

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.1

## 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 | 03/19/2023 | Dino Suljic | This is the initial version of the document. Everything has been filled out and completed. |
| 1.1 | 04/02/2023 | Dino Suljic | Modifications have been added for Project 2 |

**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 wants to develop a web-based version of their Android application Draw it or Lose It. This new web-based application should work on multiple platforms. The company needs help streamlining the development process, which is the purpose of this document as it hopes to clarify and establish all potential design constraints as well as solutions to them.

## Requirements

* A game should support one or more teams
* Multiple players will be assigned to each team
* Both game and team names should be unique to each user
* Users should have the ability to check whether a name is already in use or not when choosing a team name
* Only a single instance of the game can exist in memory

## [Design Constraints](#_2et92p0)

* **Cross-platform support.** The game is an Android application that should have a web-based version of it. The new version should support multiple platforms and operating systems.
* **Image Copyrights.** The company wants to render images from a large library of stock drawings. We must make sure that the company has a license for each image and is allowed to use it.
* **Unique names for users.** Every user must have a unique name, which will allow users to check if a name is in use already or not.
* **Singleton game instance.** Only one instance of the game can exist in memory during any point.
* **Multiple users will be connected.** A game will have multiple teams with multiple players on each team. This will involve making sure that there is networking support for multiple clients.

## [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)

The UML diagram below will be used as a blueprint during development. The diagram allows us to see the game system in a clean and visual hierarchical format and should help streamline the development process. **Entity** is a base parent class which creates an inheritance relationship with multiple other classes. **Game**, **Team**, and **Player** are all child classes which inherit from Entity. These child classes, as such, have access to all attributes and methods in **Entity**. The goal here is to use **Entity** as a base and then extend its functionality, while avoiding violation of the **DRY** (don’t repeat yourself principle). It also has **id** and **name** attributes, which mean that games, teams, and players will all have ids and names. This adheres to software requirements, and values for both should be unique. Their respective getter functions can be called to aid with checking uniqueness.

In the top left, we can see a **ProgramDriver** class which holds the program’s main entry point. This is the class that will be called when the program starts. It has a relationship with **SingletonTester** which tests the code by ensuring that there’s only one instance of **GameService** in memory at all times. The **GameService** class can essentially be seen as the heart of the application as it manages all of the core functionality for the game. It fetches games, stores them in memory, and handles the team matchmaking. Its **getInstance()** function ensures that only one instance of **GameService** in memory at all times. This helps ensure that the software adheres to requirements.

The lines that connect each class together tells us that they have a relationship with each other. The entire application is interconnected as a whole by utilizing multiple OOP principles. The different arrowheads and symbols at the end of the lines denote different kinds of relationships, such as generalization or composition. The sold lines from the aforementioned child classes have a hollow, unfilled arrowhead that connects them with **Entity**. This shows an inheritance relationship. The numbers and symbols in the middle of each line quantify associations. For instance, 0..\* means “zero or more”. The plus (+) and minus (-) symbols before each attribute and method indicate visibility. A plus symbol means that the method or attribute is public and can be accessed from the outside, while a minus symbol means it can only be accessed internally as its marked as **private**.

This diagram should constantly be used during development as it helps ensure good design and that all software requirements are met.

