

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 | 09/21/2025 | Ajay Colby | Initial draft of software design document |

**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 take their game *Draw It or Lose It*, which is currently only on Android, and bring it to the web so it works on multiple platforms. The main problem is figuring out how to design the software so that it can keep track of games, teams, and players while making sure the rules are followed

My solution is to use a base entity class that stores the ID and name, since every game, team, and player needs those. Then I created Game, Team, and Player classes that extend from Entity. A singleton GameService class makes sure that only one game can exist in memory at a time, which is part of the requirements. I also used the Iterator pattern when adding or finding games, teams, and players so that names stay unique. This design keeps thing simple, meets all of the clients needs, and sets the project up to be expanded later into a full web application.

## Requirements

*<* Please note: While this section is not being assessed, it will support your outline of the design constraints below. *In your summary, identify each of the client’s business and technical requirements in a clear and concise manner.>*

## [Design Constraints](#_2et92p0)

There are a few limits we need to work with when making this project. The most important one is that only one game can be active in memory at a time, and the code enforces that. Another rule is that names have to be unique for games, teams, and players. That way, people can’t accidentally create duplicates.

Since this will eventually be a web-based game, I had to think about what happens if multiple people are using it at once. The Singleton helps with that, and I used synchronized methods and ID counters so the program doesn’t mess up if two things happen at the same time. For now, the program just loops through lists to check for duplicates, which works fine for a small prototype. In the future, when there are more users, a database or something similar would make it faster. The good part is that the design is written in plain Java, so it can run on different platforms and be hooked up to a web service later.

## [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 domain model has four classes. Entity is the base class, and it holds the ID and name that every object needs. Game extends from Entity and holds one or more teams. Team also extends from Entity, belongs to a game, and has players. Finally, Player extends from Entity and belongs to a team.

The relationships are straightforward. A game has teams, and a team has players. Each team knows which game it belongs to, and each player knows which team they are part of.

Some object-oriented programming ideas are shown here too. Abstraction and encapsulation are used because the fields are private and accessed with getters and setters. Inheritance is used since Game, Team, and Player all come from Entity. Polymorphism shows up when all of them share the same toString() method style.

On top of that, the Singleton pattern is used in GameService so only one service object exists. The Iterator pattern is used to loop through collections to find names and prevent duplicates. This design makes sure the game follows all the rules: only one game at a time, multiple teams, multiple players, and unique names for each.

**"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 can technically host a web-based application, but it isn’t commonly used for large-scale server hosting. The main downside is that Apple hardware is expensive, and support for enterprise-level web hosting on macOS is limited compared to Linux or Windows. It does have a Unix foundation, so it’s secure and reliable, but overall it’s not cost-effective for hosting at scale. | Linux is the most popular option for hosting web-based applications. It’s secure, stable, flexible, and open-source, which means no licensing costs. Most large-scale servers use Linux because it handles heavy traffic and scaling very well. This makes it the most practical choice for The Gaming Room’s server needs. | Windows Server is another option for hosting, and it has a user-friendly interface with strong integration for Microsoft technologies. However, licensing costs can add up quickly, especially for scaling. It’s secure and reliable, but not as cost-efficient as Linux. | Mobile devices don’t host web applications themselves, but the server needs to handle thousands of mobile requests at the same time. That means scaling, load balancing, and stable connections are essential. |
| **Client Side** | Developing for Mac desktops mainly means making sure the app runs smoothly in Safari, as that’s Apple’s default browser. Since this project is browser-based, costs stay lower compared to creating a separate native app, but extra testing is needed for Safari-specific quirks. | On desktops, Linux users mainly rely on browsers like Firefox or Chrome. Making the game run in these browsers doesn’t add much cost or time, but testing is important because some Linux systems are customized. Expertise is needed to make sure compatibility works across different distributions. | On desktops, Windows users are the largest group, so supporting them is critical. The app has to work well in browsers like Chrome, Edge, and Firefox. This means more testing, but the broad user base makes it worth it. | Supporting iOS and Android is crucial. Since the app will run in a browser, it needs to be fully responsive and work across different screen sizes. Testing adds time and cost because there are so many device types. Developers also need expertise in mobile compatibility and performance optimization. |
| **Development Tools** | The most common tool is Xcode, which is free but limited to macOS machines. Developers working on Mac client testing need Apple hardware, which increases costs for the team. Programming languages like Swift or Objective-C are common for native Mac apps, but for this project, web technologies like HTML, CSS, and JavaScript are the focus. | Developers can use tools like Eclipse, IntelliJ IDEA, or Visual Studio Code, which all work on Linux and are mostly free. Programming languages like Java, Python, and JavaScript are well supported. This platform gives the team flexibility without extra licensing fees. | Visual Studio is the most common IDE on Windows, and while there’s a free Community edition, the Professional versions require licensing. Programming languages like C#, Java, and JavaScript are often used here. Since most developers are already comfortable with Windows tools, training costs are low. | Android Studio (for Android) and Xcode (for iOS) are common native tools, but since this project uses web technologies, tools like React Native or Flutter could help for mobile responsiveness. JavaScript frameworks like React and Angular are also valuable. Licensing is not a big issue here, but testing devices and environments can get expensive. |

