

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 | 09/22/2024 | Zach Fizet | Drafted initial software design document. Included executive summary, requirements, design constraints and more. |
| 2.0 | 10/06/2024 | Zach Fizet | Added development requirement analysis for Mac, Linux, Windows, and mobile devices. |
| 3.0 | 10/20/2024 | Zach Fizet | Added Operating Platform, Memory/Storage, Security, etc. and other recommendations. |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room is a company with a mobile app game called Draw It or Lose It that is available on Android devices. The game Draw It or Lose It is based loosely on the television game from the 1980s called Win, Lose, Draw. However, in Draw It or Lose It, instead of players drawing a picture, the application renders images from a large library of stock drawings as clues for players. The Gaming Room has asked Creative Technology Solutions (CTS) to design a platform-agnostic, web-based application version of their successful game. The web version of the game will replicate the functionality of the existing Android application, ensuring familiarity and ease of use for new and existing players.

## Requirements

A game must be able to have one or more teams.

Each team will have multiple players assigned to it.

Game and team names must be unique.

Game and team names must be checked to ensure uniqueness.

Each instance of a game, team, or player must have a unique identifier.

## [Design Constraints](#_2et92p0)

The application will need to have multi-platform compatibility. In the instance of a web-based application, this would include working across different browser applications (Chrome, Safari, Firefox, etc.) as well as mobile web browsers. The application will also need to be able to handle multiple connections as there can be many different teams and individual players connected to the application at any given time. A database environment will be necessary to ensure that the uniqueness constraints are met. Additionally, the game will need a mechanism that keeps track of time, as the game uses different timings to function as a main mechanic of the game. The game will also need an extensive library of stock drawings to use as part of the game’s function. The application will also need to ensure data security to protective sensitive user data and to ensure game sessions are kept private and secure. Lastly, the UI of the web application should follow a similar style and process as the already established Android application, ensuring ease of use and familiarity with existing 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)

<Describe the UML class diagram provided below. Explain how the classes relate to each other. Identify any object-oriented programming principles that are demonstrated in the diagram and how they are used to fulfill the software requirements efficiently.>

Program Driver – This class contains the main() method and is the “driver” of the application. This class uses the SingletonTester to test if the GameService class is correctly implemented as a Singleton.

SingletonTester – Has the testSingleton() method that verifies the Singleton pattern in the application. It interacts with GameService to verify that only one instance is created and used throughout the application.

Entity – Entity is a base class that contains the attributes id and name. These are common attributes across other classes in the application (Game, Team, Player).

Game – This class inherits from the Entity class. It represents a game that has a list of Team class objects. It also provides methods to add teams and to display game information. Game has a one to many relationship with the Team class.

Team – The team class also inherits from the Entity class. Each Team has a list of Player objects. The relationship between Team and Player is one to many, meaning that a team can have multiple players. The Team class also contains methods to add players (addPlayer()) and for displaying team information.

Player – The Player class also inherits from the Entity class. The Player class represents one individual player with an ID and name. The class also contains a method to display player information.

GameService – This class follows the Singleton design pattern, which ensures that only one instance of the service exists. This is necessary because it fulfills the requirement of maintaining a unique, single instance of the game. This class also manages game functionality like: storing all of the current games in the games: List<Game> attribute, keeping track of unique IDs for games, teams, and players (nextGameId, nextTeamId, nextPlayerId). The GameService class also has the functions to add, retrieve, and manage games (addGame(), getGame(), etc.). It also contains the method to access the Singleton instance of GameService.

**"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** | Mac is uncommon for utilizing it as a server host, but Mac can be used via macOS Server. Mac works well with other Apple devices within the apple ecosystem as they all work mostly seamlessly with one another (iPhone, Macbook, iMac, etc.)  Strengths – Mac features strong security features. Mac has a feature called Gatekeeper which requires that applications be signed by devs with a verified Apple Developer ID, preventing any “untrusted” software from being deployed. Additionally, Mac features a local full-disk encryption program called FileVault that protects the data within the server.  Weaknesses – Mac hardware and licensing can be very expensive. Mac is also limiting in nature due to the fact it is developed primarily to work within the Apple ecosystem. The tools for the server are more limited when compared to Linux or Windows. | Linux is very popular for server hosting due to the fact that it is open source and very flexible.  Strengths – Linux is completely free to use, so there are no licensing costs associated with using Linux. Linux is widely supported across the industry with cloud platforms like AWS and Azure. Linux is completely customizable, meaning that the sky is the limit when it comes to tailoring what kind of features you want and what you don’t need. Linux also has a wide variety of community support with server management tools like Apache.  Weaknesses – Linux requires a certain expertise, potentially needing developers that are familiar with working with Linux architecture. Linux is also not very friendly for non-technical users. | Windows is arguably the most popular and most familiar operating system for most people and businesses.  Strengths – Familiarity, because it is Windows, organizations already using Windows may benefit from utilizing Windows as a server host as it is an environment they are already familiar with. The user interface is much more friendly for inexperienced users. Windows also supports seamless integration with other Microsoft Services.  Weaknesses – Licensing costs can be high and there are a limited amount of open source tools available for use due to the fact that Windows is not open source like Linux. | Mobile platforms are not utilized for hosting servers. However, if the backend were to be hosted in the cloud, it could potentially be accessed by mobile clients.  Strengths – Cloud based server hosting is well optimized for mobile platforms.  Weaknesses – Developmental tasks on mobile devices can be complex or impossible to execute due to limitations of mobile devices. |
| **Client Side** | Costs for Mac will potentially be higher due to the fact that Mac/Apple devices are generally more expensive, licensing via Apple could also potentially be the most expensive of the options. The time to deploy via Mac OS Server will, in general, be much slower than other options as Mac OS Server lacks widespread support. The amount of expertise required is limited to the familiarity with Mac OS servers and limited by Apple-specific tools like Swift, etc. | Linux will be the least expensive option when considering cost as there are no licensing fees. However, developers will need to be familiar with Linux-based environments, leading to potentially more expensive developments overall (salaries, etc.) Deployment time will fast due to Linux’s widespread compatibility. However, this option will require expertise with Linux, web servers and different scripting languages. | Windows costs will be similar, but potentially less expensive than Mac because of licensing required for Windows Servers and other Microsoft services that will potentially be used by developers. Deployment time should be relatively fast as Windows servers can integrate easily with other Microsoft tools, but some there may be some learning curve if any sort of open-source tools were necessary. Requires expertise in ASP.NET and other windows-specific server management. | Costs for mobile development can be low if utilizing cloud hosting, but expertise in mobile development like Swift, Java, and other mobile-friendly web technologies and design frameworks will be necessary. |
| **Development Tools** | Programming languages – Swift, Objective-C, Python, Javascript.  Tools – Xcode, Visual Studio, PyCharm, Apache, Nginx, Git | Programming languages – Python, PHP, Node.js, Ruby, Java  Tools – Visual Studio, PyCharm, Eclipse, Apache, Nginx, Git. | Programming languages – C#, ASP.NET, JavaScript, Python  Tools – Visual Studio, Eclipse, PyCharm, Git, IIS. | Programming languages – Swift, Objective-C, Java, Kotlin.  Tools – Xcode, Android Studio, Flutter, Git. |

