

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.2

## 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 6**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 01/22/2023 | Danielle Eeg | Initial information, design constraints, and UML detail |
| 1.1 | 02/04/2023 | Danielle Eeg | Operating system comparison |
| 1.2 | 01/19/2023 | Danielle Eeg | Recommendations |

**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 has requested an environment for their web-based game, Draw It or Lose It. The requirements can be addressed by using unique identifiers and singleton and iterator patterns to prevent the creation of duplicate players, teams, and games.

## Requirements

Requirements for the gaming environment include the ability for each game to have one or more teams and teams to have one or more players. Each team name must be unique and for each team, each player name must be unique. Furthermore, multiple instances of a game cannot exist at any time.

## [Design Constraints](#_2et92p0)

The design constraints for Draw It of Lose it are as follows:

*Each game can have multiple teams and each team can have multiple players.* This constraint means there must be a way to store instances of each team and player created. Team lists are unique to each game and players are unique to each team, so team instances should be stored by the game class and player instances should be stored by the team class.

*Each team name must be unique to its game instance and each player name must be unique to its team instance. Each game must also be unique.* This can be accomplished by creating unique IDs for each game, team, and player after verifying that a duplicate instance does not exist. To verify team names and player names are unique, an iterator pattern should be used to validate no such name has been used yet in the initialization of a team/player object.

*Functionality as a web application.*  The program should be programmed with a pre-specified operating system in mind. Whether it is hosted on a MacOS, Linux, Windows, or mobile device operating system, consideration will need to be taken as to how the application can be run securely and with appropriate graphics. The OS that hosts the game will help determine how memory is allocated as well as the cost and time required for programming.

## [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 program driver, ProgramDriver and the singleton tester class, SingletonTester are related through uni-directional association, were ProgramDriver uses SingletonTester to print each instance of the Game class created in GameService. This can be used to validate the Singleton pattern worked as only once instance of GameService should exist, and within this instance, each Game instance should have a unique ID.

The Entity class is a parent class where Game, Team, and Player classes inherit from it. This reduces the amount of duplicate code that would be required in the three child classes since they all share the id and name attributes, as well as accessor methods for each. Each of the child classes are associated with each other: Game has a zero or more multiplicity with Team, and Team has a zero or more multiplicity with Player. The relationship is also shown through the private list attributes in the Game and Team classes. Game stores a list of Team instances and Team stores a list of Player instances. The minimum number of items that can occur in a list is zero, but more can be added (supporting the zero or more-multiplicity relationship).

The GameService class also has a zero or more multiplicity with Game, although GameService does not inherit from Entity. GameService is where the singleton pattern will occur since it stores and adds new instances of the Game class. When an instance of GameService is created, it will be stored as the attribute, *service*, and then that instance can be retrieved through the accessor method for service, *getInstance().* SingletonTester and ProgramDriver will interact with the GameService class to retrieve the existing list of game instances instead of potentially creating multiple sets of games (which would interfere with the program requirements). GameService also oversees creating new Game, Team, and Player IDs when instances are created. This is because only a single instance of GameService can be created, so IDs should not be created by any of the classes Game, Team, or Player, which can have multiple instances.

