

# 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 | <10/19/25> | Sulav U | Added 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)

Simply put, you want to bring your fun Android game, "Draw It or Lose It," to a wider audience on the web so it can be played on multiple platforms. Our plan is to build a strong, flexible foundation for the game that can grow with you.

Think of it like this: we'll create a central game hub that lives on the internet. This hub will run all the game logic and keep track of every game, team, and player. To make sure everything runs smoothly, we'll use a couple of smart programming strategies. One key strategy ensures there's only one master copy of the game service running at any time. This is crucial for keeping all the game data consistent and preventing chaos. We'll also use an efficient method to search for things like game or team names to make sure every name is unique, just as you requested. This approach will give you a solid, reliable, and future-proof version of your game.

## 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)

As we move your game to a web-based environment where people can play together online, there are a few important challenges we need to plan for.

* **Making it work everywhere**: Your players will be using all sorts of devices and browsers—Windows, Macs, phones, you name it. We need to make sure the game looks great and works perfectly on all of them. This means building it to be flexible and adaptable from the start.
* **Beating the lag**: In a timed game like yours, even a tiny delay can be frustrating. We have to make the communication between a player's screen and our central game hub as fast and light as possible to ensure the gameplay is snappy and responsive.
* **Being ready for success**: What happens when your game becomes a huge hit? We need to build the system so it can handle hundreds or even thousands of players at once without slowing down. It has to be able to scale up smoothly as your player base grows.
* **Keeping things safe and fair**: Security is a top priority. We have to protect your players' information and build the game in a way that prevents cheating, ensuring a fair and fun experience for everyone.

## [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 diagram below is essentially the blueprint for our code. It shows all the moving parts of the game and how they fit together. Let's break it down.
* Building with reusable blocks (Inheritance): You'll notice a base block called Entity. It's a simple template that just holds a unique ID and a name. Instead of re-writing that code for every part of the game, we use this Entity block as a starting point for our Game, Team, and Player objects. This is a core programming principle that keeps our code clean, efficient, and easy to update.
* A place for everything (Association): The blueprint organizes everything into a neat structure, like nesting dolls. The main
* GameService holds a list of all active Games. Each
* Keeping things tidy (Encapsulation): Each piece of our blueprint is designed to be self-contained. It keeps its own information private and only allows other parts of the program to interact with it in specific, controlled ways. This is like having different departments in a company; each one manages its own tasks, which prevents mistakes and makes the whole system much more reliable.
* The blueprint also includes a key design pattern to meet your requirements: The Singleton. We've designed the
* GameService class this way, meaning it's physically impossible for the program to create more than one instance of it. The

**"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** | **Characteristics:** macOS Server is built on a stable, secure UNIX foundation. **Advantages:** It offers a user-friendly interface. **Weaknesses:** It has a very high cost due to Apple's hardware requirements and a small market share, leading to less community support for server hosting. | **Characteristics:** Linux is the industry-standard, open-source OS for web servers, with distributions like Ubuntu Server and CentOS being popular choices. **Advantages:** It's highly stable, secure, and free. It has massive community support and offers superior performance for web applications. **Weaknesses:** It can have a steeper learning curve for administrators unfamiliar with the command line. | **Characteristics:** Windows Server is a popular choice, especially in corporate environments, offering a graphical user interface (GUI) for management. **Advantages:** It's easy to use and has strong official support from Microsoft. **Weaknesses:** It requires expensive licensing fees and is generally more resource-intensive and a larger target for malware than Linux. | **Characteristics:** Mobile operating systems like iOS and Android are designed as client systems, not servers. **Advantages:** There are no advantages to using mobile devices for hosting. **Weaknesses:** They are entirely unsuitable for hosting a web application. They lack the power, memory, networking capabilities, and OS features to handle server loads. |
| **Client Side** | **Considerations:** Supporting Mac clients requires testing the web application on browsers like Safari, Chrome, and Firefox. The cost is minimal if developers have access to a Mac for testing. It primarily adds time for quality assurance. Standard web development expertise is sufficient. | **Considerations:** Supporting Linux involves testing on browsers like Firefox and Chrome. The cost is negligible as Linux is free. The time and expertise required are part of standard web development practices for ensuring cross-browser compatibility. | **Considerations:** This is the largest desktop user base, so testing on browsers like Edge, Chrome, and Firefox is a baseline requirement. The costs, time, and expertise for this are already assumed in any standard web development project. | **Considerations:** This is a critical and distinct category. Supporting mobile clients requires a **responsive web design** that adapts to various screen sizes. This significantly increases development **time** and **cost**. It requires developers with specific **expertise** in mobile-first design and touch-based user interfaces. |
| **Development Tools** | **Languages:** Java for the backend; HTML, CSS, and JavaScript for the frontend. **Tools:** Cross-platform IDEs such as IntelliJ IDEA, Eclipse, and VS Code run perfectly on Mac. Apple's native IDE, Xcode, is not required for a Java-based web application. | **Languages:** Java, HTML, CSS, and JavaScript. **Tools:** The same cross-platform IDEs are used on Linux. The environment is also rich with powerful command-line tools like Git, Docker, Vim, and Emacs that enhance developer productivity. | **Languages:** Java, HTML, CSS, and JavaScript. **Tools:** Developers can use the same cross-platform IDEs. Microsoft's Visual Studio Code is a very popular choice for this type of development on Windows. | **Languages:** The web client is built with HTML, CSS, and JavaScript. **Tools:** Development is done using standard IDEs. Testing requires browser developer tools with mobile emulation and dedicated simulators like the Android Emulator (in Android Studio) and iOS Simulator (in Xcode). |

