

The Gaming Room

# **CS 230 Project Software Design Template**

Version 3.0

## 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 | 03/31/2022 | Caleb VanDerMaas | Initial Design |

**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 staff would like to create a web-based game application called Draw It or Lose It. The goal of the game is to guess a picture as it’s being rendered by the computer. If the player/team doesn’t guess the word within 30 seconds, then the other player/team gets another 15 seconds to guess. The key requirements include the ability to have one or more teams involved, allow each team to have multiple players, unique game and team names, as well as only one existing instance of the game in memory at any given time.

## [Design Constraints](#_2et92p0)

The system time will be a large factor of this game. The team component needs to be flexible and expandable. Modularity for the player and team components will be key to creating this application. Finally, the game instance itself must be limited to only one instance at a time. Therefore, this should be the main driver of all the gameplay and must not be duplicatable while in use elsewhere.

## [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)

This UML class diagram relates directly to the design constraints posed above. Game service sets the stage for the game to be played. It gathers the information about the teams and the players on the teams so that a game can run with that information. The game, team, and player are all modular so that they can be added and edited as needed. This way, a player can select a game, add teams & players to that game, play that game, or create another game and do the same thing. These object-oriented classes allow for the setup to be very flexible. The entity then pulls from these classes to create a running game instance. Since there can only be one game running at a time, the subclasses all feed directly to the entity class where they are used .

**"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** | Hosting a web application on most platforms should be easy. However, hosting a web-based native application becomes challenging with MacOS system architecture running on ARM. | Linux is over all good for running just about anything | Most web-based applications are build for Windows systems. There is not a whole lot you can’t do for Windows with server side. The only problem is that it won’t run any mac applications. But this typically isn’t a problem because windows usually has a sufficient replacement. | Mobile is a different kind of beast. Although, server side dev requirements might be a bit easier than the client side as there are many different write once, deploy everywhere mechanisms that are becoming popular today. |
| **Client Side** | Mac seems to have a very specific client usually geared towards music/video production. Mac has never had much support for gaming and therefore it’s not the best idea to spend the time and money for this case. | Linux has one of the smallest followings of all the listed OS. It would be silly to develop for only Linux. Instead, it’s easy to find a language that Linux supports and easy to avoid the languages it doesn’t | Windows has the most users out of any OS hands down. So, it’s a no brainer to develop most apps to run on Windows. They also make it very easy to do so. | The sheer number of different app stores is a very daunting reality. Each one of these Appstores would need their own version of your app that is compatible with your server’s program. |
| **Development Tools** | Mac lacks support for many mainstream languages and IDE’s for them. This is mostly due to the systems architecture but also because the OS is not as widely popular as Windows. The biggest downside to Mac is that if you want to develop for Apple then you must own an Apple (or at least work on one) | Linux is the most versatile for programming and testing. Although it’s not always the most user friendly, it is very good at what it does. | Windows is a very popular and mature OS that has had a lot of support over the years. This means that the tools are very optimized and efficient for developers making apps to serve Windows. | Luckily, there are many good IDE’s to develop games for mobile devices. However, if you want to build for Apple, you must own an apple. |

## 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**: Starting with a Window’s based operating platform will give you the most access to the biggest audience, also one that is more geared toward gaming than the rest. Although it’s difficult to find an exact number, average estimates would show that Windows dominates more than 75% of the total laptop and desktop OS market share with MacOS making a not so close second. This is important to consider because it’s likely that this game will be played on the Windows operating system more than the Mac operating system. It also isn’t uncommon for indie or even triple A games to exclude support for the Mac operating system. This isn’t to say that we should just completely skip over this operating system as a whole. But rather, we should start by building the application to work on Windows first so that we can deliver for that platform. Then, if there is enough demand for a MacOS native application, then resources can be dedicated to making that happen. I believe that this is the most logical and cost-effective approach in terms of choosing and operating system to develop for.
2. **Operating Systems Architectures**: Similarly, it would be the easiest to develop for a Window’s OS architecture not only because it would match the operating platform and be easier to test and develop, but also because it would serve the largest market first. Also, it is likely that designing a game for the windows platform will result in a decent blueprint for development on other platforms. As long as the development team implements good SOLID practices and writes with a modern, robust language, it should be easy enough to then translate that code into workable software for another operating system. Developing for market and industry leading technology companies is almost always the right choice because they are the trend setters and adhering to their methods will likely future-proof your program to an extent.
3. **Storage Management**: The application should manage it’s own storage on the hardware by auto-provisioning ant automatic volume migration. The application should also continuously take snapshots of the entity at regular intervals in case the application fails unexpectedly. This would keep a save state that the users could return to in case of an event like this. However, it would be very inefficient to hold on to ALL the snapshots forever, so a smart recycling system should be integrated which would drop save states from memory and storage as needed. However, these systems should be robust enough to stay lean while guaranteeing an appropriate save state for the user. Past this, it is important to condense omit where possible. Space saving actions like compressing the images and minifying code can greatly reduce the footprint of the final application. There is also the choice of storing resources and assets in the cloud, however this may be an unnecessary and expensive choice, especially for such a simple application such as gaming room. To keep the game playable forever, you could offer expansion packs and new material in the updates to the application, this way the game would not become repetitive and boring.
4. **Memory Management**: Similarly, memory management should be automatic. There should be set limits in case a bug slips through to not disrupt the OS and there should be a smart garbage collection system that keeps the memory use minimal for optimal performance. This is important to not bog down a user’s system. I know I would personally be very upset if a simple video game I accidentally left running in the background consumed a large portion of my device’s resources. Since the average user is not typically aware of this possibility, it is up to the developers to take this as a personal responsibility to the consumer. Good memory management should make the game very snappy and enjoyable to play while also sticking to a minimum.
5. **Distributed Systems and Networks**:

There are many tools available today to develop applications that can work across multiple operating systems. One of the biggest advantages of more modern technology is cloud computing. Services like Amazon Web Services (AWS) allow you to host your application nation (and even world)-wide. They offer decentralization so that if one server ever goes out, your game will still be available to play. They also offer faster, more powerful computing power which can be very valuable to a server-side game.

1. **Security**: Encryption is the key. Also, by storing as little of the user’s data as possible, the risks are mitigated up front. Implementing anti-virus software and a threat detection system is a great preventative measure and having a backup on hand is always a good idea just in case your application needs a wipe and refresh. These are all systems that can be put in place to protect the user and the entire application.