is the *ProgamDriver* which implements the *main()*. This main uses the *SingletonTester* class to ensure only a single instance is active on the server.

The *Entity* class is the parent class to the classes of *Game*/*Team*/*Player* with each child class inheriting the parents attributes, but also implementing a specific *toString()* function respectively.

The *GameService* class is responsible for adding games via *addGame(String Name),* and provides and interface for locating existing games by ID and Name, along with identifying/assigning player and Team numerical IDs via *getNext…Id().*

Once a game is added, the *Game* class manages a list of existing teams, and allows adding teams by name *addTeam(string name).* Once a team is formed, the *Team* class manages a list of players, with the ability to *addplayers(String name).* The *Player* class is a single object with a public instantiator that requires an ID (long type) and name (string type).

Architectural Question – It is observed that in the Team Class, it might be useful to include the option to remove a player (disconnected or similar). This will need reviewed with the customer.

Architectural Question – It is observed that in the *Game* Class, it might be useful to include the option to remove a team (disconnected or similar). This will need reviewed with the customer.

Architectural Question – It is stated the game will use a predefined library of images, but it is unclear from the architecture if the game will be rendered locally or on the server. This decision will directly affect the software architecture and also the performance of the game play. A local rendering on the user’s device will require either higher data streaming rates (higher ongoing server bandwidth costs) for transferring images or alternatively a significantly larger installation package on the user’s device. This will need reviewed with the customer.

* If operating locally on the user’s device, either multiple resolutions of images will be required to be downloaded, or resizing prior to display will be required for proper screen fit.
* If rendered on the server, this resizing could be either 1 time with stored images (higher storage capacity), or resized dynamically during game play (higher CPU performance costs for repeated redundant operations)

## [Evaluation](#_2o15spng8stw)

For the server side, there are a few options for operating system, with Linux and Windows being the widely adopted options. Mac OS and Mobile devices do not play well in the server environment and will require the customer to purchase hardware, create the network environment and host their own servers as these will not be available under traditional hosting farms.

From the available options, the top contenders in the SERVER operating systems are LINUX and WINDOWS based servers, with LINUX being the recommended option.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | While it is possible to create and host a server on the MAC environment, this is not a mainstream option and may require custom hardware, networking and maintenance to support.  This option is strongly advised against! | Linux is a widely deployed option within the webhosting community. It will be widely supported including security patches, OS updates, and hosting backend updates on a regular cadence as vulnerabilities are discovered.  The operating system is typically free of charge unless a paid maintenance program such as RedHat is selected. | Windows servers tend to be more comfortable for less technical users as the tools are built around the familiar windows GUI.  Licensing is a considerable concern as Windows is not free and license fees can be deployment, user, and other criteria specific and recurring.  This option needs to be carefully evaluated and understood as the current 2025 license cost is ~$6700 as an entry cost plus recurring hosting fees. | Mobile devices may be capable for very small deployments and user bases, but are exceptionally poorly suited for anything that would require scaling and typically would not meet the redundant hardwired NIC for a properly configured web server (increased reliability).  This option is strongly advised against! |

Evaluation Continued…

Client side - For the client side, there are multiple options for operating system as well as the development environment (language) options. Mobile devices currently are commanding the majority share of the market, with Android being the top with almost 50% of the total market, followed by IOS with approximately 18% of the market share, comprising about 65% in total user base.

It would be recommended to select to a variant of Xamarin (now evolved to .NET MAUI) which will allow the code the maximum reuse first in the mobile devices markets, and also allowing traditional desktop platforms to follow based on business needs and interest after the mobile devices market has been captured.

If decided to continue using a Web UI as the primary UI, it is important to keep FLASH technology as forbidden as it is not universally supported. Further, due to the limited resources and uncertain connectivity conditions, all streaming data must be optimized for compactness to ensure minimum possible latency. Where possible, account for UDP traffic that will allow for missing packets of data while still allowing the functionality to continue (a few missing pixels shouldn’t render the image unusable, or if excessive would alert the user to poor connectivity situations).

| **Client Side** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Development Tools** | For the MAC, the dominant programming languages are Objective-C and Swift.  Per <https://gs.statcounter.com/os-market-share>, Jan 2025 Data, the OSX operating system has a holding steady market share of about 5.3-5.4%. This represents a small overall business case.  With well written code, it is possible to support MAC with common code with other OS using modern programming tools such as visual studio with Xamarin. | For the Linux OS, the dominant programming languages are C/C++, JAVA and Python. Swift may be possible, but has not been well adopted currently.  Per <https://gs.statcounter.com/os-market-share>, Jan 2025 Data, the LINUX operating system has a holding steady market share of about 1.3%. This represents a small overall business case.  With well written code, it is possible to support LINUX with common code with other OS using modern programming tools such as visual studio with Xamarin. | For the Windows OS, the C/C++/C#/JAVA/VB and many other languages are well supported. Windows has long held a strong lead in the market place, and virtually all programming languages will have good support. Visual Studio and Eclipse being the most popular graphical IDEs.  Per <https://gs.statcounter.com/os-market-share>, Jan 2025 Data, the Windows operating system has a decreasing market share of about 25.4%. This represents a Reasonably strong business case. | Per <https://gs.statcounter.com/os-market-share>, Jan 2025 Data, The mobile market is clearly the future with android dominant market share of 46.1% and climbing, and IOS holding a steady 18.1-18.4%.  Android has traditionally been very strongly JAVA with Kotlin becoming a well-supported language.  Additionally, Python and C# and C++ are compatible.  The IOS platform follows the MAC programming model. |

