

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 2.0

## 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)3-4

[**System Architecture View**](#_ilbxbyevv6b6)4

[**Domain Model**](#_8h2ehzxfam4o)4-5

[**Evaluation**](#_2o15spng8stw)6-7

[**Recommendations**](#_m8aleynsvzvc)8-10

## [Document Revision History](#_grjogdjh5fi8)

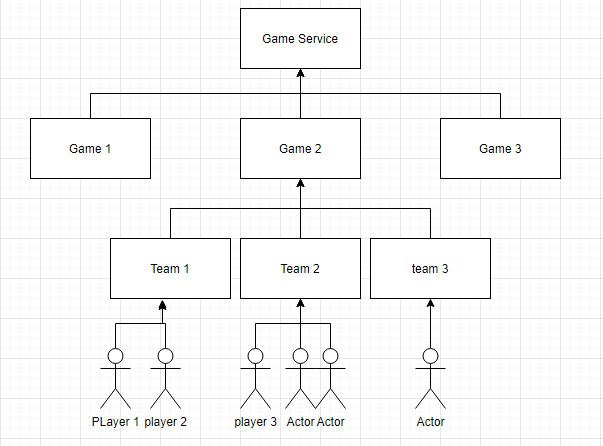
| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 11/13/2020 | Ryan Mackenzie | Creation of document and research completed |
| 1.1 | 11/14/2020 | Ryan Mackenzie | Executive summary, Constraints, and Domain Model summary complete |
| 2.0 | 11/29/2020 | Ryan Mackenzie | Completed research and the Evaluation section of the software design document. |
| 3.0 | 12/12/2020 | Ryan Mackenzie | Research performed, completed some of the recommendations portion. Updated Citations. |
| 3.1 | 12/13/2020 | Ryan Mackenzie | Finalized the recommendations section and the document. |

## [Executive Summary](#_sbfa50wo7nsh)

Our project will be to complete the Draw It or Lose it game with some specific requirements. The game will need to be converted from an Android application to a web-based application that serves multiple platforms. Our game will consist of multiple teams, who will have to take turns trying to guess a picture as it is drawn. They will have 30 seconds to guess the answer to the drawing. If they do not, the other teams simultaneously get 15 seconds to simultaneously submit a guess as well.

Our game will need to be capable of having one or more teams play, with each team containing multiple players. The names of both the game and teams must be unique. Only one instance of the game can exist in memory at any given time.

We can solve the problem of having one unique instance by creating unique identifiers for each instance. Using the singleton design pattern, we can ensure there only one instance of a game can exist at any given time. Teams need to contain an array list of each player on the team, and the game needs to contain an array list of the teams playing. The game service class will store a list of each game. The below diagram shows how the games, teams, and players will be stored.



All will likely have name and ID attributes that they will inherit from a template class called Entity. This will help keep our other classes small and easy to maintain. A game service class will interact with the main and create the games, teams, and players by calling on those classes. The game service class will handle all the validation involving names and IDs for players, games, and teams.

## [Design Constraints](#_2et92p0)

We have been provided with Java code to start with from the Android application, so we will need to develop the application in Java. Typically, Java works well for most applications and operating systems however it is slower and takes up more memory than languages like C++. Java has a garbage collector that can affect performance every time it is run and must be interpreted into machine code unlike C++. We need to keep that in mind as the game scales larger because JAVA can be too slow when needing to crunch a lot of numbers very quickly and handle a lot of stored memory. This is an important technical design constraint to consider. Our team should consist of experienced Java programmers.

Currently, Draw It or Lose It currently must be downloaded locally onto an Android device to be run. To expose our game to a larger player base, we need to convert this application into a web-based application that works on multiple platforms. This means we need to have the application loaded onto a server that can be accessed via URL. Anyone on the internet should be able to access the website and play our game regardless of the platform. This means we need to consider how our program will handle situations where our players lose connection, and how our application will interact with each operating system. This technical constraint should be handled during the development process.

Web based environments bring about security issues that we need to consider. If someone knows the name of someone else’s game, they could load it up and start messing with it. We need to implement password protection for each game to ensure fair play and prevent tampering. At the same time, we should try to make our program as secure as possible to prevent users from hacking into the system by entering code into certain fields. This is another important technical constraint to consider.

Only one instance of any game can exist at one time, or issues can arise in our program with users’ games being overwritten and lost. We must consider using a singleton design pattern with our program to ensure unique identification of all games, teams, and players can be stored and protected. This will also prevent users from launching two instances of the game at the same time. This technical constraint can be handled with the design pattern we use for each class.

We need to establish who is hosting the web-based environment and the costs. There could be restrictions on the amount of data we can use, and our game service could be throttled if it receives too much traffic. This is an important business constraint that needs to be considered.

