

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 | May 20, 2021 | Mark Rossmiller | Initial Revision |

**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)

Draw It or Lose It requires us to create a catalog of images and programmatically present them as clues to the game’s players. We will require the program to have the ability to create and manage multiple teams of players with multiple players on each team. There will be four rounds of play. Each team must have a unique name. And only one instance of the game can be in memory at any given time.

## [Design Constraints](#_2et92p0)

We will use Java language for programming a web-based application. Images should be compressed to a good amount to allow fast transfer over the web. Players should be able to log in from any location on the Internet, allowing for slow connections to be established for gaming.

## [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)

GameService was designed to be a Singleton class object, meaning there can be only one instance of the GameService class instantiated for the duration of program execution. It runs multiple Game instances. Each Game instance has a list of Team objects which are associated with that Game. Teams have a number of Player objects. Both Teams may be added to Games, and Players may be added to Teams. Players are identified by either a unique long or a human-readable String. Games and Teams are also identified by either a unique long or a String. Entity is an abstract class representing either a Game, a Team or a Player. ProgramDriver is the class containing the main() function for our program, and SingletonTester is a class used to test the validity of our GameServer’s Singleton status.

****

## [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 not a good platform for running server-side software. It is much more expensive than PC hardware and is developed mainly for its end-user experience. | Linux is ideal for hosting server-side software. The majority of web-hosting software is Linux-based as it is both cheap, secure, and also very robust. | Windows has a market for server-side hosting applications but is not as cheap or as secure as Linux-based systems. | Mobile devices are not ideal for running server-side software. They have unreliable connections and their hardware is ideal for end-user applications, not server software applications. |
| **Client Side** | Mac has a great end-user experience, and has a large share of the computer market, but is not the most commonly used operating system. I would recommend deferring development for Mac until the more commonly used operating system applications have been deployed. | Linux has come a long way in its development of the desktop environment, but still lacks the user base that Windows and mobile devices hold; I would consider delaying development of Linux software for the client side. | Windows is probably the most commonly used desktop operating system, as it comes with most new computers; if our application intends to run on a desktop/laptop system, I would recommend putting most of our initial development into this focused area. | Mobile devices have become the most common computing platform in recent years. Where I live, computers are very seldom used and most people are online via their mobile devices. I would recommend investing most of our development dollars into a client side mobile app. |
| **Development Tools** | Macs use mostly Objective-C and Swift for development of their apps. Command line tools are heavily used for Mac development. | Linux has a wide variety of development tools available for it, ranging from graphical IDEs to command line tools. Most modern languages have available tools on Linux for development. | Windows has its own official Microsoft development environment for coding, but there are also many open source tools available. Many different languages can be used. | Depending on the platform, either Swift for iOS or Java for Android. Both have development environments available to them. |

## 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 creating the server-side code on an Intel/AMD-based Linux machine, and developing a mobile app for the client-side program, specifically targeting ARM-based Android as its primary platform. Ideally, I like OpenBSD better than Linux as it has a more stable and bug-free architecture, with enhanced security, but it is much more difficult to find a suitable hosting environment that allows the installation of OpenBSD on its servers. Since we will be mainly just serving images over the web, a simplified installation of OpenBSD with its native httpd web server closes a lot of security holes that are native in both the Linux kernel and its operating system environment.
2. **Operating Systems Architectures**: Linux is the most widely-used server-side operating system, and Android is the most commonly used operating system for mobile devices, targeting end-users. Targeting these platforms makes the most sense. The server won’t be doing much besides hosting images and running DNS, so closing off all other services and ports is highly recommended in order to keep security holes to a minimum. Hosting on two separate service providers with two separate DNS records helps to minimize downtime if a server goes down. Mirroring the web server and its hosted image files is trivial.
3. **Storage Management**: Storage on the server-side should be done on disk, specifically our images should be highly-compressed for ease of transmission over unstable or slow connections. I recommend keeping storage on end-user devices to a minimum, to allow for a wider userbase. Since the images only take 1.6GB in storage, and almost every reasonably priced server comes with at least 4GB of storage, I recommend keeping the images on a tmpfs filesystem in RAM memory to keep access time down to a minimum. Of course, the bottleneck will not be access time but rather network bandwidth, so buying a server with increased network capacity should be a priority.
4. **Memory Management**: The app should only hold the images in memory on an as-needed basis. There is no reason to keep them in memory once they are not needed by the application. On the server side, they should be kept on disk and accesses only to serve the files to the client, unless we are planning on running very many game instances on our server, in which case we should cache oft-used images in memory. On the client side, I recommend a memory buffer of 8 images at most, and loading into the buffer starting from game instance launch and replacing images that have been cached as they expire after users have seen them.
5. **Distributed Systems and Networks**: Since we will be hosting our images over the web using an httpd server, there is little network programming to be done on the server side. On the client, we will need to issue http requests. There are libraries for such access in almost every sensible language choice, so this shouldn’t be an issue at all.
6. **Security**: While security is important, I don’t see it as essential for a drawing game. Of course we don’t want to leak personal information entered, especially contact information for young users which might break privacy laws in some countries. I suggest for a game to limit the kinds of information which can be entered into the strings containing user and team names. As already mentioned, on the server side we can install OpenBSD instead of the usual Linux server environment. The Linux kernel is loaded with tons of unaudited code, and most installations of Linux come preloaded with a lot of unnecessary programs which could cause security holes to be exploited. OpenBSD, on the other hand, has a very secure default installation, with very few programs and a very refined code base for the core operating system, and doesn’t allow easy commits of unaudited code like Linux and a lot of other open source projects do.