

Muhiz Bada

# **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  2.0 | 5/15/25  6/03/25 | Muhiz Bada | Document Revision History:  Executive Summary, Design Constraints, Domain Model completed  Filled in Evaluation section per Project Two requirements; polished for clarity and completeness  Recommendations section completed per Project Three |

**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 currently offers Draw It or Lose It as an Android-only app. CTS will re-implement it as a web-based, multi-platform game service. Our solution:

* Introduce an abstract Entity base class for all domain objects (Game, Team, Player), each with a unique id and name.
* Use the Singleton pattern for GameService so there is exactly one in-memory service managing all games, preventing inconsistent state across users.
* Use the Iterator pattern within GameService, Game, and Team to enforce unique names when adding new games, teams, or players.
* Provide simple, in-line documentation and industry-standard naming conventions for maintainability.

## 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)

**Stateless Web Front-Ends:**

The game UI will be served statelessly; all shared state is held in the singleton GameService.

**Concurrent Access & Thread Safety:**

Multiple users may create or query games simultaneously; future implementations should synchronize or use thread-safe collections.

**Unique Naming Enforcement:**

All addX(name) operations must traverse the existing collection via an iterator before creating a new instance.

**Single In-Memory Service:**

GameService must use private static instance, a private constructor, and public static getInstance() to guarantee a single JVM-wide service.

## [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 class diagram defines:

* Entity (abstract base)
  + Attributes: id: long, name: String
  + Behavior: getId(), getName(), toString()
  + Inheritance: Game, Team, and Player extend Entity
* GameService
  + Manages all Game instances.
  + implements Singleton via a private static instance and private constructor.
  + Provides addGame(name), getGame(id), getGame(name), getGameCount(), and ID generators for games, teams, and players.
  + Uses Iterator pattern to traverse games: List<Game> for uniqueness checks.
* Game
  + Extends Entity.
  + Contains teams: List<Team>.
  + addTeam(name) uses an iterator to enforce unique team names within the game.
* Team
  + Extends Entity.
  + Contains players: List<Player>.
  + addPlayer(name) uses an iterator to enforce unique player names within the team.
* **Player**
  + Extends Entity.
* Object-Oriented Principles
  + Inheritance: Common fields and methods live in Entity.
  + Encapsulation: All collections are private; modifications only via addX() methods.
  + Single Responsibility: Each class has one clear role (service vs. domain object).

