

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 | 09/19/2024 | Cody MacLeod | Initial design |

**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)

## Requirements

Our client, The Gaming Room, has a game called Draw It or Lose It, and they would like to expand past the current Android only application to a web-based game so it can be accessed on multiple platforms. Per the clients’ requirements, there must only be one instance of the game at a time. This can be resolved by utilizing a singleton design pattern to ensure there is only ever one instance possible to be created at a time. There also needs to be unique names not just for the game instances, but also for team names from which the user will be able to input themselves. We can use the iterator design pattern to search through a list of already used names to ensure the new name is unique. These teams should be able to have multiple players, and there can be more than one team per game. From a business standpoint, we will need to keep in mind the budget and timeline available for the project, making sure to include ample time for testing as we are aiming for this game to be available on Windows, Mac, Linux, and mobile across a variety of different browsers.

## [Design Constraints](#_2et92p0)

As mentioned, the client wishes to create a web-based version of the game, meaning that when designing the new application, we will need to carefully consider what will work for the game to be able to run on Mac, Linux, Windows, and mobile systems. The user should be able to have the same experience regardless of platform, so not only does the application need to be compatible with the aforementioned systems, but also uniform across them. This will make the available timeline for the project an important constraint as we will need plenty of time to test for that compatibility. Not only would we test across the various systems, but also for multiple browsers on each one to give our client the ability to reach out to as many potential new users as possible.

## [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)

In our UML diagram, the Entity class serves as a base class to pass down common attributes and common methods to each of the child classes. The Game, Team, and Player class all extend the Entity class, meaning they all inherit the id and name attributes, as well as the common methods such as the accessor methods. With the toString() method, each of those child classes all showcase an aspect of polymorphism by overriding the method with their own. This allows those methods to change the text returned to one more appropriate for the individual classes. Connecting the child classes are arrows showcasing the zero to many relationships between the classes. This showcases how each team can have multiple players per team and each game can have multiple teams per game, but they all must be unique. Inside the addPlayer and addTeam functions of the team and game classes we will use the iterator pattern in order to ensure that each added element is not already in our lists. The iterator pattern is important for both encapsulation and abstraction as it is used to iterate through our various lists without having to be aware of the contents. Not only does this prevent accidental changes to the objects, it also adds a bit of extra security to those finer details.

The GameService class utilizes a singleton pattern with our private constructor and getInstance() method returning a GameService object. This ensures that only one service can exist at a time as per our clients requirements. This class also contains the getNextGameCount(), getNextPlayerID(), and getNextTeamID() methods which are able to be accessed by the addTeam() and addPlayer() methods from the zero to many relationships. This ensures that the IDs used by these classes when adding a new player or team to the list are all unique.

The last part would be the ProgramDriver and SingletonTester classes. Inside our ProgramDriver is the main method, from which our application will function. In it, we create our game instance and create a few example games to confirm functionality. We also use our SingletonTester class to ensure that our implementation of the singleton design pattern functions correctly.

**"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 capable of hosting a web server for our game, though it would not be my first recommended choice. The biggest advantage of Mac, particularly compared to Linux, is the lower technical expertise required as Mac has quite a user-friendly GUI. The downside of utilizing Mac for the server side would be the costs, where the licensing of the operating system is expensive compared to some of our other options. Older Mac devices may have server-based deployment methods through Mac OS X Server, but support for it was ended in 2022. Using these older devices that can still deploy through it can bring potential security issues. | Linux is a fantastic choice for the server side of this project. Of all the operating systems, it is the most customizable and can run on nearly any machine. Being a free and open-source project means it does not have the expenses to license like both Mac and Windows do. The downsides of Linux come from the technical expertise required to set up the OS. Without a GUI, it can be a bit more confusing for the average person. | Windows is also perfectly capable of hosting the web server for the project. Having the largest market share, Windows would have the most support available online. Similar to Mac, it also has a user-friendly GUI which reduces the technical expertise needed to set up the server. On the other end, Windows shares similar downsides to Mac. It has expensive licensing costs compared to the free Linux systems and requires stronger hardware to run the OS. | While mobile would be able to host if needed, it would be the one I’d recommend the least. As an advantage, mobile operating systems are more portable than most other systems. This comes with a steep downside of not having a wired connection, meaning there is potential for connectivity issues. Security wise, it would also be the weakest with Mac, Linux, and Windows all having considerably more choice for security. |
| **Client Side** | On the client side, users would not need much expertise to run this project on a Mac. Being able to open a browser and visit a webpage is all the users would need to do. On Mac’s the user has a few options for browsers they could use such as Safari which comes by default with Mac, or more universal browsers such as Chrome or Firefox. | Linux is also plenty functional enough to use on the client side, though proper set up of the operating system requires a bit more technical knowledge compared to Windows or Mac which are a little more user friendly. Once the user has the operating system set up, they would only need knowledge of how to open a browser to access this project. On Linux Firefox is typically the default browser the comes with it, but the user could also install Chrome or Edge. | Window’s similarly to the previously mentioned operating systems is easy for the client to use. Having access to the most amount of software of all the operating systems, the user has plenty of ways they can access the game. Simply having knowledge on how to open a browser is all the users would need for this platform. By default, modern Windows systems come with Edge, though the user has many other options such as Chrome or Firefox. | Mobile devices are a great platform for users to access our game. The variety of devices and portability of them make them ideal for the user to be able to play the game where they would like. This does come with some extra time spent to ensure the game works uniformly for the wide variety of screen sizes from different phone or tablet devices. While we already have an Android application, making the site accessible from a browser on Android devices could help expand to more users, of which there are quite a few options such as Chrome or Firefox. On the iOS side users should also be able to use Safari which comes by default with most devices alongside other browsers such as Chrome. |
| **Development Tools** | To make best use of our toolset, we’ll use polyglot programming to design our application by separating the front and backend. This may require multiple teams depending on the experience of our programmers with the different languages. On Mac devices, we’ll utilize HTML, CSS, and JavaScript for the front end of this project, creating the website the user interacts with. Java will work great for the backend for this project on the Mac side. As for IDE’s, there are a few options for the Mac side. For the front-end languages, Atom or Visual Studio are both available on the system. For Java on the backend, Eclipse is also usable on Mac devices. | Linux can similarly be split in the front and back end, which may result in needing multiple teams. On the actual website part of the project, the same languages of HTML, CSS and JavaScript will be used. Java also works well on Linux and can be used for the backend of our project. Linux has quite a few IDEs that can be used, such as Visual Studio or VS Code for a more lightweight IDE. When it comes to Java, Eclipse will work on Linux as well. | The biggest advantage of Windows is that it has the highest number of options for tools and programs to use on this project. Similar to Mac and Linux, we can once again use HTML, CSS, and JavaScript for the front end of the project, and Java for the backend which means we might need multiple teams. While many could be chosen, Visual Studio or VS Code are common choices for HTML, CSS and JavaScript, and Eclipse will once again be a great choice for working with Java on Windows. Windows also has quite the variety of emulation tools to help test for other platforms. | When it comes to mobile systems, most of the actual programming would be done on another system then tested on the platform. There are some IDEs specifically designed for mobile app development such as Android Studio for Android devices, or Xcode for iOS devices. Android Studio would be a good choice for utilizing Java on the project. Xcode meanwhile uses Swift to optimize for the iOS platform. A major part of developing mobile devices will be testing. Having a variety of devices or emulators to simulate different devices will help ensure our web application works on as many mobile devices as possible. |