## 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**: My personal recommendation for an operating platform to ensure the widest range of cross-platform compatibility would be to use Linux as Linux supports multiple programming frameworks, enabling seamless development and hosting of web-based games that are accessible from Windows, Mac, Linux, and even mobile devices. Linux is capable of handling high server loads, which is critical for “real-time” games like Draw It or Lose It. Linux is also the most cost-effective option, as there are no licensing requirements necessary for using Linux as many Linux distributions are open-source and free. Linux also features strong security features like file management, user permissions, etc. that will ensure the game operates safely and securely. Cloud providers also offer excellent support for Linux servers.
2. **Operating Systems Architectures**: Linux uses a model of architecture where all of the necessary/essential services operate within the same space. Linux is also POSIX compliant, which allows for it to be easily ported to MacOS and maintain compatibility with Windows. Linux also has virtualization support, allowing developers to run and test the game on multiple different operating systems within the same hardware, which can be helpful when testing compatibility.
3. **Storage Management**: For storage management, I would personally recommend using a Cloud-based storage system with an SQL database. A database system would be helpful for structured data management like handling user accounts, game states, team details, and more. Amazon, Google, and Microsoft all offer cloud storage options at various price points. Cloud storage is also relatively cheap and makes storage management essentially seamless. Cloud storage also typically has built in features like automated backups, ensuring that important data is always recoverable.
4. **Memory Management**: Linux uses virtual memory to provide more memory for handling processes than what is physically available, which ensures smooth execution of the game. Additionally, only necessary parts of the program, reducing the overall memory usage. Data will also be swapped to disk storage if all RAM is exhausted. This is probably not ideal if coupled with cloud-based storage, at this could potentially affect game performance, but it is an option. Linux also aggressively caches files and frequently used data to speed up application performance. If the same data is used over, and over, and over, that data will be cached and called efficiently.
5. **Distributed Systems and Networks**: REST APIs can be used for non-real-time tasks and for connecting players across devices/networks. Websockets can be used to handle game tasks. Websockets enable real-time communication between a client and the server once a connection has been established. This will allow the game to have things like live score tracking and reduced latency for game actions. For example, if an image is loading on a 30 second timer, it would be very detrimental if you experienced an additional 10-15 seconds of latency due to the connection between client and server. Websockets ensures that communications happen in real-time. Additionally, cloud networking tools can connect servers across different platforms and regions. Lastly, some sort of automatic reconnection mechanism would be helpful for players that experience an outage when playing the game, allowing them to reconnect automatically when a connection is available.
6. **Security**: TLS/SSL encryption is a must for transmitting data between client and server and AES encryption would be my recommendation for any stored data. AES-256 is considered the strongest form of encryption available and touted as an industry standard for some time now. While it might be slightly overkill for use in this situation (depending on the sensitivity of data stored), it is still extremely reliable and secure. I would also recommend implementing some sort of two-factor authentication. This would require users enter some sort of authentication code that is generated and sent to authorized devices upon login. This can be done via text message or using a pre-existing authenticator app. This ensures account security, greatly reducing the risk of unauthorized access to user accounts. Linux and cloud services feature firewalls that can filter out malicious traffic that can potentially overload the servers. Leading to them being taken offline and causing gameplay disruptions. Additionally, I would recommend using a secure API gateway as a centralized access point for controlling access to backend services and APIs. An API gateway can enforce things like authentication and authorization, rate limiting/throttle requests (limits the number of requests a user can make within a certain time frame so they don’t singlehandedly overload the server), input validation, and more. An API gateway would, overall, enhance the security of the entire application.