## 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**: from the SERVER-CLIENT model, the server and client can operate on different OS. This is advantageous as it allows the server to be a centralized server that utilizes commonly available server operating systems while also allowing the user base to use a mixed environment of client operating systems to access the game play front-end. With examples of modern online RPG systems, even multiple servers could be implemented allowing load sharing should the need arise.
   1. **Server** - It is strongly recommended to select a LINUX based operating environment for the server. Using a LINUX operating platform will minimize operational costs by eliminating licensing and usage fees associated with other operating system choices, while maximizing the deployability between server farms globally. Using a widely deployable operating system allows for utilizing competitive bidding to further reduce operating costs between suppliers.
   2. Client - Currently there is a strong demand for portable gaming stations including mobile phones, tables, and handheld gaming platforms such as the Nintendo switch. Given the customer is looking for the gaming market, and not the business category, this warrants special consideration when selecting the targeted client environment.

In the primary market areas, it can be assumed that nearly everyone owns and carries a modern smart phone every day and also utilizes either Android or IOS operating systems which account for approximately 65% of all client-side devices, with Windows accounting for an additional 25% in the form of traditional computers.

* + 1. Android is currently the highest contender by market share and should be the primary client target.
    2. IOS is also a strong contender in the mobile market which should not be excluded.

Given there is no application licensing fees for either platform, and the client hardware is provided by the end user, there is no per-user fees in the client side to need considerations.

Overall, the C++ language offers the maximum reusable code given tools such as Visual Studio with Xamarin tools which allow compiling for multiple platforms are utilized for the client devices. Knowledgeable cross platform developers will be needed to ensure applicable safety and procedural standards are implemented that are supported by all targeted operating systems.

1. **Operating Systems Architectures**: Using the Server Client model makes the most sense for this given application.
   1. **Server** - All game resources should be implemented on the server where each user will log-in and manage their games via remote devices. This will ensure consistent game-play for all players and allow better compatibility between all clients long-term and will scale naturally to higher player counts. Game play image storage and rendering can be done via optimized functions that stream the graphics in a minimal acceptable resolution to the client device based on the client device settings. Streaming the images in the optimized resolution will help to manage user download latency under typically network latency conditions, while allowing the images to be centrally refreshed to keep the game from getting repetitive upon frequent game play. The game mechanics do not require real-time reactions so higher network latency can be handled in the local client if required.
   2. Client – For the client device, the minimum footprint possible should be implemented to keep the application small and highly responsive on the largest number of compatible hardware devices. The network will be a highly variable performance indicator that must be implemented with sufficient buffering and fault handling capabilities to handle a variety of network faults such as sporadic connections, network device source switching (wifi vs cellular, wifi vs lan, etc), very slow downloads, etc. Upon image API requests to the server, the user’s device resolution should be provided to select the optimal sized image to prevent unnecessary over sized images from being downloaded (bonus reduce server bandwidth consumption) which will also eliminate unnecessary device computations for image transformations.

From the user interface, the UI should support adaptive scaling of the font for users with limited eye-sight, as well as providing a selection of themes such as daylight and dark modes to allow best user experience in variable lighting conditions.

For maximal flexibility, the programming language of the server and the clients is recommended to be C++ language that can easily be compiled and distributed to multiple devices from a common project repository. Refer to the evaluation section for more detailed analysis of the trade-offs between both server and client operating system architectures.

1. **Storage Management**:
   1. **Server** - Presuming the customer chooses to use a pre-established web hosting service, storage management will be handled by the hosting vendor thus relieving the customer of needing to evaluate, procure, and manage storage media.
   2. **Client** - The local client device will use native available non-volatile memory for the application code, and volatile memory for the game play renderings. Closing of the app on the client device will clear the game renderings which are held in volatile RAM memory, but can be recovered from the server with applicable recovery functions in the API. Due to the variability of the end devices in terms of speed and storage capacity, the use of RAM should be optimized for general performance with minimal footprint. Buffering of the entire game play images is recommended, with images being immediately flushed upon completion of each game round to free up the memory for other functions.
2. **Memory Management**:

*https://tldp.org/LDP/tlk/mm/memory.html*

**Server Side** - Linux uses a caching architecture as the first layers of memory, followed by disk media-based memory. When cache misses occur, new pages are loaded into the cache, and the least recently used data is written back to a paging file on the hard media which can be recalled again later or written back to disk as applicable. As long as the application is written correctly where memory leaks are not created, no advancement memory management is required.

