

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.0

## Table of Contents

[**CS 230 Project Software Design Template** 1](#_Toc115077317)

[**Table of Contents 2**](#_Toc115077318)

[**Document Revision History 2**](#_Toc115077319)

[**Executive Summary 3**](#_Toc115077320)

[**Requirements 3**](#_Toc115077321)

[**Design Constraints 3**](#_Toc115077322)

[**System Architecture View 3**](#_Toc115077323)

[**Domain Model 3**](#_Toc115077324)

[**Evaluation 4**](#_Toc115077325)

[**Recommendations 5**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 05/23/24 | Joe Massa | Initial design outlining the clients major requirements |
| 1.2 | 6/8/24 | Joe Massa | Evaluation complete of various OS platforms |
| 1.3 | 6/19/24 | Joe Massa | Recommendations and Final Design Report |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room would like to take their popular Android App, Draw It or Lose It, and develop it into a web-based game that will serve multiple platforms. The game is similar to the popular 80’s television game show Win Lose or Draw. Teams will compete against each other trying to guess a puzzle, phrase, title or thing which is depicted in a drawing that will slowly render on screen over 30 seconds. If after 30 seconds the team cannot correctly guess the puzzle, the other teams will get one guess each. This document will outline the major requirements the client has as well as other technical constraints that will influence our design.

## Requirements

The following requirements were given to us by the client (The Gaming Room):

* Web Based – the game must be available to all internet users.
* The game will consist of one or more teams.
* Each team will consist of multiple players.
* Game name and Team names must be unique.
* Users will be able to check if a name already exists.
* Only one instance of the game service can exist in memory at a single time.

## [Design Constraints](#_2et92p0)

Based on the client requirements above, these are the initial design constraints. Note that these are subject to change as the development progresses.

* Browser compatibility – Since this is to be a web-based game, we need to ensure that our code will support a wide range of popular browsers including Chrome, Firefox, Safari, Edge, Internet Explorer and mobile browsers.
* Game Name – ensure that each game has a name. Each name must be unique. User iterator to check game name for a match.
* Multiple teams – allow for multiple Team objects to exist. Each object must have a unique name. Use Iterator to cycle through existing names looking for a match.
* Multiple Players per team – ensure that multiple player objects can be part of the same Team object. Players must also have a unique name. User iterator to look for a match of existing name.

## [System Architecture View](#_ilbxbyevv6b6)

The Operating System Architecture recommended for this game application is a Client-Server Model. The Server system should be a powerful computer with a stable and secure operating system that will handle all game logic, store game data, manage game users and handle communication and data transfer between the server and the clients. Specific details about the Server can be found in the Recommendations section later in this document.

The Clients for this game can be desktop/laptop workstations running Windows, Linux or Mac Operating systems. The client application can run on mobile devices running Android or IOS operating systems. The client application can also run on popular web browsers such as Google Chrome, Firefox, Safari or Microsoft Edge.

The Client-Server model offers wide flexibility for clients and scalability for future expansion. The Server can also expand capacity as the number of concurrent users accessing the game fluctuates.

## [Domain Model](#_8h2ehzxfam4o)

In the domain model we see that the Game, Team and Player classes are child classes of the Entity class indicating that they each inherit the entity’s ID and name variables. The Game class consists of a team’s array, while the Team class consists of a player’s array. The player class does not contain any private variables and simply adds players to the team array from the Team class.

The game Service and Entity both feature Singleton patterns keeping their constructors private to ensure that only one instance of the service and entity exists in memory at a single time.

The Program driver uses the SingletonTester class to ensure that the singleton pattern is functioning.

**"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)

The following matrix outlines key aspects and considerations for developing and distributing cross platform applications on the popular platforms available today. This information is the basis of our recommendations which come later in the document. It should be noted that this by no means is a complete list of features or available tools, but rather the major points of consideration. Given the changing nature of the computer landscape this information is current as of June 2024 and should be updated as technologies / features change could impact our recommendations.