## 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 server, I strongly recommend **Linux**. It's the industry standard for hosting web applications due to its rock-solid stability, robust security, and the fact that it's open-source (free). This choice will give The Gaming Room the most power and flexibility for the lowest cost. For the client side, the "platform" isn't the operating system itself but rather the web browser. Our goal will be to support the latest versions of all major browsers across desktop and mobile devices.
2. **Operating Systems Architectures**: The recommended Linux platform uses a monolithic but highly modular kernel. This architecture is layered, with the hardware at the bottom, followed by the **Linux Kernel**, which manages all core processes, memory, and devices. On top of the kernel is the **Shell** (the command-line interface) and other system utilities that allow us to manage the server. Our Java gaming application will run as a user-level process on top of this stack, relying on the kernel to handle networking and system resources efficiently.
3. **Storage Management**: I recommend using a relational database management system (RDBMS) for storing all game data (players, teams, game state, etc.). A great choice would be **PostgreSQL**, which is a powerful, open-source database that works perfectly with Linux and Java. It's highly reliable and can scale to handle a large number of players. For the server's file system, the standard Linux **ext4** filesystem will be used for its reliability and performance.
4. **Memory Management**: Linux uses a sophisticated **virtual memory** system. This means our "Draw It or Lose It" application will run in its own isolated memory space, preventing it from interfering with other system processes. The OS uses techniques like **paging** to efficiently manage RAM and will swap less-used data to the hard disk if necessary. The Java Virtual Machine (JVM) will request a large chunk of memory from Linux and then manage the memory for all the game's objects (like players and teams) within that chunk using its own highly optimized **garbage collector**.
5. **Distributed Systems and Networks**: The application will use a client-server distributed model. Players on their devices (clients) will connect over the internet to your central Linux server. To ensure real-time gameplay, we should use **WebSockets** for communication. Unlike standard HTTP, WebSockets provide a persistent, two-way connection, allowing the server to instantly push game updates to all players.

We must plan for network issues. A player's connectivity might drop, or the server could have an outage. The client-side application will be designed to handle temporary disconnects gracefully. The server will be hosted with a reliable cloud provider that guarantees high uptime to minimize the risk of outages.

1. **Security**: Security is a top priority, and we will implement it at multiple levels:

* **Network Security**: All communication between players and the server will be encrypted using **HTTPS (TLS)** to protect user information from being intercepted.
* **Server Security**: The Linux server will be hardened by using a firewall to block unauthorized access, running the game application with limited user permissions, and keeping the system updated with the latest security patches.
* **Application Security**: Within the Java code, we will validate all incoming data from users to prevent malicious attacks, securely handle user logins and sessions, and ensure that sensitive data like passwords are never stored in plain text but are instead protected using strong **hashing algorithms**.