

<Name-of-Software-Application>

# **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 | 09/17/2025 | Jordan Valladares | The code will now have the ability to have one or more teams involved in the game.  Each team will have multiple players assigned to it.  Game and team names will be unique to allow users to check whether a name is in use when choosing a team name.  Only one instance of the game will exist in memory at any given time. Unique identifiers for each instance of a game, team, or player will be created. |

**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)

The Gaming Room wants to modernize **Draw It or Lose It** into a web-based, multi-platform application. The game supports multiple teams with multiple players, requires unique names (for games and teams) to prevent confusion during selection, and must ensure only one **GameService** instance manages IDs and global state. We propose a service-centered design using a **Singleton** GameService (centralized IDs and registries) and **Iterator**-based lookups for enforcing unique names, implemented in Java with clean OOP inheritance (shared base Entity for Game, Team, Player). This approach keeps logic cohesive, scalable, and easier to deploy across platforms while meeting all requirements.

## Requirements

The Gaming Room requires the modernization of Draw It or Lose It into a web-based, multi-platform game that supports multiple teams and players. Each team and game name must be unique, and a single authoritative GameService instance must control all identifiers and sessions. The solution must enable responsive play through standard web browsers on desktop (Linux, Mac, Windows) and mobile (iOS, Android). It should provide secure authentication, scalable server performance, and efficient management of drawings, scores, and user data while ensuring portability, maintainability, and low operating cost.

## [Design Constraints](#_2et92p0)

**Single authoritative service (Singleton):** Only one in-memory instance of GameService may exist to generate unique identifiers for games, teams, and players and to provide a consistent registry. Implication: The constructor must be private and access provided via getInstance().

**Name uniqueness (Iterator):** Game and team names must be unique. We must iterate over the existing collections before adding new items. Implication: addGame(), getGame(name), Game.addTeam(), and Team.addPlayer() perform case-insensitive searches over lists before insertion.

**Web-based, distributed environment:** The app must run across platforms and browsers, so server-side data structures should be thread-safe (or synchronized) and client-agnostic. Implication: avoid device-specific APIs, prefer simple POJOs for domain logic, and keep persistence/network concerns abstracted for later tiers.

**Extensibility & clarity:** Shared attributes/behaviors (id, name) belong in a base class (Entity) to reduce duplication and centralize validation. Implication: domain classes (Game, Team, Player) inherit from Entity.

