

# 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 | 07/25/25 | Kashish Prajapati | Added Evaluation and Recommendations sections for Project Two |

**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)

<Write a summary to introduce the software design problem and present a solution. Be sure to provide the client with any critical information they must know in order to proceed with the process you are proposing.>

## 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)

<Identify the design constraints for developing the game application in a web-based distributed environment and explain the implications of the design constraints on application development.>

## [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)

<Describe the UML class diagram provided below. Explain how the classes relate to each other. Identify any object-oriented programming principles that are demonstrated in the diagram and how they are used to fulfill the software requirements efficiently.>

**"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 large web application on a Mac isn’t very common. While macOS Server exists, it’s not built for high traffic or large user bases. It works fine for internal testing or development, but the hardware is expensive and it doesn’t scale well like Linux or cloud platforms. If you’re considering Mac, it’s better suited for developers building and testing apps, not running them live. | Linux is honestly the gold standard for hosting web applications. It’s free, secure, and highly customizable. Most cloud platforms like AWS and Google Cloud use Linux servers under the hood. It can handle thousands of users easily, though you’ll want a developer who’s comfortable working with the command line and server configs. | Windows Server can definitely host web apps, especially with tools like IIS (Internet Information Services). It’s stable and integrates well with other Microsoft tools. However, licensing can get pricey, and it’s not as lean as Linux when it comes to performance at scale. Still, it’s a valid choice for teams already using Microsoft infrastructure. | Mobile devices like Android and iOS don’t host apps themselves — they connect to the server where the game runs. So you won’t host anything here, but you’ll need to make sure the backend is fast and stable enough to handle mobile users from all over. |
| **Client Side** | Supporting Mac users means testing the app on Safari and Chrome, especially since Safari can behave differently from other browsers. You’ll also need Mac hardware for testing, which can add to your development cost. That said, the user experience can be polished and consistent once it’s optimized. | On Linux desktops, users usually run Chrome or Firefox. As long as the app is responsive and browser-friendly, it’ll work well. Linux has fewer quirks than Windows or Mac, which makes cross-browser testing a little easier here. | Windows users make up the largest desktop audience, so you’ll need to make sure the app runs smoothly on Edge, Chrome, and Firefox. Compatibility testing is important here, especially since older Windows machines may act up in unexpected ways. | Mobile users expect speed, smooth performance, and a design that fits their screens. You’ll need to test the game on both Android and iOS, and make sure it’s responsive and touch-friendly. This adds time to development but is crucial for a good user experience. |
| **Development Tools** | Developers on Mac usually rely on tools like Xcode, Visual Studio Code, and Safari’s developer tools. These are all powerful and free, but require a Mac to run. It’s a great setup for teams already familiar with the Apple ecosystem, especially if iOS support is also planned. | Linux gives developers tons of flexibility. Most tools — VS Code, Node.js, Git, and even full-stack frameworks — are free and open-source. There are no extra costs, and experienced developers will feel right at home. This makes it ideal for cost-effective, scalable development. | Windows developers can use Visual Studio, which is great for managing larger projects. Some features are locked behind a paid license, but there’s a free community edition. It’s a powerful and well-supported environment that works well for teams using .NET or hybrid stacks. | For Android, Android Studio is the go-to (and free). For iOS, you’ll need Xcode, which is also free but only runs on Mac. You could also use cross-platform tools like React Native or Flutter, which let you build both versions at once. It saves time and keeps things consistent across platforms. |

## 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 hosting the backend portion of your web-based game, I strongly recommend choosing **Linux** as the primary operating platform.

Linux stands out for several reasons:

* It’s free and open-source, which immediately lowers your long-term operating costs.
* It’s highly stable and secure, meaning less downtime and better protection for your users.
* Most of the internet’s infrastructure already runs on Linux — including platforms like Google Cloud, AWS, and Azure — so it’s future-proof.
* It scales easily. Whether you’re supporting 100 players or 10,000, Linux-based systems can grow with you.

Compared to Mac or Windows, Linux offers far more flexibility and efficiency, especially in distributed environments where performance and resource usage matter. Mac is great for testing and local development, but it’s not commonly used for hosting live web applications. Windows is powerful but involves complex licensing and tends to consume more resources, which may be overkill for your needs.

