

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 | 05/25/2025 | Xavier Canidate | Completed Executive Summary, Requirements, and Design Constraints. |
| 1.1 | 06/09/2025 | Xavier Canidate | Updated Evaluation Section |
| 1.2 | 06/22/2025 | Xavier Canidate | Updated Recommendations Section |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room would like to expand their Draw it or Lose it game to a web-based application to serve multiple platforms. Their game is currently available on Android. The game consists of two teams trying to guess what picture is being rendered from a large library of stock drawings. The render takes 30 seconds and if neither team can guess the image before it is fully rendered, each team will have one guess (with a 15 second time limit) at the completed image. This lasts for four rounds.

## Requirements

1. The application must support one or more teams.
2. Each team will have multiple players
3. Unique game and team names
4. Only one instance of the game can exist in memory at a time.

## [Design Constraints](#_2et92p0)

1. The web-based application will have to have a slightly different interface to fit non-mobile or non-touchscreen devices.
2. Must make sure the game checks for other instances across platforms to ensure there are no two same instances or same names being used for games or teams.
3. Design should be adaptable with potential iOS in mind for future proofing.

## [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)

We Start with ProgramDriver which houses our Main() method. This is directly connected to SingletonTester which ensures there is only one instance of GameService in Memory. The entity class has common attributes and methods. Game, Team, and Player inherits these as well. These three child classes also contain their own attributes and methods as well. We can see some aggregation between Player and Team, Team and Game, and between GameService and Game. The Game class is connected to the Team Class with a zero-to-many multiplicity. The Team class is connected with the player class the same way. GameService is connected with Game with a zero-to-many multiplicity.

**"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** | MacOS is Unix-based, which offers a stable environment for applications to run. Mac is compatible with things such as Apache, nginx, and tools like Xcode. These are developer friendly but are overall costly can be around $2,000 - $3,000 for hardware. Mac also has a limited share in the market which could be a cause for concern when scaling. | Linux takes a simpler approach as it is open source and very flexible for the developer. Linux dominates in web server environments such as Ubuntu and CentOS. It is free, secure, and lightweight. However, there is a steep learning curve as the interface is not as user friendly as the other options. You can get a Linux server for around $9.99 a month if you want to host a website. | Windows ensures compatibility on a wide spectrum. The downside to this is vulnerability for breaches. This is typically not the best option for web servers as windows tends to be a resource hungry operating system. This is the most familiar operating system for people which is good for efficient navigation. There are licensing costs to consider when using windows, however, hardware does tend to be more affordable to run windows. | Mobile devices are attractive due to their portability, and easy access to the everyday person. Mobile devices are not typically used for server hosting. You will normally see mobile devices connecting to APIs or servers. That doesn’t mean mobile devices can’t run servers, in fact some are adept at running lightweight servers using Node.js or Python. However, Mobile devices are not reliable for hosting apps on a larger scale. Costs for this can vary from cheap to expensive as there are many options available. |
| **Client Side** | Mac does simplify things with easy-to-use interfaces and streamlined testing. This can be seen as a downside since you will have to test things on mac/safari. Hardware is costly so you must consider that as well. Time could be efficient since the overall operating platform is intuitive. | There is overall less support for a desktop user on Linux. This is the most cost-effective option when speaking from a developer standpoint. It will take time to learn how to operate the environment, but in the long run it will save you time and money. | When using Windows, you are using the system with the largest market share. You will have to test things using Edge or a different legacy browser. Costs can widely vary as well. The time spent and learning curve is relatively tame compared to other options. | When developing a program for a mobile device there are a few considerations. One is the screen size for each device. This along with the resolution can vary greatly. You also want to consider connection limitations, as every user will not have a great connection to the internet when using an app. It is also important to have a responsive and intuitive UI. This is not super expensive to do and can be relatively cost effective. |
| **Development Tools** | Typically, we would use Node.js, and JavaScript for Mac. For IDE’s, Xcode and visual studio are popular options. | When building software for deployment common languages used in Linux include C++, C, as well as Python. Common IDE’s are VS Code, Eclipse, and JetBrains. Git, Docker, and PowerShell are common tools. | Common languages for Windows include C#, .NET, JavaScript, and Python. Popular IDE’s include Visual Studio, VS Code, and Pycharm. Tools we see being used are IIS, PowerShell, Docker, and Git. | Languages include Java for Android, Swift for iOS. Also, Kotlin, Dart, and React Native. IDE’s include Android Studio, Xcode, and VS Code. |

