

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 |
| --- | --- | --- | --- |
| 1.0 | 11/14/21 | Alex Jarratt | Creation of Document |
| 2.0 | 11/28/21 | Alex Jarratt | Update to Development Requirements and Evaluation |

**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 currently has an Android mobile application called Draw It or Lose It but would like to develop a web-based version of the game for other platforms. The game’s structure contains multiple games which proceed to host multiple teams. Each of these teams has several players. For the game to work as intended each game and team name needs to be unique to other game and team names. Draw It or Lose It, the mobile app, contains a large library of stock drawings for use by the players. The staff at The Gaming Room do not know how to create the environment for the web-based version of their mobile app.

## [Design Constraints](#_2et92p0)

There are three differing software development methods from the platforms requested. These three are:

* Android
* IOS
* Web-based

The API needs to work with all three of these software development methods equally.

Several teams must be available from each of these platforms.

Each team and each ‘game room’ must have a completely unique identifier that differs from the other games/teams currently running.

There must be a method to check for if a team or game has that name before creating said team/name.

If the answer is yes, tell the creator of the game/team to make a new name. Otherwise, the game/team is made.

Utilize differing and randomized strings for each instance of a game/team/player, so there is no repeating each game/team/player.

## [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)

ProgramDriver is a Class.

ProgramDriver contains the main method.

ProgramDriver is using Directed Association in regards to SingletonTester.

ProgramDriver is using SingletonTester to test for a single instance of GameService.

Entity class is a parent-type class for three child classes:

* Game
* Team
* Player

Game, Team, and Player child classes, as a direct result, inherit Entity’s required attributes.

(Though this is self-evident)

The next few sections of the Domain Model are in possession of a one-to-many relationship:

Teams have a one-to-many relationship with Player.

Game have a one-to-many relationship with Team.

Game Service has a one-to-many relationship with Game.

Game Service must only have one instance of each game running at any time, as checked by SingletonTester.

Each Game can only have one unique Team at any time. (Whether this is checked by SingletonTester is unclear)

Each Team can only have one of Each individual Player at one time. (Whether this is checked by SingletonTester is unclear)

Assumption: SingletonTester does check ID/name for their identifier and if it is a unique identifier.

**"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** | Licensing is the first issue. That is a monetary investment. Additionally, the demand for Mac-based products has lessened as Windows has risen.  Furthermore, the OS is inundated with additional unnecessary features, which drives up cost more. The technical costs are lower, however, as the implementation of Mac is easier and more in-line with OOP than other OS. | Linux is a prime choice because it is free and is used for this purpose already, making Linux servers a well tried and tested platform for success. Overall the cheapest choice for the open-source application and zero licensing fees. However, requires usage of the command line style of administration. | Windows servers are highly secure and easy to put in to place. This comes at the cost of licensing issues, however. Additionally, there are technical constraints. The price rapidly increases as the scale of the project increases, making this an overall poor choice. Best for administration-based functions, worst for the user. | While potentially an option, the lack of power and the specialized software needed to make mobile devices servers make this a poor choice. There are some frameworks for this side, but the internet-required computer support for them makes this the worst choice. |
| **Client Side** | Requires usage of Swift. Also requires Mac-books for development. This is an additional money sink, making this a not great choice. Works well with REST API. | Requires usage and knowledge of Python. Supports low amount of file structures. More programming would be required, as the media types available to see ‘out-of-box’ is potentially not enough. | Recommend .NET, but supports REST API. Works best with Microsoft browsers, but not necessarily what all client-side will be operating on. This is another technical constraint. | Considering the Gaming Room already has an Android app, this would be their most experienced option in-house. However, this is positively the worst of the choices here. Not enough processing power and no plugin support means this would all happen on the servers. Android Studio, iCode, or UX would be required and these do not always work well with each-other. |
| **Development Tools** | This would utilize a Macbook. The coding language that would be used would most likely be iCode. However, to avoid a licensing fee, there are open-source IDEs that are available to be used. This does allow for multi-threaded one-to-one architecture. Also plays nice with the OOP style of programming. | Python is already preinstalled. This also has multi-threaded one-to-one architecture. This version, however, has added overhead costs. There are complied languages to offset some of this cost. Python is not perfect for OOP, however, so it might reduce scalability in the future. | Visual Studio is usually preinstalled. It is also considered the best for coding Windows-based applications. Other options include C# or C++. A one-to-one architecture. Has been optimized for the .NET framework. Either C# or C++ can work with OOP style. If using C#, there is a technical constraint on scalability. | The usage of Android, iPhones, or Unity would be the smartest. Android’s are best in conjunction with Java, but iPhones are best used with iCode. Unity would be done through C++ and then manipulated into an application for further development. If using Unity/C++, this would also require a Macbook for the iPhone support. This would require a lot of experience in many differing frameworks to utilize best. |