## [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 provided UML shows a base **Entity** with id: long and name: String. Game, Team, and Player inherit from Entity. **GameService** manages List<Game> and static ID counters (nextGameId, nextTeamId, nextPlayerId) and exposes getInstance(), addGame(), getGame(id), getGame(name), and ID accessors. Game contains a List<Team> and addTeam(name). Team contains a List<Player> and addPlayer(name). ProgramDriver uses SingletonTester. This demonstrates **inheritance** (shared base Entity), **encapsulation** (private fields + getters), **Singleton** (only one GameService), and **Iterator** (search over lists for uniqueness). These choices meet the requirements for single service instance, unique names, and multi-team/player structure 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** | macOS can host a production web app (e.g., Nginx/Apache reverse proxy + Node.js/Java/Python app servers + PostgreSQL/MySQL), but it’s **not a common or cost-effective choice** for internet-facing workloads. Hardware options are limited, and cloud providers don’t offer macOS instances for general web hosting. **Licensing:** macOS is licensed with Apple hardware—there’s no separate server OS license cost, but you’re tied to Apple devices, which raises TCO versus commodity Linux servers. | Linux is the **industry standard** for web hosting: stable, high-performance, secure, and easy to scale on any major cloud (AWS, Azure, GCP) or on-prem. It supports the full modern web stack (Nginx/Apache/Caddy; Node.js, Java/Spring, Python/Flask or Django, Go; PostgreSQL/MySQL) with mature automation (systemd, Docker, Kubernetes, Terraform). **Licensing**: popular distros (Ubuntu, Debian, AlmaLinux) are **no-cost**; optional paid support is available ( Red Hat, Canonical) but not required. | Windows Server can host the stack using IIS/ASP.NET Core or via Nginx/Node/Java. It integrates well with **Active Directory,** Windows auth, and .NET tooling. Scaling and observability are solid on Azure/AWS. **Licensing**: Windows Server typically requires **paid licenses** (per core/CALs), increasing operating cost compared to Linux. Operational overhead (patch cadence, GUI administration) can be higher, though modern Windows Core/Nano images help. | Mobile phones/tablets are **clients,** not appropriate as the web server host for thousands of users. They cannot reliably provide public ingress, persistent processes, or server-grade security/observability. At most, a device could run a **local dev server** for debugging, but mobile is **not suitable for production hosting. Licensing**: not applicable for server OS; app store policies and dev program fees apply to client apps, not hosting. |
| **Client Side** | For desktop Safari and Chrome/Firefox on macOS, ship a **responsive, standards-compliant** web app (HTML5/CSS3/ES modules). Cost/time are modest if you maintain **one web codebase** and use responsive design + progressive enhancement. Ensure keyboard and game input latency are optimized for Safari/WebKit. | Support Firefox and Chrome on diverse distros/DEs. Ensure fonts, audio (PulseAudio/PipeWire), and hardware acceleration work without proprietary codecs. Focus on **cross-browser testing** and avoiding OS-specific assumptions (paths, case sensitivity). Cost is primarily QA time; dev expertise is standard front-end skills plus basic Linux troubleshooting for users. | Target Chromium/Edge and Firefox with **pointer/keyboard/gamepad** support, good IME handling, and accessibility (UIA/ARIA). Consider high-DPI scaling and variable refresh rate monitors. Time/cost center on **QA coverage** (multiple Windows versions, GPU/driver differences) rather than separate code. | Deliver a **mobile-first PWA:** responsive layout, touch gestures, viewport meta, offline caching for assets, and performance budget for 60fps canvas/animations. Test Safari iOS (WebKit) and Chrome/Firefox Android (Blink/GeckoView). Handle virtual keyboards, safe areas/notches, battery/network variability, and add-to-home-screen. If needed, wrap the same web app via **Capacitor/Cordova** for store distribution. Cost/time are mainly in **mobile QA** and performance tuning, not separate feature code. |
| **Development Tools** | Languages/frameworks: **TypeScript/JavaScript** (React/Vite), **Node.js, Java/Spring Boot, Python/Flask or Django;** DB: PostgreSQL/MySQL. Tooling: VS Code, IntelliJ IDEA/CLion, Xcode (for iOS device testing of the web view/PWA), Docker Desktop or Colima, Postman/Insomnia, Git. **Licensing**: most tools are **free/open-source;** IntelliJ is **commercial** (optional). No mac-specific runtime lock-in. | Same language choices as above; Linux is often the **primary environment** for Docker, Kubernetes, Nginx, and CI/CD agents. Tools: VS Code, JetBrains IDEs, Docker/Podman, kubectl/Helm, cURL, OpenSSL, tmux. **Licensing**: predominantly **free/open-source;** optional paid JetBrains licenses or vendor support. Excellent parity with production servers. | Languages/frameworks: **.NET/ASP.NET Core** (first-class on Windows and runs cross-platform), plus Node.js/Java/Python stacks. Tools: VS Code, **Visual Studio** (Community free for individuals/small orgs; paid for larger orgs), WSL2 for Linux-like dev, Docker Desktop, Postman, Git. **Licensing:** Visual Studio Pro/Enterprise is **paid**; otherwise many tools are free. | You’ll develop on desktop OSes and **test on devices/emulators**. iOS testing needs a Mac (Xcode + Simulator) if you do any native/PWA integration tests; Android Studio works on all three desktop OSes. For a web-only PWA, use Chrome DevTools/Safari Web Inspector for device emulation and remote debugging. **Licensing**: IDEs/emulators are free; optional Apple ($99/yr) and Google Play fees only matter if you ship wrapped apps. |

## 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. **L**inux is the best operating platform for The Gaming Room’s expansion. It provides cost-effective scalability, high availability, and broad compatibility with cloud environments such as AWS and Azure. Linux supports a wide range of programming languages, web servers, and frameworks that fit the project’s distributed architecture.
2. A three-tier architecture should be used: (1) Presentation Layer – the web-based front-end application (HTML5, CSS, JavaScript); (2) Application Layer – the back-end business logic (Node.js, Java, or Python Flask); and (3) Data Layer – database systems like PostgreSQL or MySQL, hosted in the cloud. This architecture separates functionality, making the system more modular, maintainable, and scalable.
3. A relational database such as PostgreSQL is ideal due to its open-source nature and strong ACID compliance. It supports concurrent access and structured data, which is necessary for tracking users, scores, and sessions securely. Cloud storage services like Amazon S3 can handle large image libraries for game assets.
4. Linux employs efficient memory management using paging and virtual memory, preventing memory leaks and optimizing server performance. Garbage collection in Java or Node.js will ensure resources are released after each user session, preventing resource starvation during heavy load.
5. Draw It or Lose It will use RESTful APIs to communicate between clients and the web server. The distributed architecture allows scaling through load balancers, CDN delivery, and redundant databases. Cloud-based solutions (AWS Elastic Beanstalk or Azure App Service) can handle high concurrency while maintaining minimal latency.
6. Security is achieved through HTTPS, TLS encryption, and OAuth2 authentication. Role-based access controls (admin, player, system) will protect data integrity. The Linux environment supports regular patches, firewalls, intrusion detection, and encryption at rest. Sensitive user data such as login credentials will be hashed and salted before storage, ensuring compliance with data protection standards.