We should do product testing after development is completed to find the minimum connection speed required to play the game. If our game requires a high-speed internet connection, not everyone will be able to play it. We may be able to reduce the amount of speed required through the development process, but this is an important technical constraint to consider for a web-based environment.

## [System Architecture View](file:///C:\Users\Ryan\Desktop\CS%20230%20Project%20Software%20Design%20Template%20(1).docx#_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)

Our UML class diagram below shows an extremely detailed view of the classes Draw It or Lose It will include. It also describes the attributes, functions, and relations of each class to each other. By looking at the diagram below, you can get a clear picture of how the program will need to work. One thing stands out right away is each game, team, and player must have a unique name and identification number. This is a situation where creating a template or base class that contains those common attributes is appropriate. That way, we can simplify each class further making them easier to understand and troubleshoot.

The Entity class will have a private long storing the ID, and a private string for storing the names. The entity class will contain several functions that the game, team, and player classes will all utilize. The Entity class has a public constructor that can be used to create any one of the three previously mentioned classes. It also contains functions to return the stored names and IDs. These functions can be used for each of the game, team, and player classes so they will be included in the base class. The default constructor for the Entity base class must remain private as we do not want any games, teams, or players being created without having unique identifiers. This is required for the singleton design to function properly. I will not dive into the toString methods too deeply as typically every object has a method like that which prints all stored object data to the display/console when called.

We have the Game, Team, and Player classes all extending the base entity class. Each of these are classes designed to create objects that can be stored. They will inherit all attributes and functions of the Entity class that are common which is represented by the white filled arrowhead. The functions and attributes inherited from the base class are mostly there to ensure that each game is unique, and that each team and player within that game are also unique. The Game class has a zero to many relationships with the Team class, and the Team class has a zero to many relationship with the Player class.

The Game class has a private teams array list for storing the teams. The public Game function in the Game class is an overwritten version of the constructor in the base Entity class that specifies that you want to create a game. The constructors for the Game, Team, and Player classes all take the same parameters so the system will not know what you are constructing without specifying. The addTeam function will call the default constructor from the team class and create a new team object to be stored in the teams list.

The Team class is very similar to the Game class, but its list stores players instead of teams. The constructor is again an overwritten version of the one inherited from the Entity class. It has a public method designed to accept a name input from the user as a parameter and creates a new player. Only a Team object can create a Player object using its addPlayer method.The Player class will exist to create player objects and simply consists of an overridden version of the base constructor and a toString method.

The GameService class performs a great deal of the work involved with the program. It has a private list of attributes that are there to temporarily store id values taken from Game, Team, and Player objects. It also has several public “get” methods that are designed to load existing games or provide more information about what is stored. The method that creates a new game is designed to check to make sure the name does not already exist. GameService creates a private local instance of GameService and a private constructor. The getInstance method calls the privately created instance of GameService. This ensures no additional GameService instances are created, which is what makes it a Singleton program design. Additional GameService instances would cause unnecessary complexity in the program and potentially cause errors. Finally, the ProgramDriver is there to interact with the user and drive the program and uses the SingletonTester class to ensure the program is working as intended.