## 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**: While any of the platforms could be used to create the web application for Draw It or Lose It, the one I would recommend would be Linux. Other platforms are considerably more costly to use, and while Linux does require a higher barrier of entry to use, it is the most popular platform for web servers which will be useful for finding developers for the team or finding support for any issues that arise. Linux also has robust options for security, which will be useful for protecting the servers from potentially malicious traffic.
2. **Operating Systems Architectures**: Based on the requirements of this game since there will need to be multiple instances open at once, we will be utilizing a client-server pattern as our architectural pattern. The backend would host the actual game instances, but the actual rendering of images would be done on the front end with the client’s devices. Those front-end servers would be accessed from various web browsers, of which Linux can natively utilize some of the most popular such as Chrome and Firefox, making it useful for testing purposes. We can also utilize a reactive language like React Native to make the process of porting to other systems easier. On the physical aspect of the server, we would need decent RAM to help properly display images without stuttering, though the bulk of the storage of all the 200 HD images should be done using cloud storage such as AWS as it is easy to access and easy to scale up should the client wish to build upon this project. At the core of Linux systems is the kernel which controls and allocates resources to the rest of the architecture to ensure everything is functional. Linux comes with many security options to help keep the kernel safe and secure.
3. **Storage Management**: For storage management, utilizing Amazon Web services would be recommended. Of all the cloud storage providers, AWS tends to be the cheapest available and is quick to access the data as needed. Cloud services also have the advantage of providing an extra layer of separation from the project in case something happens to the main machine, meaning our images would still be safe in the case of an emergency.
4. **Memory Management**: Memory management is an incredibly important part for our game as the image is expected to be revealed slowly to the player over the course of 30 seconds. If we do not use proper memory management techniques, that rendering process could slow down resulting in the player running out of time before the image is properly visible to them. This is where our consideration of random-access memory (RAM) is important for the images currently being shown. Since we have 200 images, storing them all in RAM would be rather intensive, especially considering the wide variability of user devices between all our platforms, so utilizing techniques such as caching to make the data easier to pull into RAM as needed would be beneficial for ensuring the user has a smooth experience. Looking at our clients’ potential devices, particularly the mobile platform, may not have the strongest devices, so we’ll want to be as efficient as possible with our memory and utilize a technique such as paging.
5. **Distributed Systems and Networks**: As previously mentioned, having our data stored in a cloud server helps keep the images safe in the case of disaster such as disk failure, and proper caching of the images can help us access them in the case of connectivity issues with the cloud provider. Since our project was designed based on the fundamentals of Restful API, we will be utilizing HTTP requests to communicate between the server and user devices, helping provide a consistent experience across any of the supported platforms.
6. **Security**: Linux comes with its own robust options for security and our cloud host, Amazon Web Services, also has additional tools of their own for security. In the actual game itself, we will utilize role-based access control to limit users to only having access to what they would need. Furthermore, utilizing passwords to authenticate the user will help ensure expected users have access to administrator accounts. We would want to set up requirements for stronger passwords rather than those easily guessed by a dictionary attack, and potentially utilize another avenue of verification to have two factor authentication. We’ll also want to encrypt our data so people on the same network as the user can’t simply listen in to communications and have access to private information.