

The Gaming Room

# **CS 230 Project Software Design Template**

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 |
| --- | --- | --- | --- |
| 3.0 | 10/19/2025 | Marcelo Barbosa | Added recommendations and comments to Operating Platform, Operating System Architectures, Storage Management, Memory Management, Distributed Systems & Networks, and Security. |

**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)

## Problem:

The Gaming Room wants a web-based, multi-platform version of their Android only game. The application must support one active game instance in memory, multiple teams and multiple players per team, unique names for games and teams so users can validate name availability, and unique identifiers for games, teams, and players.

## Solution:

Provide a clean domain model with Entity as a base class and a concrete Game, Team, and Player classes that inherit from Entity. Implement a thread-safe GameService singleton to manage game lifecycle, ID generation, and collections. Enforce unique names by iterating existing collections before adding new entities. Deliver a software design document, a Java prototype implementing these patterns, and guidance for deployment and platforms. This provides a maintainable, testable foundation for expanding to distributed, multi-platform web clients.

## Requirements

1. App must support one or more teams per game and multiple players per team.
2. Only one instance of the game manager (GameService) exists in memory at a time (singleton).
3. Unique identifiers for games, teams, players.
4. Game and team names must be unique so clients can check availability when registering names.
5. Use object-oriented design (base Entity class) and design patterns (singleton, iterator usage).
6. Provide an initial prototype in Java (Eclipse) demonstrating these behaviors.

## [Design Constraints](#_2et92p0)

## Single-process in-memory singleton:

The current requirement that only one instance of the game service resides in memoryimplies a single-process context. In distributed/multi-instance deployments, additional coordination (distributed locks or centralized services) will be required to ensure logical singleton behavior across nodes.

## Unique name enforced at runtime by iteration:

This is simple and sufficient for small data. It will create O(n) checks for name creation.

## Web-based multi-platform clients: The backend must expose platform-agnostic APIs (REST, WebSocket for real-time). Cross-origin resource sharing and authentication must be considered.

## Concurrency:

Multiple clients will attempt concurrent operations (create team, add player). The service must guard against race conditions (synchronization or transactional persistence).

## Security & privacy constraints:

Names and IDs are non-sensitive, but eventual user accounts and authentication must use TLS, server-side validation, input sanitization, and secure storage.

## [System Architecture View](#_ilbxbyevv6b6)

1. Web browsers (responsive web), mobile apps (iOS/Android), using REST + WebSocket for real-time draws and game state updates.
2. Java-based REST API (e.g., Spring Boot) hosting GameService logic and persistence adapters. Use WebSocket (or Socket.IO) for real-time drawing events.
3. Relational DB (PostgreSQL).
4. When scaled to multiple servers, use a session store / centralized coordination to manage the logical single-game constraints.
5. Cloud provider (AWS/Azure) for flexibility.

## [Domain Model](#_8h2ehzxfam4o)

## Class relationships & behavior:

1. Entity (base) contains the common attributes id: long and name: String, with getters and toString().
2. Game extends Entity and contains List<Team> teams and methods like addTeam(String name) that ensure team name uniqueness within the game.
3. Team extends Entity and contains List<Player> players and addPlayer(String name) that ensures uniqueness of player names within that team.
4. Player extends Entity.
5. GameService is a singleton with a list of Game objects; it holds nextGameId, nextTeamId, nextPlayerId and generates unique ids. GameService has methods addGame(name), getGame(id), getGame(name), and count/getNextId helpers.

## OOP principles demonstrated:

1. Inheritance: Game, Team, Player inherit common behavior and fields from Entity, reducing duplication.
2. Encapsulation: Each class manages its own list of children (Game manages teams, Team manages players) and provides methods to modify internal state. Fields are kept private and manipulated through methods.
3. Single Responsibility Principle: GameService handles lifecycle and id generation; Game handles teams; Team handles players.
4. Design Patterns: Singleton (GameService) ensures one instance. Iterator-style checks (traversing lists to find names/ids) enforce uniqueness.