****

## 

## [Evaluation](#_2o15spng8stw)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | MacOS offers an application designed to allow you to use your device as a server. According to <https://www.apple.com/macos/server/> you can download it and set it up for only $19.99. The server application has some basic hardware requirements such as 2GB of RAM as well as 10GB minimum storage. Current MacOS has a minimum requirement of 4GB of RAM and 35.5GB of storage. You will need a device that has at least 6GB of RAM and has more than 45.5GB of storage. | Linux has several server application options with the most popular being the Ubuntu distribution of Linux. It has a feature called Ubuntu Server which can be used to create a server. Linux is open source and free, so the costs of creating a server on Linux would be minimal. Linux is also a very stable OS, meaning it does not crash or have issues often. Ubuntu also has some hardware requirements such as 4GB of RAM and 25GB of storage. From what I have found, there is no up-front cost to setting up an ubuntu server. | Windows offers an application called Windows Server; however, it can be quite expensive. Looking at <https://www.microsoft.com/en-us/windows-server/pricing> the least expensive option is $501 and can range all the way up to $6,155. There are other, less expensive 3rd party options available for windows like Ubuntu. It seems like Windows Server is the best option for large scale operations that have heavy traffic. Windows Server needs at least 512MB of RAM and 30GB of storage space. | There are 3rd party applications you can download that allow an Android device to host a web page and act as a server. These seem fairly limited however and may not work if we need multiple pages or have heavy traffic. If you traveled somewhere that had no internet service, your clients would lose access to their data and the website. Here is a free one that I found on the Google Play store which could be useful in some cases: https://play.google.com/store/apps/details?id=ar.com.lrusso.tinywebserver&hl=en\_US&gl=US |
| **Client Side** | To be able to develop something that the client can download, we will need to develop an application that can be downloaded from the App store. Development will require engineers who are experienced with Swift and Xcode. There are costs to host your application in the app store and it must be approved by apple first. Currently Apple takes a 30% commission from their app store sales, which should be considered. Only one team would be necessary to develop a native application for Mac Os. | Developing applications for Linux can be very complex, because there are hundreds of different distributions of Linux. To make an application that will work on every distribution seems like it would be very difficult. There are also multiple app stores available to Linux users, so choosing which platform to distribute the application is important. You need engineers experienced with C and Visual Studio or Eclipse. Linux has a very small user base compared to Windows or Mac OS. Only one team would be necessary for the development of a native Linux application. | Windows has four different application platforms. We will need engineers with experience developing in Windows Visual Studio and using C++/.NET. Microsoft charges a fee for every application sold through the Microsoft store. If we do not host our application there, we will need to develop a way to sell keys that are required to download the application. Only one team would be needed to develop a windows application. | The cost of developing applications for mobile devices is very expensive. The average cost of an iOS app is around $28,000 and for Android about $23,000. You would need to develop both separately and would incur both of those costs. The average mobile app developer in the US makes a salary of about $107,000, so if long term app development is needed additional costs will need to be considered. It typically takes a developer 18 to 24 weeks to develop an application for just one mobile platform. You can offer your application to Android users without having it approved by Google, but for iOS it must be on the approved app store which costs money. Multiple teams will be needed in this case. |
| **Development Tools** | The language that is most common for the development of Mac applications is called Swift. Apple provides an IDE for writing Swift programs called Xcode which you can download directly from Apple. Also, Apple provides instructions for free on how to develop basic applications using Swift and Xcode. It appears that most of the development tools for mac are free. | The native programming language in Linux is C, so knowledge of C is extremely important for developing a Linux application. Eclipse and Visual Studio can be used on Linux and are great options for an IDE to use for those platforms. Eclipse is free but Visual Studio has a monthly subscription cost. | C++ is very highly supported on all Windows application platforms. Microsoft Visual Studio is an excellent IDE for C++ and are essentially the default language and IDE for Windows applications. Eclipse is another option for a good IDE for developing with C++ applications. Eclipse is free but Visual Studio has a monthly subscription cost. | For Android, C++ is a good option as Chrome was designed in C++ and it offers a lot of good tools for app development. Swift will be necessary to develop for iOS applications. Visual Studio or Eclipse can be used to develop Android applications and Xcode for iOS applications. |