## 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 Draw It or Lose It game, I recommend using a Linux-based server platform hosted in the cloud, like AWS or Azure. Linux is reliable, cost effective, and it works well in almost every environment. Using cloud servers also gives the flexibility to scale as more players join, without having to buy new physical hardware.

1. **Operating Systems Architectures**:

Linux uses a monolithic kernel design, which means the important system services run together in the core of the operating system. This makes it fast and efficient for handling a lot of requests at once, which is exactly what we need for a multiplayer game. Another big benefit is that Linux supports containers like Docker, so we can run the game in separate, isolated environments that are easy to manage and scale.

1. **Storage Management**:

For storage, I recommend a distributed file system, like Amazon Elastic File System or Azure Files. This allows multiple servers to share the same game data, so no matter which server a player connects to, their progress and information are always available. Linux also supports file systems like ext4 and XFS, which are stable and widely used. Cloud storage also comes with backups and replication built in, which protects the data in case of a failure.

1. **Memory Management**:

Linux manages memory through virtual memory, which divides memory into smaller pages. It uses techniques like paging and swapping to keep active processes running smoothly while moving less-used data out of the way. This means the game can handle lots of users at the same time without slowing down. Memory isolation also makes sure one program doesn’t interfere with another, which adds stability and security to the system.

1. **Distributed Systems and Networks**:

Since the client wants the game to work across multiple platforms, it will need to run as a distributed system. The main game logic will run on central servers, while users connect from their devices over the internet. A load balancer will spread traffic evenly across servers so no single one gets overwhelmed. If one server goes down, others will keep the game running. Cloud networks also offer redundant connections so the game keeps working smoothly even if part of the network has an outage.

1. **Security**:

Security is one of the most important parts of this design. All data will be encrypted when it travels across the network using HTTPS, and stored data will be encrypted on the server. Passwords will be protected using hashing, and user accounts will have role-based permissions so only the right people can access certain features. The system will also use firewalls and monitoring to block intrusions. Since this will run in the cloud, it also benefits from the provider’s built-in security and compliance standards.

## Sources:

LMSBoss, C., & LMSBoss, C. (2024, October 15). *Azure vs AWS Reliability: Most Reliable Cloud Platform [2025] - CCSLA Learning Academy*. CCSLA Learning Academy. https://www.ccslearningacademy.com/azure-vs-aws-reliability/

Susnjara, S., & Smalley, I. (2025, July 22). Docker. *What is Docker?* https://www.ibm.com/think/topics/docker

*Is ext4 and xfs only for usage with internal file systems?* (2024, September 23). Unix & Linux Stack Exchange. https://unix.stackexchange.com/questions/783867/is-ext4-and-xfs-only-for-usage-with-internal-file-systems

*What is load balancing? | How load balancers work | Cloudflare*. (n.d.). https://www.cloudflare.com/learning/performance/what-is-load-balancing/