

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.0

## 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 |
| --- | --- | --- | --- |
| 1.0 | 03/15/2025 | Benjamin Roberson | Commit changes to Executive Summary, Design Constraints, and Domain Model. |
| 1.1 | 4/4/20205 | Benjamin Roberosn | Commit changes to Evaluation. |
| 1.2 | 4/18/2025 | Benjamin Roberson | Added information to “Recommendations” section. |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room wants to create a web-based version of “Draw It or Lost It”. Only one instance of the game can be running at a time. Each team will have multiple players. Teams and player will have unique names. To make this work, the singleton and iterator patterns will be used.

## Requirements

N/A

## [Design Constraints](#_2et92p0)

Design constraints include the singleton design pattern, iterator pattern, unique identifiers, and scalability. The singleton patter needs to be used to make sure only one instance of the game is running at any given time. This will prevent multiple games being open at the same time and will be a part of the GameService class. The unique identifiers will be the team names and the player names. The iterator pattern will have to use the unique identifiers to ensure that each team name is unique, and each player name is unique. This will help with existing game retrieval. This will be used in multiple methods. The scalability of this program is dependent on object-oriented programing. For the game to grow in the future, the architecture must be easily changed and adaptable for future design and feature choices.

## [System Architecture View](#_ilbxbyevv6b6)

N/A

## [Domain Model](#_8h2ehzxfam4o)

The UML Diagram shows each class and their relationship with one another. It describes each class and gives methods and attributes that will be used in each class. Entity class holds common attributes of many other classes like id and name. Game, Team, and Player classes all inherit attributes from Entity class. Game class contains the active game session and its teams. The Team class contains the information for each team within the game. This includes its players. The Player class holds the methods and attributes for each player within the teams. The GameService class works on a singleton pattern and ensures only one instance of the game is open at any given time. Each class that uses attributes from the Entity class is an example of Inheritance. Many classes have private variables that are accessed with setters and getters. This shows the use of encapsulation.

**"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)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | MacOS uses Unix. This helps with great command line tools and working with Java. Its smaller, and costs are high making it less used. | Linux has a large community with good support. Also has great support for Java servers. It is harder to learn though compared to Windows or MacOS. | Better for large companies. It works well for databases and management. It is more expensive than Linux and less expensive than MacOS. | These are difficult for scalability and performance. They are often less powerful, thus needing something like a cloud environment for better operation. |
| **Client Side** | Java has good support in MacOS. The cost is about average and the time to produce is as well. There are things specific to MacOS that developers need to know to make a good user experience. | Linux dev tools are often open-source, development is cheaper than the other two. The time and difficulty are a little higher due to how custom it can be. | The cost and the time required are about average. Most java libraries are compatible. Need to know file paths and any windows specific features. One thing to note is terminal commands are different than Linux/Unix. | If developing for IOS and Android, the time and cost is higher due to working, in a way, two projects. Need to worry about being able to be in the app stores and working with different hardware too. |
| **Development Tools** | Java works well on MacOS, most major languages do too. IDE’s like VS Code, IntelliJ IDEA, and Eclipse work well. MacOS also has Terminal, Homebrew, and Git to help out.  MacOS takes 30% of revenue from apps. This is a large chunk of costs to think about. | Same tools that work on MacOS work on Linux. GCC, Cradle, Make, Git, and Maven are great for your builds and compiling. The terminal when learned is also strong.  Most costs are going to come from things like cloud services. | There are some great GUI frameworks for it e.g.: JavaFX, Swing. Most IDEs work well with Windows, so take your pick on those. Windows also has a great file system. Windows store takes 30% of revenue from app sales. This is less than the in app purchases of MacOS. | Android mostly uses Android Studio. IOS mostly uses Xcode. Developing in Java works on Android. IOS uses swift and Objective-C the most. There are frameworks that allow you to work on both at the same time. React is a good example. IOS is the same as MacOS in cost. Android charges $25 for app submission and some dev tools have costs. |

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**: For the operating platform, I would recommend Linux. It has a small footprint in terms of resource requirements. It is open sources and stable for expanding the game to be compatible across multiple platforms.
2. **Operating Systems Architectures**: Linux’s Ubuntu server runs in a way that allows direct communication between the system and the kernel level. It supports many technologies that allow developers to easily manage the dependencies for all the different consoles and computer systems.
3. **Storage Management**: Cloud storage is the best option. There is less that is needed to be stored on the system hardware making the game more available to more people. For this I would recommend AWS. It has tons of features to help keep the game running smooth. It also has the ability to help the game scale as it reaches more users.
4. **Memory Management**: Ubuntu uses something called demand paging memory management. This allows parts of the game to be loaded into RAM only when it is absolutely needed. This allows for smoother operation and less required resources to run the game. It also uses swap space similar to how Apple uses swap for there RAM and storage to be able to handle heavier loads.
5. **Distributed Systems and Networks**: To make this work well across the different platforms, something like RESTful APIs would be good for instant interactions. This way communication can happen no matter the platform. Each platform will interact with the backend through HTTP requests.
6. **Security**: using HTTP for data transmissions will help keep user information secure. Google OAuth 2.0 will be great for user authentication. Encrypting things like passwords will help protect users as well. Role-based access will allow users to only gain access to what they absolutely need for what they are currently doing and are allowed to do. For mobile devices, they each have their own safe ways for local password storage. Keychain for IOS and Keystore for Android. Frequently checking and patching security risks will be an ongoing thing that will help keep users and their information safe as well.