**"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 provides a solid development environment and can run local web servers (Tomcat, Spring Boot, Node.js) for testing and small deployments. However, macOS is not commonly used for production. macOS servers, and licensing/hardware costs make it impractical at scale. For The Gaming Room, macOS is best suited for developer workstations and testing, not large-scale hosting. | Linux is the standard choice for production web servers and cloud deployments. It is lightweight, stable, and integrates seamlessly with Java, Node.js, and containerized environments. Most cloud providers offer Linux-based hosting (Ubuntu, Amazon Linux), often with minimal or no licensing costs. For a scalable, cost-effective architecture supporting thousands of concurrent players, Linux is the ideal platform. | Windows Server can host Java (OpenJDK), .NET, and containerized backends, but its licensing fees and higher system overhead make it less efficient for Linux-oriented tech stacks. While it supports IIS and Windows-specific tools, Linux remains the more common and economical choice for web-based applications. Windows is most useful for testing and supporting cross-platform development teams rather than production hosting. | Mobile devices are not designed for server hosting. Instead, they act as clients that connect to centralized servers. They rely on network connections to interact with the backend system, which manages authentication, data storage, and game state. The focus for mobile platforms is optimizing communication with the server through secure, low-latency APIs and WebSocket connections. |
| **Client Side** | macOS desktops use modern browsers (Safari, Chrome, Firefox) that fully support HTML5, WebSockets, Canvas, and responsive UIs. Development cost is moderate: developers can build and test responsive web front ends on macOS machines. macOS is also required for building native iOS apps (Xcode). | Linux desktops run modern browsers and are excellent test targets for cross-browser compatibility. Linux has smaller desktop market share, so testing priority is lower but still necessary. Web client behavior is consistent, but differences in fonts, rendering, and input devices should be checked. | Windows desktops have the largest desktop user base; browsers are widely used. Because user base is large and varied, testing breadth must be greatest here. Ensuring the responsive UI behaves well across Windows versions and browsers is critical to delivering a consistent gameplay experience. | Mobile clients require careful UI design for touch interactions and varying screen sizes. For the drawing/displaying of images and fast, synchronized updates. |
| **Development Tools** | Developers on macOS can use Java toolchains, Node.js/NPM, and front-end frameworks. macOS is mandatory for building/signing iOS native apps (Xcode). Licensing: IntelliJ Ultimate and certain proprietary tools may incur costs; the JDK, Node, and many frameworks are open source. | Linux environments favor OpenJDK, Maven/Gradle, Docker, Kubernetes CLI tools, NGINX, and CI/CD tooling. Most tools are open source so licensing costs are typically minimal. For production, Linux supports containerization and automation better than other platforms; CI/CD pipelines are commonly Linux-hosted. Team impact: fewer platform-specific constraints simplify ops and reduce tool licensing costs. | Windows developers will use JDKs, IDEs (IntelliJ/Eclipse, Visual Studio for .NET components), and front-end toolchains (Node/NPM). Windows is required if parts of the stack must integrate with Microsoft-only technologies. Many teams standardize on cross-platform tools to avoid fragmentation. | Mobile development requires additional toolchains and expertise. Mobile store deployment (App Store, Google Play) adds process overhead and potential costs (Apple Developer Program, Google Play Console fees). For web-first approach, using a responsive HTML5 front-end minimizes mobile-specific toolchain costs but may limit access to native capabilities. |

* Server hosting: Use Linux, containerized and orchestrate with Kubernetes or a managed container service. It is the lowest-cost, most scalable, and best-supported option for Java/Node/WebSocket backends.
* Client delivery: Implement a responsive HTML5 front-end that uses WebSocket for real-time drawing/state. For better native-feel or higher drawing performance, adopt a cross-platform mobile framework or build thin native shells.
* Development staffing & tools: Keep a unified web development team (front-end + back-end) and add one or two mobile specialists to cover iOS/Android. Use open toolchains where possible (OpenJDK, Node, Docker) and budget for any commercial IDE/licenses and for Apple Developer Program fees if publishing to the App Store.

## Recommendations

Analyze the characteristics of and techniques specific to various systems architectures and make a recommendation to The Gaming Room. Specifically, address the following:

By hosting on Linux with containerized services, using PostgreSQL plus Redis for smart data handling, and deploying a secure distributed architecture. The Gaming Room can achieve a scalable, cross-platform, and secure version of the Draw It or Lose It.

1. **Operating Platform**: Host application servers on Linux. Linux is mature, performant, widely supported in cloud providers, has strong container support, and is cost-effective. Use container and a managed container orchestration service on a cloud provider like AWS.

Linux is the industry standard for web backends and cloud hosting. It is mature, low-cost, well-supported by Java and Node ecosystems, and integrates tightly with containers and CI/CD pipelines. Container orchestration enables rapid horizontal scaling, health checks, rolling updates, and predictable operations for real-time multiplayer workloads.

1. **Operating Systems Architectures**: Use 64-bit x86\_64 Linux instances, containerized with Docker. For horizontal scaling, use Kubernetes or managed container services (EKS/GKE/AKS). Use separate tiers for API servers and WebSocket real-time servers.   
   All tiers are isolated in containers, making deployment predictable and reducing configuration drift.
2. **Storage Management**: Use PostgreSQL for core persistent data (games, teams, user accounts) and Redis for ephemeral real-time session state and pub/sub for notifications and transient game state. Put unique constraints for names in the DB to ensure global uniqueness.  
   S3 storage for uploaded assets. This hybrid approach balances durability with low-latency state required for a real time game.
3. **Memory Management**: Use JVM tuning with appropriate heap sizes; rely on OS-level memory management; as traffic grows, scale out with containers instead of only scaling up memory. Use health checks and monitoring (Prometheus/Grafana).  
   Container aware memory management plus horizontal scaling provides predictable performance and simpler operational control for peak concurrent players.
4. **Distributed Systems and Networks**: Use **REST APIs** for standard operations and **WebSocket connections** for real-time gameplay updates. Deploy services behind **load balancers** or **session stores** to maintain state across instances. Implement **fault-tolerant mechanisms** like retries, circuit breakers, and reconnection logic to handle outages. Ensure global consistency through **centralized ID generation** (using database sequences or UUIDs) for unique game, team, and player identifiers across distributed environments.
5. **Security**: Implement TLS encryption for all communications (HTTPS/WSS) to protect data in transit. Use secure authentication and authorization methods with encrypted tokens and hashed passwords. Apply server-side input validation to prevent injection attacks and sanitize all user data. Enable rate limiting and DDoS protection through the cloud provider. These measures ensure strong data protection and maintain user privacy across all platforms.