To ensure optimal performance under very heavy loads, the server system needs to start with a maximal amount of RAM memory to reduce page swapping that could compromise performance during these heavy demand periods. Additionally, the server software architecture should account for the need to repeatedly transfer the images to multiple clients in a non-synchronized fashion which will require non-consecutive memory addressing frequently. The type of RAM chosen should be optimized for random address seek times.

In addition to the system RAM, the non-volatile memory (disk drive) should be designed to ensure data redundancy in both local drives as well as remote storage locations. The optimal configuration for all game data files will be handled as database files with applicable traceability logs to ensure complete ability to synchronize and reproduce the data set after each transaction.

The database architecture should be chosen again to prioritize random item selections. User details should be contained in encrypted formatting, ideally using an AES-256 type encryption to help protect against data theft in the event of a security breach. Where ever possible, these files should only be exclusively accessible by a local virtual user that cannot be logged in via a remote connection to further reduce exposure by preventing any read/write/copy by any other users. Under no circumstances should the database return the user details, but rather only accept a defined set of details that are either written to the database, or checked against existing entries with a valid/invalid result being returned.

**Client-Side** - At the client device, consideration needs to be made to ensure minimal memory is consumed and allocated memory is reused wisely. This will maximize the performance of the device overall as typically mobile applications will have many APPS paused in the background with a main app in the foreground that each need to utilize some percentage of available volatile RAM. Rarely accessed or rarely changed data should be written to disk and utilized in a READ-ONLY model whenever possible to allow reclaiming the RAM efficiently in real-time.

Memory should be prepared in advance to hold the images to be displayed. This may be in the form of either full resolution images, or scaled down images depending on the source quality. It is recommended to have storage allocated sufficient to hold the entire game-play workflow in memory, where each image as consumed is refreshed in the background with the images that would be required for the next game. This allows a more seamless gameplay even when in non-optimal network environments where image downloading could take some additional time to complete. Special care should be taken to ensure the memory is release as soon as it is not needed, and only the required new memory is taken to minimize the application footprint and to ensure smooth operation alongside other running applications.

1. **Distributed Systems and Networks**: With a distributed system, the networks and servers play a critical role, and must guarantee a maximum up-time through redundancy at all levels including internet connectivity, server hardware redundancy (preferably in separate locations), and 100% fault monitoring for rapid fault response at every instance.

With centralized servers, and distributed clients, the network bandwidth and reliability must be a primary consideration when selecting service providers, and once the available bandwidth is understood, modeling must be done to understand what the maximum client capacity is that the server infrastructure can handle. Once this capacity is understood, other design details can be tailored to suite the capacity. Such items could be the format the images are transferred in, the encryption and logging methods, their packet frequency, etc. can all be optimized to fit within the overall capacity available while also servicing the maximum number of clients concurrently.

It is required to keep the overall server resource consumptions below 90% to ensure background tasks not accounted for such as Denial of Service monitoring, virus scanners, back-up utilities, and other such tools have sufficient processor capacity to complete their work at a reduced priority without interrupting game-play.

In the server, maintenance breaks will take down the entire community’s ability to play, which if unplanned and extended in nature can really hurt the reputation of the company and the game. To help ensure minimal impacts, preventative maintenance must be scheduled in advance, and occur during the quietest user load periods (after analysis).

A client-side maintenance /upgrade/reboot should be a last resort as well, but should they be required, the user must be allowed to choose their convenience time with perhaps restricting access to the server until the required updates/etc. are completed. This client device is an isolated user, and will not affect the community at large, so it can occur any time the client device gains approvals.

1. **Security**: Security is a hard requirement for the customer. Given this requirement, the server should implement the strictest admin protocols possible, and the game server code must use continuous authentication when executing API calls to ensure the right data is handled to the right client device. This not only prevents mishandling game data, but also protects both the server and the client from bad actors. In addition to continuous authentication, all customer identifiable data must be handled using encryption techniques both in transit and in non-volatile storage.

Additionally, under the current game model, it is possible to create the majority of the file system as a read-only architecture, with only player user details and logging requiring actual file writing to disk which can be tightly protected using separated and independently encrypted databases for each user detail such as user names, passwords, payment methods, etc.

In the event of someone attempting to corrupt or otherwise mishandle the server code, it will not be possible due to the read-only nature of the file system. Attempts to steal customer data will be much more difficult as multiple encrypted files will need to be stolen and then decrypted to gain any useful user identifiable data.

On the local client device, the user must log-in locally and authenticate with the server before game mechanics will become available. In these client devices, the user credentials and any server tokens will be stored only in volatile memory that will be reclaimed and reused each time the client application is closed, leaving no record locally that can be stolen and misused. Data integrity checking should also be performed routinely to look for tampered data on the customer device as this is the most likely point of vulnerability.