**"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** | MacOS is a good option if all clients use Mac products. MacOS servers are straightforward to use and can easily increase processing power. A major downside of MacOS servers is it does not integrate well with other operating systems. | Linux servers are open source and extremely secure relative to other options. A con is that there are a lot of distributions, so not all have adequate support. | Windows servers integrate well with other operating systems and a single server can be supported by multiple processors. A downside is that is not open source and has higher risk of viral infection than other servers. | An android mobile phone can be used as a web server, requiring minimal processing power. Additionally using an Android device can be inexpensive. However, mobile device-based servers can have poor security. |
| **Client Side** | MacOS is more expensive up-front for the client, although they have high security and its hardware is well-made, which can lead to lower costs long-term. One tricky thing with MacOS is that it has a smaller userbase than Windows PCs and it does not integrate well with non-MacOS applications, which can lead to high maintenance requirements whenever updates are needed. | Linux is cheap to maintain as it is free and open source. Linux has a very low client base, which means that it can require a higher level of expertise and more time investment to develop and keep Linux web apps running. | Windows has the largest client-base in comparison with MacOS and Linux. It does come with some cost, but it integrates well with third party applications and its large user base means there are a lot of resources available to create and support Windows web applications. This leads to low expertise requirements and less time investment than a Linux may need. | When developing a web app for a mobile application, the application should be compatible with iOS and Android systems, which may require significant expertise and more time invested since most developers specialize in one or the other. |
| **Development Tools** | MacOS web applications are generally programmed using Objective-C. In my research, I have found Visual Studio as the top recommended IDE for Objective-C development. Visual Studio is free to use across all operating systems. | Linux web app development can be done with many languages, but C and C++ or Java are most common. Once again, I would select Visual Studio as my IDE. | When coding on Windows OS, C, C#, and C++ are commonly used languages. Visual studio would be a preferable IDE. | For a mobile application, the programming language would depend on the phone OS. Swift, Java, and Objective-C would likely be used because it can integrate with either system |

## Recommendations

Analyze the characteristics of and techniques specific to various systems architectures and make a recommendation to The Gaming Room. Specifically, address the following:

**Operating Platform**:

For an operating platform, I recommend building the server on a Linux machine. Not only is Linux free and open sources, but it is also highly secure when compared to other operating systems. Linux servers are very common and can integrate well with client-side machines that have different operating systems. Android mobile phones, in face, have operating systems that are rooted in a Linux kernel. If the distribution of Linux used has adequate support and documentation, this will be the best route to take.

**Operating Systems Architectures**:

Linux architecture is made up of four main parts: the hardware, kernel, shell, and utilities/applications. The hardware of a Linux system is exactly as it sounds; it is comprised of all physical assets required to build an operating system including the CPU, RAM, storage, and terminals for interacting with other hardware devices, such as USB ports. The kernel is the foundation of the operating system. It interacts with both the hardware and software layers, managing the transfer of information between the two using abstraction to improve system efficiency and security. The shell is used for direct communication between end users and the kernels. Linux applications and utilities are software units in the system that ultimately provide practical applications for the hardware, kernel, and shell. This software includes things like settings and device managers that can manage system settings and updates, as well as web browsers, games, and so on.

**Storage Management**:

For storage management, I recommend a solid state drive. Accessing data stored on a solid state drive is much quicker than on disk drives because of their page structure. Hard disk drives require mechanical movement and are more likely to break and cause system overheating than solid state drives. Although solid state drives cost more than hard disk drives per unit of storage, I would recommend going that route for the sake of efficiency and maintenance.

**Memory Management**:

Linux systems utilize virtual memory to remove the limits placed on program size by physical memory. This will be useful for improving performance of the Draw It or Lose It by separating logical memory of a program from physical memory. Using demand paging through virtual memory, the program will avoid wasting physical memory space on program files that are not currently in use. For example, say 3 games exist for Draw It or Lose It, each with 3 teams and 4 players on each team. If only one game is in-use, that game will be accessed from physical memory, but the other two games, their teams and players will be left in virtual memory until they are in use.

**Distributed Systems and Networks**:

For sharing data between different platforms, a database should be managed by the server. By using a REST API, all operating systems (end user and server) will be able to access and update data as needed throughout gameplay. This will also promote secure connections between the clients and the database, as no user login information is stored in REST APIs. If at any point the client is disconnected from internet or there is an outage, the database will not be able to be accessed. This is one downside of using an API. Scheduled maintenance can take place for the REST API to keep it running smoothly over long periods of time.

**Security**:

Having a Linux server will be inherently most secure, in comparison with Windows and MacOS. Beyond the security built into the structure of the kernel, there are some ways to make all aspects of the Draw It or Lose It application more secure. End users may be on MacOS, Windows, Linux, or mobile machines. I recommend managing application security by requiring end users to set up a username and password the first time they play. The password should be at least eight characters long and contain at least one number, one lowercase letter, one uppercase letter, and one special character. Each time they start a new session, they will receive a one-time password via either text or email.