### ****2. Operating System Architecture****

Choosing Linux also means you’ll benefit from a **layered system architecture**, which naturally supports separation of concerns — an important principle in web development. For example:

* The **presentation layer** (HTML/JS frontend) runs independently in the browser on any device.
* The **application logic** lives on a Linux web server — potentially powered by Node.js, Python, or Java — and handles user input, scoring, and game progression.
* The **data layer** is handled by a database system like PostgreSQL or MongoDB, either hosted locally or in the cloud.

This kind of modular setup makes the game easier to manage, update, and scale in the future. Linux supports a wide variety of open-source tools that align perfectly with this architectural style.

### ****3. Storage Management****

To handle your game’s data — like user accounts, scores, session information, and game histories — I recommend using a **cloud-hosted database**. Options like **Amazon RDS** (for SQL-based storage) or **MongoDB Atlas** (for flexible NoSQL structures) would be ideal.

These services offer:

* **Automatic backups**, so your data is never lost.
* **High availability**, meaning your game stays online even if one part fails.
* **Encryption** at rest and in transit, keeping user data secure.
* **Scalability**, so your storage can grow with your user base.

This also takes pressure off your internal team — since the infrastructure and maintenance are handled by the provider, allowing your developers to focus on game features rather than server upkeep.

### ****4. Memory Management****

Memory usage becomes especially important in games where multiple users are interacting in real time, possibly across many rounds or sessions. Linux is built to handle this efficiently.

Linux uses a combination of:

* **Virtual memory**, which keeps the game responsive even under heavy load.
* **Caching and prefetching**, which helps serve frequently used assets quickly.
* **Lightweight resource allocation**, so you’re not wasting memory on background processes.

This means your server can support more simultaneous users without slowing down or crashing. For a game that will likely experience traffic spikes — such as during evenings or weekends — this is a major advantage.

### ****5. Distributed Systems and Networks****

To achieve the multi-platform support your team wants — including Android, iOS, and browser access from desktops — the game will need to use a **distributed system architecture**.

Here’s how that would look:

* The **frontend** (the user interface players interact with) runs in a web browser or mobile app.
* The **backend** (where game logic, timers, guesses, and scoring happen) runs on a Linux server.
* Communication between the two happens over a **secure API**, using either **REST** or **WebSockets** depending on how real-time you want the experience to feel.

Hosting this architecture on a cloud platform like **AWS**, **Azure**, or **Google Cloud** will also give you access to:

* **Load balancers**, which spread out incoming traffic so no server is overwhelmed.
* **Redundant backups and failovers**, which keep the game online if part of the network fails.
* **Content delivery networks (CDNs)** to deliver game assets (like drawings) quickly around the world.

This kind of architecture is not just modern — it’s essential for creating a seamless experience that works equally well on a phone in New York and a laptop in California.

### ****6. Security****

Security is one of the most important considerations for any platform, especially with a game that allows user accounts and tracks progress.

Here’s what I recommend implementing:

* **HTTPS encryption**: All data transferred between the client and server must be encrypted to protect against attacks like man-in-the-middle eavesdropping.
* **JWT tokens for authentication**: These tokens allow users to stay logged in securely, and the server can verify their identity without storing passwords in memory.
* **Role-based access control (RBAC)**: Admins and moderators should have different access permissions than regular players. This limits the chance of unauthorized changes or data access.
* **Encrypted databases**: Whether you use MySQL, PostgreSQL, or MongoDB, ensure that sensitive data (like user credentials) is encrypted both in transit and at rest.
* **Firewall rules and access logs**: Your servers should only allow traffic through approved ports (like 443 for HTTPS), and your logs should capture suspicious activity to allow early detection of potential threats.

Linux is very strong in this area — with built-in firewall (iptables/ufw), user permission management, and strong community tools like Fail2Ban or SELinux to lock down your environment.

**Conclusion:**

Expanding “Draw It or Lose It” across multiple platforms is not only possible — it’s a smart next move. With a backend powered by Linux, responsive web technologies for the frontend, and mobile optimization through tools like React Native or Flutter, the game can reach players on virtually any device, anywhere.

By leaning into a distributed architecture, leveraging cloud infrastructure, and making smart choices about security and storage, your team will be able to deliver a smooth, scalable, and fun experience to players — without being bogged down by expensive infrastructure or technical limitations.