## 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 using Linux as the primary server-side operation platform. Linux is the industry standard for several reasons. It is stable, secure, and open source. This will help keep the cost low. For the client-side I recommend supporting Windows, macOS, and Mobile to reach as many people as possible. This will ensure the app is accessible across phones, tablets, laptops, and desktops.
2. **Operating Systems Architectures**: Linux will typically follow a monolithic kernel architecture; this means all operating system services run in a kernel space. This results in faster performance for web hosting. It’s commonly used in three-tier architecture (presentation, logic, and data), which works great for web-based applications. With this model, the user interface will run in the browser (presentation layer), the logic runs on the application server (middle tier), and the data is stored on a separate database server (data tier). This makes the system modular, scalable, and easier to maintain overall.
3. **Storage Management**: For reliable and scalable storage, I would recommend using cloud-based storage solutions such as Amazon S3, Google Cloud Storage, or Azure Blob Storage. These services are great for storing images, large media files, and drawings that might be used in the game. There are also features such as replication, lifecycle management, and file versioning, which ensures long-term durability and availability of data.

For structured data like user accounts, scores, and game sessions, I would recommend a relational database such as MySQL. This database is reliable, secure, and compatible with web applications. If we need added performance, we should consider using an in-memory cache system like Redis to store temporary game states, and user session data.

1. **Memory Management**: Linux uses advanced memory management techniques such as demand paging, swapping, buffering and caching. Demand paging only loads memory pages that are used by the process, reducing memory overhead. Swapping moves inactive processes from RAM to disk when memory is limited. Buffering and caching helps optimize read/write performance by temporarily storing data in RAM.

These techniques are important for Draw it or Lose it since the game only allows once active game instance in memory at a time. The system will be able to efficiently manage multiple players connecting and interacting without overloading the memory, especially when using a singleton pattern to control game instances. By managing memory dynamically, Linux ensures efficient use of system resources even as the user base grows.

1. **Distributed Systems and Networks**: For smooth communication between different platforms (desktop, mobile, tablet), Draw It or Lose It should use RESTful APIs for regular interactions and WebSockets for real-time gameplay features such as drawing updates or timers. This will allow for low-latency, bidirectional ccommunication between clients and servers.

The application should be built with horizontal scalability in mind – meaning it can handle more users by adding more servers rather than upgrading a single machine. Kubernetes or Docker can be used to containerize and manage different services across multiple machines.

To handle network or connectivity issues, the application should include timeout and retry mechanisms, fallback options, and load balancers and CDNs. These distributed system features make the game more robust, fault-tolerant, and scalable.

1. **Security**: Security is really important when deploying a cross-platform game involving user accounts and personal data. A few practices we should implement are Data encryption, Authentication and Authorization, and password hashing.

Using HTTPS with TLS 1.2+ will ensure all data sent between the client and server is encrypted. Implementing secure authentication methods using Oauth 2.0 or Multi-factor authentication ensures enhanced security of user accounts. We should use secure hashing algorithms like bcrypt for storing passwords. To prevent injection hacks, we implement input validation and sanitization which will ensure all user input is validated on both client-side and server-side. Regular security patches should be done to stay up to date with the latest security tools.

All traffic between platforms should be protected using API Keys, rate limiting, and firewalls to reduce the risk of brute-force or DDoS attacks.