

Draw It or Lose It Online

# **CS 230 Project Software Design Template**

Version 3.0

## 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 |
| --- | --- | --- | --- |
| 3.0 | 10/14/2025 | Nick Vulcano | Updated the “Recommendations” section |

## [Executive Summary](#_sbfa50wo7nsh)

The client The Gaming Room does not know how to setup the programming environment for their web-based game, which was based off their current mobile game Draw It or Lose It. They have asked CTS to step in and assist with streamlining the development. The application will need to be able to render images on the screen that will then be guessed on by each team. These images will be pulled from a large library of stock images.

## Requirements

* A game will have the ability to have one or more teams involved.
* Each team will have multiple players assigned to it.
* Game and team names must be unique to allow users to check whether a name is in use when choosing a team name.
* Only one instance of the game can exist in memory at any given time.

## [Design Constraints](#_2et92p0)

* Technical Constraint – Only on instance of the game should exist in memory.
  + This will require a centralized state management like a Singleton Design Pattern. This also mean synchronizing across multiple servers in a distributed environment.
* Technical Constraint – Unique game and team names
  + During the creation of the team’s name there needs to be some logic to check against a database to make sure the name is not in use. Requires real-time validation during creation.
* Technical Constraint – Multiple teams and multiple player support
  + Requires relational data models from Team to Players to be mapped to ensure multiple players are allowed in teams.
* Technical Constraint – Game can have one or more teams involved.
  + Requires relational data models from Game to Teams to be mapped to ensure multiple Teams are allowed in a Game.
* Technical Constraint – Must run on all types of browsers
  + Requires responsive design and cross-browser compatibility

## [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 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.**

* ProgramDriver contains the main() function where the actual code is executed.
* SingletonTester test that only one instance of a unique game is running at a time, but it is dependent on testing ProgramDriver class, which is one of the requirements.
* Game, Team, and Player class all inherit from the Entity superclass. These classes come with an id tied to the name, to ensure there is always only one unique name at any given time.
* Team class can have zero to many instances of unique Players within a game listed in it.
* Game class can have zero to many instances of unique Teams within a game listed in it.
* GameService class can have zero to many instances of unique Games listed in it.

## 

## [Evaluation](#_2o15spng8stw)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Using tools like Apache, Mac OS can host web-based applications but due to licensing costs and limited server support it’s not very common. | Linux is the gold standard when it comes to hosting web-based applications due to its open-source nature and support. Its scalable and stable. Very cost effective. | .NET based environments are common for web-based apps but can have high cost related to licensing and is harder to maintain compared to Linux. | Mobile devices are not very suitable for server side due to limited processing power, storage and battery life. They are mor suitable for Client side. |
| **Client Side** | There are many proprietary requirements to consider when developing on MacOS’s Safari. Development on MacOS hardware can be costly but languages like HTML5, and JS are very cross-platform. | Cost effective developing for browsers such as Chrome or Firefox. Testing across Linux distributions can be time consuming, unless you are familiar with open-source tools. | Developing on Edge or Chrome is very common and widely supported. Development is straight-forward but testing must account for many different versions and hardware configurations. | Requires testing across both iOS and Android which can be costly and time consuming. Many hardware and software configuration to consider due to many different mobile devices. |
| **Development Tools** | MacOS supports IDEs like Xcode, Visual Studio Code, and JetBrains. Languages include Swift (for native apps), JavaScript, Python, and Java. Xcode is native for MacOS but has a steep learning curve and is only available on MacOS systems. | Linux supports many IDEs including Eclipse. Languages include Python, Java, and C++. Using Eclipse is a very useful and intuitive IDE but can be a resource hog on older systems. | IDEs such as Visual Studios and JetBrains are most common here. Languages include C#, JavaScript, Python and Java. Visual Studio is a full-featured IDE for .NET but is very resources intense and has long installation times. | Mobile development uses Android Studio and Xcode for iOS. Java & Kotlin for Android, and Swift & Objective C for iOS. Coding in Swift is a fast and modern language, but it is limited to Apple platforms and would then have to rewrite the program in Java for example. |

## Recommendations

1. **Operating Platform**: Linux is a great choice for your game based on your needs. Its open source so its cost effective and works great for multiple different environments. It is scalable and very stable under load and supports multiple different languages. I think it would be a perfect fit for your game and budget.
2. **Operating Systems Architectures**: Linux uses a monolithic kernel that packages server calls, device drivers, and file system management all in one large block of code running in a single address space. It is a modular design that supports concurrent users and processes, which fits your multiplayer game requirement.
3. **Storage Management**: The recommended storage management for your program would be an XFS file system with a Logical Volume Manager. The XFS file system is excellent for large files and has high performance for Inputs and Outputs. The LVM allows for dynamic resizing of storage volumes for better space management. Since the number of games and players is unknown at any given time, having a storage management system that can dynamically resize is nice to have.
4. **Memory Management**: There are some nice techniques that the Linux OS utilizes. The virtual memory uses paging and segmentation to provide each process with its own address space. It also uses shared memory that enables efficient communication between certain processes, this will be useful for multiplayer synchronization. There is also a feature called OOM Killer, which will automatically terminate processes when memory is critically low to maintain a stable system.
5. **Distributed Systems and Networks**: To enable cross-platform communication the game needs to break apart its leaderboards, authentication, and logic. Using something like a RESTful API can help keep all these microservices separated and able to communicate over a network. It can utilize Web Sockets to synchronize real-time gameplay.

1. **Security**: The most basic form of security can come from input validation. This will thwart attacks against buffer overflows and injection attacks. Role-based access control (RBAC) within the RESTful API will help keep access to the user’s limited to only what they need to play the game. Linux also has built-in security features such as AppArmor, SELinux, and iptables. These added security features further reinforce our recommendation to use Linux as the operating system of choice for this application.