

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 |
| --- | --- | --- | --- |
| 3.0 | 10-19-2025 | Daniel Norman | 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)

Draw It or Lose It is currently only available as an Android application. The Gaming Room wants to extend the game into a web-based version that will be supported by multiple platforms. The new application must allow for multiple teams, each with multiple players, while ensuring that game and team names remain unique. Additionally, only one instance of the game can exist in memory at a time.

To address the client’s needs, the design incorporates established software design patterns. The Singleton pattern ensures only one GameService instance exists in memory, the Iterator pattern enforces uniqueness for game and team names, and a common Enitiy base class provides shared attributes. These design decisions support scalability, security, and efficiency in web-based distributed architecture.

## Requirements

* **Team Structure**
  + A game must support one or more teams, and each team can include multiple players
* **Unique Identification**
  + Game and team names must be unique so users can check availability before creating a new game or team
* **Single Active Instance**
  + Only one instance of the game should exist in memory at a time
* **Scalability and Maintainability**
  + The application must support future growth, preserve data integrity, and manage players and teams efficiently in a distributed environment.

## [Design Constraints](#_2et92p0)

* Singleton
  + Only one GameService instance to prevent global state duplication
* Uniqueness
  + Game and team names must be unique, requiring collection iteration before insertion
* Concurrency
  + Real-time multiuser play demands thread-safe structures and efficient communication
* Web Deployment
  + Cross-platform support mandates standards-based Java and web frameworks

## [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)

* Entity (abstract) attributes id, name
* Game inherits Entity has collection of Teams
* Team inherits Entity has collection of Players
* Player inherits Entity
* Demonstrated OOP principles: inheritance, composition, encapsulation, and abstraction.

**"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:** Stable and secure; good for local development, testing, and staging.  **Advantages:** Excellent developer tools, Unix-based (like Linux).  **Weaknesses:** High hardware cost; macOS Server discontinued for most production use.  **Licensing Costs:** Requires Apple hardware; no separate OS licensing fee, but not viable for large-scale hosting. | **Characteristics:** Industry standard for web servers.  **Advantages:** Open-source, highly scalable, cost-effective, supported by all major cloud providers (AWS, Azure, GCP).  **Weaknesses:** Requires Linux admin skills; less GUI-based management.  **Licensing Costs:** Free and open-source, minimizing total cost of ownership. | **Characteristics:** Common in enterprise and corporate hosting (IIS, .NET, Azure).  **Advantages:** Excellent integration with Microsoft ecosystem.  **Weaknesses:** Higher resource usage and licensing costs; less popular for open-source stacks.  **Licensing Costs:** Windows Server licenses and CALs can add significant costs. | **Characteristics:** Not viable for hosting; client devices only.  **Advantages:** N/A for server deployment.  **Weaknesses:** Limited resources and OS restrictions.  **Licensing Costs:** N/A (devices use bundled OS). |
| **Client Side** | **Characteristics:** Great for web and cross-platform development; supports Safari, Chrome, Firefox.  **Advantages:** Required for iOS builds and testing; strong UI/UX tools.  **Weaknesses:** Higher hardware cost; limited browser diversity.  **Development Considerations:** Easy integration with Xcode and web stacks; responsive web design testing possible. | **Characteristics:** Popular among developers for web and backend tools; supports all modern browsers.  **Advantages:** Ideal for open-source dev environments; close to production parity.  **Weaknesses:** Limited native GUI tools.  **Development Considerations:** Great for containerized app testing and browser-based UI validation. | **Characteristics:** Widespread enterprise use; supports Chrome, Edge, Firefox.  **Advantages:** Excellent for .NET, React, and Node.js apps; strong debugging tools.  **Weaknesses:** Differences in environment vs. Linux production servers may require extra testing.  **Development Considerations:** Supports responsive browser testing across major platforms. | **Characteristics:** Final end-user platforms (Android/iOS).  **Advantages:** Native app support and mobile browser access for responsive design.  **Weaknesses:** Fragmentation (screen sizes, OS versions); device testing needed.  **Development Considerations:** Responsive design critical; must use mobile frameworks and device testing services (e.g., Firebase Test Lab, BrowserStack). |
| **Development Tools** | **Languages & Tools:** Node.js, npm/yarn, VS Code, Xcode (for iOS), Docker Desktop.  **Impact on Team:** iOS builds require macOS; may need at least one Mac in the dev team.  **Licensing Costs:** Most tools free; Apple developer license required for iOS distribution (~$99/year). | **Languages & Tools:** Node.js, npm/yarn, VS Code, JetBrains IDEs, Docker/Kubernetes, PostgreSQL.  **Impact on Team:** Matches production environment; ideal for backend and DevOps work.  **Licensing Costs:** Mostly free/open-source; optional JetBrains license (~$250/year per user). | **Languages & Tools:** Node.js, npm/yarn, VS Code, WSL2, Visual Studio, Docker.  **Impact on Team:** Windows-native devs can use WSL2 for Linux parity; slightly more setup overhead.  **Licensing Costs:** Windows OS and Visual Studio licenses may add cost. | **Languages & Tools:** React Native, Flutter, Android Studio, Xcode, Chrome DevTools.  **Impact on Team:** May require specialized mobile developers; separate builds for Android/iOS.  **Licensing Costs:** Android Studio free; Apple Developer Program ($99/year). Device farm testing tools may incur cost. |

## 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**

I recommend using Linux server to host Draw It or Lose It. Linux is free, stable, and works well with most web technologies. It is also the most common operating system for servers on the internet, making it easy to find support and cloud hosting options. Linux will allow the game to scale as more players join and keep operating costs low.

1. **Operating Systems Architectures**:

Linux uses a modular architecture that separates the kernel from user processes. This setup makes it reliable and secure for running multiple services at the same time. For Draw It or Lose It, the game will run on a three-tier setup – the frontend (the web app or module app users play on), the backend (the web server that runs the game logic and connects teams), and the database (where game and player data are stored). This structure helps keep the system organized, easy to update, and able to handle lots of players at once.

1. **Storage Management**:

We will use PostgreSQL, a reliable database that keeps all games, team, and player data safe and organized. It supports unique names for games and teams, which fit the client’s needs. For faster access, the system can also use Redis, which temporarily stores active game data in memory so that gameplay feels instant. Player drawings or images can be stored in cloud storage like AWS.

1. **Memory Management**:

Linux manages memory efficiently by sharing resources between processes and freeing memory when it is no longer needed. The system will limit how much memory each part of the game can use, so one busy process does not slow down everything else. The application itself should also clear out old data and use caching wisely, so real-time gameplay runs smoothly without lag or crashes.

1. **Distributed Systems and Networks**:

Since Draw It or Lose It will run on many devices, it needs a distributed system. This means different parts of the game can run on different servers and still talk to each other over the network. Regular actions, like logging in or checking scores, can use standard web requests, while live drawing and guessing should use WebSockets, which allow instant two-way communication. If one server goes down, others can keep the game running, which makes the system more reliable.

1. **Security**:

Security is very important for protecting users and their data. All communication between devices and servers should use HTTPS to encrypt data. Players should log in using a secure authentication system, like usernames and passwords stored safely or accounts linked to Google or Apple. Sensitive information, such as passwords or personal data, should be encrypted in the database. The system should also limit how often a user can send requests to prevent spam or cheating. Regular updates and backups will help protect against hackers and data loss.