**"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 works better as a user operating system, so it doesn’t have as much support for web-hosting. Its advantage is it supports Unix environments and has a great user interface. However, it is proprietary software and it isn’t as reliable for large programs. Licensing will cost, and there is less server support overall compared to Windows or Linux. | Linux is very good for web-hosting. It’s free, open source, and as such gives users full control. It’s light-weight, secure, quick, and highly customizable. Linux also tends to come with a lot of essential libraries and frameworks installed by default, including Python and Node. The only disadvantage might be its learning curve. | Windows works good for web-hosting and has operating systems dedicated just for this purpose. Microsoft also has support for the Azure Cloud and .NET. However, as it’s proprietary software, we would have to pay for the license. There also isn’t as much control and you’re forced into a Microsoft ecosystem. This is only worth considering if we want to develop in a Windows ecosystem. | Mobile web-hosting is not practical. Although it is possible with technologies such as Oracle’s Mobile Server, most developers have better experience with the other operating systems and environments. One advantage might be built-in support for mobile systems, but there will be less server-side support overall. |
| **Client Side** | Moderate costs for clients and developers as there will be licensing costs. Developers and expertise will be harder to come by. The simple user interface makes it easier to use, however. | Licensing will be cheap as it’s open source and free. A higher level of expertise is required as a lot of things have to be done through the terminal. | Moderate costs for licensing. Tons of support for both clients and developers. It is pretty easy to start using, and the developers won’t need as much expertise. | Moderate costs for licensing. Expertise is harder to come by, and this will be harder to implement compared to the other operating systems. |
| **Development Tools** | For the frontend, the languages in use will likely be a typical frontend stack: HTML, CSS, and JavaScript. TypeScript may be used.  Development tools might include what comes with Unix and Mac environments: Node, Python, and a decent text editor. Eclipse and PyCharm are also option as they are cross platform. | For the frontend, the languages in use will likely be a typical frontend stack: HTML, CSS, and JavaScript. TypeScript may be used.  The tools on the system will be what comes installed by default on every Linux system. This includes Node, Python, vim, PHP, Ruby, gcc, and more. There are great build tools too that come with all Linux systems, which makes compiling easy. Installing packages is also very easy. | For the frontend, the languages in use will likely be a typical frontend stack: HTML, CSS, and JavaScript. TypeScript may be used.  We might also consider .NET and Microsoft Azure.  Development tools might include Visual Studio Code, Microsoft’s compiler, and the .NET framework. It depends on if we want to develop in a Microsoft ecosystem or not. Gcc and other compilers are not available on here. | For the frontend, the languages in use will likely be a typical frontend stack: HTML, CSS, and JavaScript. TypeScript may be used.  However, we might also consider languages and IDEs more appropriate or mobile systems.This includes Java and an IDE more appropriate for mobile development. C++ is also a popular programming language in this area. We might consider Eclipse or PyCharm, or other cross platform options. |

## 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**: The recommended operating platform is Linux. It will also The Gaming Room to expand their application to the most environments in the simplest and cheapest way possible.
2. **Operating Systems Architectures**: A Unix environment can be installed on both Windows and Mac systems. It’s free and cheap, highly customizable, open source, secure, light weight, and it comes pre-installed with nearly every essential tool for development, especially for building a website.
3. **Storage Management**: A decent storage management system would be would be PostgreSQL or MySQL. Both have great support on Linux platforms. If we’re using Python for the backend, we would likely wanna use PostgreSQL as it has better support. Otherwise, we could consider MySQL.
4. **Memory Management**: Memory management for the application will be very lightweight. Linux will only run the software on one thread and process when we start, but we can easily scale it up and turn it into a multi-threaded application later on. It’s likely not going to use a lot of memory. There might also not be a graphical user interface as a lot of things are done through the terminal, so even less memory will be used.
5. **Distributed Systems and Networks**: We will want to use a database such as one of the aforementioned ones to be able to setup an authentication system. Data will be stored in the database as well as in memory during active sessions. In a distributed system, we will have one or more clients running on the same network. These clients will connect to each other, and we should handle the communication and take care of potential issues. If there is an outage, for example, it should simply disconnect one client and alert the other clients. We need to make sure to support potential networking issues such as this. The clients should have no dependency on each other, and should receive data from the server.
6. **Security**: Linux is considered very secure. Although there are some security tools installed by default, there is no antivirus suite like there is on platforms such as Windows. The user will have to be more careful and make sure to install signed packages only and avoid running something as an administrator if not necessary. In short, if the person using the operating system uses careful security practices, the operating system is one of the safest. User protection and security capabilities such as very high.