| ***Development Requirements*** | ***Mac*** | ***Linux*** | ***Windows*** | ***Mobile Devices*** |
| --- | --- | --- | --- | --- |
| ***Server Side*** | *\*AS of 2022 Apple has discontinued their Server Applications and Hardware making this a nonviable option without existing hardware/software*  **Pros**  -Unix based made this a stable platform for development  - Popular development environment with familiar interfaces.  -supported by high performance stable hardware  -Secure platform  **Cons**  -Discontinued in 2022 and no longer updated software or hardware  -limited 3rd party support  -limited scalability for larger scale application  - very high implementation costs. | **Pros**  -Open-Source code reduces licensing costs making scaling more affordable.  -Wide support for almost any application need  -Scalability – Easily handles large scale application  -Large community of developers for support and training.  **Cons**  -Steep learning curve for those unfamiliar with the platform  -Heavy reliability on command line tasks-difficult for beginners  -Strong understanding of Security best practices needed to secure applications (due to open-source nature of Linux)  -Niche hardware can have driver compatibility issues. | **Pros**  -Familiarity with windows development environment  -Powerful and mature ASP.NET framework offers tight integration with windows clients.  -broad compatibility with development tools  -wide hardware compatibility  **Cons**  -Cost due to licensing fees makes scalability costly  -more vulnerable to malware attacks than Linux based systems.  -complex scalability possibly requiring additional licensing fees for additional features.  -limited open-source applications | \*Mobile devices are not suitable for web hosting due in part to the following:  -Limited processing and memory  -Security risks due to a lack of server security applications  -battery dependance  -network reliability (lack of hard-wired connectivity)  -instability issues as mobile devices are meant to regularly go to “sleep” mode and power down. |
| ***Client Side*** | **Costs:**  -Xcode is free but additional tools and libraries may incur costs  -broader skills required for developers.  **Time:**  -Building functionality across platforms increases when compared to a single client.  -testing and maintenance times are also increased  **Expertise:**  -Cross-platform development skills  -UI/UX design expertise  -specific knowledge for testing | **Costs:**  -Open-Source development keeps costs low  -Large developer community keeps development costs lower  **Time:**  -Similar to Mac, building functionality across platforms takes additional time  -testing across platforms can be a significant time investment  -Maintenance is an ongoing commitment  **Expertise:**  -Cross platform development skills needed  -UI design expertise is important  -testing experience across platforms is necessary | **Costs:**  -development tools are licensed adding to development costs.  -developing cross platform compatibility can require more experienced developers.  -testing software incurs additional costs.  **Time:**  -development for cross platform increased development time  -testing and maintenance also increase development time  **Expertise:**  -expertise in specific .NET technologies is needed  -cross platform frameworks reduce the expertise needed for cross platform development  -UI/UX design experience is needed. | **Costs:**  -many development tools for Android and IOS are free  -specific libraries and sdk’s may incur costs  -app store fees for publishing applications  -developing compatibility for different devices and screens can incur costs.  **Time:**  -developing native apps for both android and ios platforms significantly increases time.  -increased learning curve for mobile application development vs. desktop and web applications.  -testing on mobile platforms is time consuming.  **Expertise:**  -platform specific development knowledge required  -strong UX/UI design experience needed.  -device specific knowledge such as camera, microphone, gps, touchscreen |
| ***Development Tools*** | **Languages:**  -Swift  -ObjectiveC (for legacy applications)  **Tools**  -XCODE (IDE)  -Visual Studio  \*Many IDE’s offer cross platform support  -Platform Specific SDK’s | **Languages:**  -C, C++  -Java  -Python  -Javascript  **Tools:**  -IDE(Most popular IDE’s such as Pycharm, NetBrains, Visual Studio and Eclipse offer Linux versions.  -Platform Specific Libraries and SDKs | **Languages:**  -C#, C, C++  - Java  -Python  -Javascript  **Tools:**  -Visual Studio  -IDE(Most popular IDE’s such as Pycharm, NetBrains, Visual Studio and Eclipse offer Windows versions.  -Platform Specific Libraries and SDKs | **Languages:**  -Android (Java, Kotlin, C++)  -IOS (Swift, ObjectiveC)  **Tools:**  -Android (Android Studio)  -IOS (Xcode)  -Platform Specific Libraries and SDKs |

## Recommendations

The following contains our final recommendations for Operating Platform, Architecture, Management of Storage and Memory, Networks and Security. Current requirements, future expansion, reliability, cost and management have all been considered when making these recommendations.

1. **Operating Platform**: For our application server, we recommend using a Linux based server. Linux is an excellent choice for hosting web applications of all sizes. Its stability, security, flexibility, and vast software ecosystem make it a powerful platform for developers. While the initial learning curve can be a challenge for some developers, we feel that the scalability, security of the platform, and the lower cost of future expansion make this the best recommendation.
2. **Operating Systems Architectures**: As mentioned earlier, we recommend the Client-Server architecture offers the broadest compatibility for clients and client access while providing a robust and stable platform for game data. With this model and a Linux server system, we can easily scale our game to accommodate a large range of user loads.
3. **Storage Management**: Storage requirements for our game data and application on the server requires about 2GB of space. This is not a problem since our server would have a minimum of 2TB of high-speed SSD storage. Our image database can easily increase exponentially with minimum investment in additional storage. On the client side, clients will have the option to download the full image library, a compressed image library, or stream the images over the internet with only the minimum amount of storage on their client machines. Our images should render easily over current internet speeds especially since the game play requires them to fully render over 30 seconds. Most images will render fully in about 2 seconds without problem. The operating system of the client will handle how these files are stored on the local hard drive with only standard Hard Disk Management required on the part of the user (i.e. defragmentation, etc).
4. **Memory Management**: Memory management is more of a concern than storage management for our game. Memory capacity on the Server should not be an issue as most servers will have a minimum of 64GB RAM. However, the clients, specifically mobile devices, are more of a concern. For desktop clients we are recommending a minimum of 4GB of RAM. Since most desktop users will be running other applications a capacity of 8GB is advised. To optimize the smaller memory capacities on mobile devices, a combination of image optimization (compression), responsive design to dynamically serve various compressed images based on client system restrictions, and leverage browser caching systems to store frequently accessed images such as game graphics and textures.
5. **Distributed Systems and Networks**: To enable cross-platform communication we can take advantage of distributed systems and networks come. The client, running on various devices (PC, mobile etc.), will communicate with the game server through a network protocol. This server manages the game world, player interactions, and updates clients. Optionally, a matchmaking server can pair players based on preferences. A database will store persistent game data. The entire system depends on reliable internet connections for all users, as outages or slow connections can disrupt gameplay. Latency, the time it takes for data to travel, is also a factor, and geographically distributed servers can help minimize lag for distant players. Clients and servers communicate using a common network protocol for compatibility. Security measures like encryption are crucial to protect player data and prevent cheating.
6. **Security**: On the client side, user data should be minimized and stored securely using encryption and platform-specific solutions like keychains or Secure Enclave. Regular updates are crucial to patch vulnerabilities. We can leverage built-in security features offered by mobile platforms (like Apple's App Store guidelines) and desktop OSes (like prompting user consent for downloads). Communication between clients and servers should be encrypted using TLS to prevent eavesdropping. User authentication via passwords, two-factor methods, or platform logins ensures only authorized users access the game and their data. Authorization controls further restrict access based on in-game permissions.