

Draw It or Lose It

# **CS 230 Project Software Design**

Version 3.0

## 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 | 09/16/2025 | Zachary Russo | First submission: Software design and implementation plan. |

| 2.0 | 10/4/2025 | Zachary Russo | Second Submission: Evaluation of the development requirements on the server side, client side, and development tools. Along with the respected OS from Mac, Linux, Windows, and Mobile |
| --- | --- | --- | --- |
| 3.0 | 10/16/2025 | Zachary Russo | Third Submission: Analyzed the characteristics of and techniques specific to various systems architectures and made recommendations to The Gaming Room. |

**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 a multiplayer game that is proposed to be developed on a web-based, distributed game service. The client, “The Gaming Room,” requires a robust server-side design that supports multiple teams and players, enforces unique game/team names, and ensures only one in-memory service manages games at runtime. This document proposes a clear domain model, describes constraints for web-based deployment, and delivers a Java implementation that demonstrates the Singleton and Iterator patterns to meet the client's requirements. The provided design strikes a balance between simplicity for initial development and clear extension points for later phases.

## Requirements

Business & technical requirements:

* A game can have one or more teams.
* Each team can contain multiple players.
* Game names and team names must be unique (so users can verify availability).
* Only one instance of the game service exists in memory at a time (Singleton).
* Entities (Game, Team, Player) must have unique identifiers.
* Provide the ability to add/get games, add/get teams & players, and return counts/IDs.

## [Design Constraints](#_2et92p0)

Key constraints and implications:

* Stateless web front-end, stateful server: The application must initially maintain game state in memory for the prototype; later phases should persist to a database. For now, design the server-side services so persistence can be added without changing domain classes (Repository/DAO pattern later).
* Single-service in memory: Use Singleton pattern for GameService to guarantee one in-memory manager. For clustered deployment later, a shared persistence layer or distributed lock service will be required — this prototype focuses on single-process correctness.
* Concurrency: Multiple clients may attempt to register names simultaneously. The in-memory service must be thread-safe; the prototype uses synchronization for ID generation and name checks. For production, transactional DB locks or optimistic concurrency control will be needed.
* Name uniqueness scope: Game names are globally unique. Team names are treated globally unique to allow simple "is name in use?" checks. Player name uniqueness is enforced per-team (players within a team must have distinct names).
* Platform independence: Use plain Java domain classes and avoid platform-specific libraries so server code is portable among Linux/Mac/Windows.

## [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 UML defines:

* Entity (base class): id: long, name: String, constructors, getters, toString().
* GameService: holds List<Game> games, nextGameId, nextTeamId, nextPlayerId, service (singleton), and methods getInstance(), addGame, getGame, getGameCount, getNextPlayerId, getNextTeamId.
* Game extends Entity: List<Team> teams, addTeam(name).
* Team extends Entity: List<Player> players, addPlayer(name).
* Player extends Entity.

OOP principles used:

* Inheritance: Game, Team, Player inherit common id/name behavior from Entity, reducing duplication.
* Encapsulation: Collections are private; access through methods.
* Single Responsibility: GameService manages lifecycle and IDs; domain classes manage their contained entities.

Design Patterns:

* Singleton: GameService provides a single access point for game lifecycle.
* Iterator (simple use): for / enhanced-for loops used to search lists for name uniqueness and retrieval.

**"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** | Mac is convenient for development and local testing. It supports Java servers well (Tomcat, Jetty) but its use for production is less common than Linux. | Linux is ideal for server deployment: mature tooling, performance, package management, and container support. Recommended for production. | Windows can host Java servers but has extra admin overhead; sometimes preferred in Windows-centric shops. | Mobile devices are clients only; they are not recommended for hosting server-side services. |
| **Client Side** | Web dev and testing on Mac are smooth (Chrome, Safari). Mac is good for developing native macOS clients if needed. | Linux works for web clients and development; fewer proprietary testing tools for iOS. | Windows has broad tooling for building electron apps and browser testing. | Mobile requires responsive web UI or thin native wrappers (Android/iOS). Testing on emulators and devices is needed; build pipelines differ per OS. |
| **Development Tools** | Java + IntelliJ/Eclipse, Node/npm for front-end, browsers for testing. | Java + Maven/Gradle, Docker, Linux terminal-friendly tools. | Java + IntelliJ/Eclipse, Windows installers for dev tools. | Android Studio for Android; Xcode for iOS (mac required for building iOS apps). Hybrid/web tech (React Native/Flutter) can reduce complexity. |

## 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**: Linux seems like the best option for actually running the server side of the game because it’s used a lot for web hosting and has a lot of support online. I think Windows and Mac could still be used for development on our own computers, but for production, Linux might be more stable and cheaper, since it is open source, you wouldn’t need to buy licenses to run the Linux OS.
2. **Operating Systems Architectures**: Linux servers usually run on x86\_64 hardware and can be put into containers like Docker. That seems like it would make it easier to deploy or scale the game in the future without having to rebuild everything.
3. **Storage Management**: At this point, we’re just keeping everything in memory, but eventually the game will need a database so we don’t lose information when the server restarts. Something like PostgreSQL or MySQL seems common for this kind of application. These databases are reliable, easy to back up, and have good support for web-based applications. Static assets, like image files, could be stored on a cloud service or CDN for faster delivery.
4. **Memory Management**: Since this is Java, I know the JVM handles garbage collection for us, so we don’t have to free memory manually. We should still be careful about keeping only the active games and removing finished ones so memory doesn’t keep growing. Maybe we can build monitoring tools.

**Distributed Systems and Networks**: Right now everything is in one process, but to let mobile and web clients talk to the game we’d probably need to build some kind of API (maybe REST over HTTPS). If the game expands to multiple servers, the Singleton pattern would need to be adapted since each server instance would have its own memory space. Using a shared database or distributed cache like Redis could help synchronize game state across servers.

1. **Security**: Security is critical since players will log in and send data over the network. At a minimum, we’d use HTTPS so the data between the client and server is encrypted. For user accounts and information, we’d probably need some form of authentication like tokens or passwords stored securely. Linux also provides strong permission controls and firewall options to help protect the application from outside attacks.