## 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 recommend the latest edition of the Linux OS. Specifically, I recommend a cloud-based, or server-less, format. This will help scaling the game up to meet a growing user population. It also allows for multiple access points, such as with HTML5 and API. This also allows for various computer-like devices to access as well. Meaning, Mac products, Windows devices, various phones that do not have a phone application and more are all able to access the game through this format.
2. **Operating Systems Architectures**: This would best be made in a three tier system. This has a client-side, a server-side, and a middle platform between. This middle ‘layer’ allows for the transfer of information between users and servers in an advantageous manner. This stems from the benefits it offers a cloud-based format. They can take this middle layer and format it into a modular system that opens and closes users to servers as necessary. This also means that the Game Room only has to pay for the parts of this middle layer that it needs. The OS only needs the absolute core for the game to operate on while the inclusion of modules allow for the Game Room to expand their systems in the needs of the platform directly. Linux itself runs on something similar, where it does not access the core files. This promotes system security as well.
3. **Storage Management**: Because we are using a server-less platform, the best form of storage management is a distribution over ‘servers’ in a layer from the OS Architecture. These servers are not really servers as they are part of the OS itself in just another layer that is accessed and copied as the platform requests information. As such, the primary method of access to the storage is direct. This neatly allows for virtualization to be added as well, making loading times fast and bypasses some memory requirements. We simply make sure there is an index for the files and virtualization excels. Coincidentally, Linux uses this very system for storage. Linux manages to reduce the time to find files even faster by making sure that files are grouped into ‘blocks’ where the search function calls a ‘block’ then the file it needs.
4. **Memory Management**: As said previously, virtualization helps reduce the need for this. However, the most important part of memory management is the OS. In Linux, partial memory exists. This allows for the code to be made of parts that run as needed, promoting modularity. However, the overall amount of code needed to run a process is larger. Thus, some maintenance on the code for bloat would be necessary from time to time. This would allow for the CPU to run the code in partial memory, where Linux would utilize two different aspects to reduce memory requirements. These two aspects are called virtual memory and demand paging. This means that the memory is actually online, reducing the need for physical memory. Additionally, this allows for multiple instances of the program to run simultaneously. This also improves the applications speed on loading even more.
5. **Distributed Systems and Networks**: This is where server-less systems shine. Because there are no servers, the scaling of the application is tied directly to the amount of users. As long as the application utilizes API’s, which it should be, it could work across any platform. The usage of the Internet’s HTTP statelessness allows for a strong method of communication between the user and the server which allows for REST API to be used. REST API is a form of HTTP communication for users/servers, which works nicely. Additionally, REST works well with server-less architecture. Server-less architecture utilizes functions to form the messages in the communication system, where REST does this well with its various functions. Server-less architecture still uses some machines to act as physical servers, but are very much not as much a server as the other forms of architecture are. This means, eventually, there will need to be a small system for these ‘servers’. In this case, there should be one main server and the rest tied to it. This allows for master/slave replication. Master/slave replication creates one writing and many reading databases. This is because users will be requesting images more than there will be admins creating new images to be added, so reading databases will be able to pull for the users while the single or few writing databases work for the admin to improve and update the application.
6. **Security**: The easiest and simplest method of security is the one that works best here. The other methods are too difficult to implement and are, ultimately, unnecessary. The usage of something like a fingerprint scanner is too unwieldy for the application. The simple solution of a username and password is perfectly acceptable here. This can include a two-form authentication process for added security measures, but for the base of the application, simply a username and password is fine. To prevent users from changing the code, roles are an easy way of limiting the public’s access to files they do not need to access to play the game. Users, Moderators, potentially even Administrators are all easy to implement roles with more security measures along the way. More passwords, specific admin accounts, additional authentication methods, and more are all easy ways to prevent an accident from the public changing the game. Linux promotes this style as well because it utilizes the independence of processes. Meaning, Linux has each process work in virtual space independently of other processes so that nefarious programs do not interfere with other programs. Linux also utilizes the methodology of discretionary access control, where file owners lock down access to parts they want to keep out of the hands of people. Linux adds to this process through its ACL system. Thus, simplicity is the best for security.