## 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**: For an operating system to run the gaming room, we should select Linux as it would be the most cost-effective platform to run the server on. Linux is based on the Unix OS and has tons of resources available due to it being open source. It offers several security features for your server such as password protected access to certain areas of the system. We would have the ability to access files through a file hierarchy on the server and control access to those files. Linux supports Light-weight Directory Access Protocol, or LDAP, which is a protocol used to store and retrieve data quickly from servers that use an Active Directory. LDAP and ADP are critical components that offer security for usernames and passwords. Linux also offers support for Docker, which is a useful tool for testing and deploying updates efficiently to your application. Draw It or Lose It is not going to need an extremely powerful server, so it seems like Linux is the best operating platform to choose in this scenario.
2. **Operating Systems Architectures**: Unix has four main layers in its architecture. The lowest level is the computer hardware, which is needed for the operating system to run. The next level is the Kernel level, which is code that interacts directly with the PC hardware. Things the processor and RAM are controlled at the Kernel level which is mainly composed of drivers and configuration code. The next level up from the Kernel level is the Shell which contains system libraries. The shell is the interface between the user and the kernel level. This allows the user or programs to interact with code at the kernel level without needing the same level of access.  The outermost layer is the utility layer, where the programs that offer user functionality reside. This is where the web browser will be that is going to be displaying the Draw It or Lose It application to the user. The web browser will interact with the system libraries as needed once users access Draw It or Lose It from their web browser.
3. **Storage Management**: The storage management will be important for our program to be able to handle large amounts of users logging in from all over the world who will be attempting to save their data. We should store all usernames and passwords in a secure place located in the server directory. Our program uses REST API to authenticate users. They will have to login using their username and password to access their game, teams, and player data. The data will be accessed based on the name of the file that is stored. We should equip our servers with solid state drives since they are becoming increasingly inexpensive and have much faster read/write speeds than traditional hard drive disks. My current PC has an SSD that has been running windows 7 years and it is still going strong. I also had a hard drive disk that was only used for secondary storage go bad after only 2 years. The SSD (we may want more than one) will need to store the Linux OS *and* 200 8-megabyte images. This will only be about 27 GB of storage, but we also need to make sure we leave as much room as possible to store all the game data users will be creating. “Among Us” is an example of a small game created by only a few developers that exploded into popularity with millions of users buying the game and playing. We should plan for the eventuality that we are flooded with more players than we thought we would and leave room for upgrading our servers with more space. In the code, we will be using JAVA array lists to store the teams, players, and games. This will allow us to prevent the program from using more space than is needed because we can modify the size of the array list to prevent it from having empty elements. We can set up a Network File System, or NFS, to structure our servers if we end up having more than one server. This will make it so that one server can access files located on another.
4. **Memory Management**: Our application will use memory on the client side, and we should consider using memory efficiently so that more users are able to play the game. Most modern PCs being sold currently have somewhere between 8-16 GB of RAM (occasionally 4 GB for budget devices). The biggest thing we can do is to make sure that we only load one image into the client memory at a time. After an image is used, it should be removed from the active memory. There is no reason to have all 200 images loaded into active memory at the same time. We also should not be loading unneeded game, team, and player data into the client’s active memory. This would be a security risk and a waste of active memory. The application will load the image onto the screen at a fixed rapid rate, so the image being used for the current game will need to be loaded into active client-side memory from server storage. We should consider using Dynamic Loading which enables the system to only load the parts of the program into active memory that are being called. Many of the unused parts of the program would be left on the disk/drive until needed. This would be ideal as we need to make sure we are not loading all 200 images into active memory. Dynamic Loading is supported by Linux and all other platforms so we should be safe to use this method (Silberschatz, 2020).
5. **Distributed Systems and Networks**: We will be creating one of the most commonly used distributed network architectures which is called the Client-Server Architecture. The client will open a browser on their device, and type in a URL address to access the server (Silberschatz , 2020). Once authenticated, they are connected to the server where they can access certain files or programs on the server. After the server has completed the clients request, it communicates back to the client. The benefit of running a Client-Server architecture is that we can design our application so that the server only handles user data and does not actually perform any application logic. Most of the program is sent to the client and run client side. This will allow our server to handle a much larger load of clients. If our server were running the program for each client’s game, it could slow down quickly during peak times. Our servers will mainly be used as databases and will handle the database management. We may want to consider looking into backup servers located in different places around the world in case of a natural disaster or power outage. If one server goes down due to a local power outage, the other servers could pick up the slack and users would still be able to access the game. Using a distributed network allows us to have our hardware wherever we need if all are connected to the internet and are able to communicate with each other. The clients will be able to access our distributed network by connecting to the internet and navigating to a URL using a web browser. If we at least have one server up and running users should be able to access the distributed network from their device. As long as the user can download Google Chrome, which all operating platforms can, they will be able to play Draw It or Lose It.
6. **Security**: Our program utilizes REST API to ask the user for credentials as soon as they access the application. Once they enter their credentials, they will gain access to their game data. They will have restricted permissions, meaning they will not be able to access game data from other users with their login. All game, team, and player data will be stored on secure servers that can only be accessed by logging in through the rest API using valid credentials. This ensures that can be no unauthorized access to data unless someone’s username and password has been compromised. Our servers will need to have firewalls active to ensure that users are not able to transmit malware to the distributed network. Our program should include checks for user input to ensure users are not trying to enter anything other than their names or guesses in the games played. Direct server access should have a very strong username and password and it should be changed frequently (Plesky, 2020). Another step we can take to increase our security for our servers is disabling network ports that are not in use. With Linux, you can disable any open network ports on your server to prevent them from being accessed by hackers (Plesky, 2020).

References:

A. Shams, S. Böhm, P. Winzer and R. Dörner, "App Cost Estimation: Evaluating Agile Environments," 2019 IEEE 21st Conference on Business Informatics (CBI), Moscow, Russia, 2019, pp. 383-390, doi: 10.1109/CBI.2019.00050

Lastovetska, A. (2019, April 18). Understanding App Development Cost in 2020 [Full Guide]. Retrieved November 29, 2020, from https://mlsdev.com/blog/app-development-cost

DOGTIEV, A. (2020, August 28). How Much Does App Development Cost? Retrieved November 29, 2020, from <https://www.businessofapps.com/app-developers/research/app-development-cost/>

Team, E. (2020, October 20). Best Languages For App Development. Retrieved November 29, 2020, from <https://www.ecodelogic.com/best-languages-for-app-development/>

Operating System - Linux. (n.d.). Retrieved December 13, 2020, from <https://www.tutorialspoint.com/operating_system/os_linux.htm>

Plesky, E. (2020, August 07). Linux Server Security – Best Practices for 2020. Retrieved December 13, 2020, from <https://www.plesk.com/blog/various/linux-server-security-best-practices/>

Silberschatz, A., Galvin, P. B., & Gagne, G. (2020). Operating System Concepts (8th ed.). John Wiley & Sons. Retrieved from https://learning.oreilly.com/library/view/operating-system-concepts/9780470128725/silb\_9780470128725\_oeb\_tp\_r1.html