**"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 Server (a separate, paid product) can host Apache, Nginx, and other UNIX-based web servers. Its UNIX foundation ensures POSIX compliance and easy porting of server software.  Advantages: macOS’s integrated GUI tools (Server.app) simplify configuration for smaller teams. Its developer ecosystem (Xcode, Homebrew) is robust, and hosting on Mac hardware can be more straightforward for existing iOS/web developers familiar with the Apple ecosystem.  Weaknesses: Licensed macOS Server costs apply per instance, and scaling horizontally (adding many macOS servers) is expensive compared to Linux. Apple’s macOS licensing prohibits running on non-Apple hardware (“Hackintosh” is unsupported), so hardware costs are higher. Limited headless-only support compared to Linux. | | Linux distributions (Ubuntu Server, CentOS, Debian) are the de facto standard for hosting scalable web applications. They support all major web stacks (LAMP, LEMP, MEAN, etc.) and container platforms (Docker, Kubernetes).  Advantages: Zero OS licensing costs; can run on commodity hardware or virtual cloud instances. Upstream community support and long-term support (LTS) releases provide stability. Excellent headless operation; enormous library of packages (apt, yum) for installing web server software, databases, and caching layers. High configurability (firewalls, SELinux/AppArmor).  Weaknesses: Steeper learning curve for administrators unfamiliar with Linux; GUI tools are available but often third-party or less integrated than macOS. Support contracts (e.g., Red Hat) incur costs if enterprise-grade support is needed. | Windows Server (2019, 2022) supports IIS out of the box, integrates tightly with Microsoft SQL Server, Active Directory, and .NET Core/ASP.NET deployments. Windows Subsystem for Linux (WSL) provides limited Unix tooling but is not used in production server setups.  Advantages: If the client’s back-end is built on .NET technologies, Windows Server offers seamless deployment. GUI management via Server Manager and PowerShell scripting for automation. Integration with Azure and other Microsoft services can simplify cloud scaling. Many enterprise companies already have Windows Server expertise in-house.  Weaknesses: Per-core licensing costs can be substantial, especially for large deployments. Windows Server is typically more resource-heavy than Linux equivalents. Less community-driven open-source tooling than Linux; patch cycles and reboots can disrupt uptime more frequently. | Mobile OS (iOS, Android) cannot directly host full web servers; apps rely on a back-end hosted elsewhere. However, lightweight web server frameworks (e.g., Ktor on Android) exist for peer-to-peer or local network discovery, but are impractical for large-scale hosting.  Advantages: Mobility and offline capabilities if paired with local server components; can act as a thin client communicating with a central server via REST/WebSocket. Client-server separation permits a single cloud-hosted back-end to serve mobile users.  Weaknesses: Mobile platforms are not viable for “server-side” hosting at scale. Apple’s App Store and Google Play Store policies restrict persistent background services. Relying on mobile devices for hosting would force the client to build an entirely peer-to-peer network, which is beyond scope and difficult to maintain. |
| **Client Side** | | Modern Safari, Chrome, and Firefox run on macOS with full HTML5, CSS3, and JavaScript support. Electron and native wrappers (e.g., Mac Catalyst) allow desktop apps using web views.  Development Considerations:  • Cost: macOS requires Apple hardware for testing, raising hardware budgets. Xcode is free, but third-party frameworks (e.g., commercial UI kits) may have licensing fees.  • Time: Ensuring cross-browser compatibility means testing on Safari Tech Preview, Chrome, and Firefox. macOS’s WebKit engine can differ in subtle ways from Blink/Chromium, so extra QA time is needed.  • Expertise: Developers must understand WebKit quirks and macOS code signing for any Electron/Ionic wrappers. macOS users expect polished UI/UX and native behaviors (e.g., menu bar integration, dark mode support).  Weaknesses: Browser fragmentation (older Safari versions) can cause rendering inconsistencies; third-party libraries occasionally lack first-class macOS support. Emulators for older OS versions may be needed for regression testing. | | :On Linux desktops, a wide variety of browsers exist (Chromium/Chrome, Firefox, Epiphany). Linux users may also use lightweight window managers and different font stacks, so responsive design must account for DPI scaling and varied screen resolutions.  Development Considerations:  • Cost: Linux itself is free, and most development tools (VS Code, IntelliJ, Sublime Text) are either free or open-source. Docker, Node.js, and other frameworks install directly via package managers at zero cost.  • Time: Testing must cover multiple distributions (Ubuntu, Fedora, Debian), various desktop environments (GNOME, KDE), and multiple browser engines. Linux fragmentation means more QA cycles to verify that WebSocket, SSL certificates, and local storage behave uniformly.  • Expertise: Developers need familiarity with various package managers, environment differences (e.g., different OpenSSL versions), and potential security hardening (AppArmor, SELinux). Application packaging (e.g., Snap, Flatpak, AppImage) might be needed for desktop wrappers, adding extra complexity. | Windows desktop browsers (Chrome, Edge, Firefox, IE11/legacy) have large market share. Legacy Internet Explorer support may be required for enterprise customers.  Development Considerations:  • Cost: Visual Studio Community is free for small teams, but enterprise editions carry licensing fees. Third-party Windows-specific libraries (e.g., .NET journalism) may incur costs.  • Time: Ensuring compatibility with older Windows versions (Windows 10, 11) and legacy browsers (IE11) increases time spent on polyfills, fallbacks, and targeted CSS hacks. Automated testing (Selenium WebDriver on different Windows VMs) is required.  • Expertise: Developers need Windows packaging knowledge (MSIX, installers) if shipping a desktop wrapper, as well as understanding of Windows security settings (UAC, SmartScreen) to avoid install-time warnings. Familiarity with Internet Explorer’s quirks (ActiveX, legacy document modes) may still be necessary if targeting broad enterprise audiences.  Weaknesses: Legacy Windows environments can lag in HTML5/CSS3 compliance. Antivirus or corporate security settings can block WebSocket ports or mixed-content loads. | : Mobile browsers on iOS (Safari/WebKit only; no alternative engines permitted) and Android (Chrome/Chromium webview) both support modern web standards, but with platform-specific limitations (iOS forbids WebSocket over unsecured HTTP, for example).  Development Considerations:  • Cost: Must purchase test devices (iPhone, iPad, multiple Android form factors) or use cloud testing services (e.g., BrowserStack) to validate responsive UI. Development IDEs: Xcode (free) for iOS, Android Studio (free) for Android native wrappers. Hybrid frameworks (Ionic, React Native) may have licensing tiers for enterprise support.  • Time: Responsive design requires media-query QA across dozens of screen densities, pixel aspect ratios, and device-specific quirks (like the iPhone notch safe area). Ensuring that push notifications, offline caching (Service Workers), and GPS integration work uniformly adds significant testing overhead.  • Expertise: Developers must know Swift/Objective-C code signing for iOS or Gradle/Java/Kotlin for Android to wrap the web app into a native shell. Mastery of responsive frameworks (Bootstrap, Tailwind) and mobile-first performance optimizations (lazy loading, minification) is critical.  Weaknesses: App Store and Play Store review processes can delay updates. iOS’s strict background-refresh policies limit certain real-time features. On Android, OEM modifications can break WebView behavior unexpectedly. |
| **Development Tools** | HTML5, CSS3/SCSS, JavaScript/TypeScript, React.js or Vue.js, Node.js for build scripts.  IDEs/Tools: Xcode (for any native wrappers), Visual Studio Code (free), Homebrew for package management, Git/GitHub for version control, Docker for local Linux containers on macOS, NPM/Yarn.  Impact on Team: Development teams with experience in macOS will find Xcode’s integrated UI debugging (Instruments) useful; however, building and packaging Electron apps for macOS requires setting up code signing certificates (Apple Developer Program costs $99/year).  Licensing Costs: Xcode itself is free; Apple Developer Program ($99/year) is required to distribute Mac App Store versions or to code-sign desktop apps. Some mac-specific UI libraries (e.g., macOS Catalyst licenses) might incur moderate fees. | | Same web stack (HTML5, CSS3/SCSS, JS/TS, React/Vue/Angular), with additional emphasis on CLI tooling (Webpack, Babel). Back-end: Node.js, Python/Django/Flask, Ruby on Rails, or Java/Spring Boot.  IDEs/Tools: Visual Studio Code (cross-platform), JetBrains IntelliJ IDEA/WebStorm (commercial), Docker and Kubernetes (both free/community editions), Git/GitLab/GitHub for version control, package managers (apt, yum).  Impact on Team: Linux’s scripting prowess (Bash) simplifies CI/CD pipelines—Jenkins or GitLab CI runners often run on headless Linux VMs. No separate team is needed for Linux development, but sysadmin skill sets (shell scripting, server security, firewall, SELinux) are essential.  Licensing Costs: Almost zero for OS and primary tools; optional enterprise support contracts (Red Hat, SUSE) can be procured if needed (starting ~$350/yr per node). JetBrains IDE licenses are $149–$499 per user per year, depending on edition. | | Web front-end as above; back-end frameworks often use .NET Core (C#), ASP.NET MVC, or ASP.NET Core Web APIs for seamless IIS integration. PowerShell for automation scripts.  IDEs/Tools: Visual Studio Community (free for small teams), Visual Studio Professional (~$1,199 one-time + $799/yr). Windows Subsystem for Linux (WSL) for cross-compatibility testing. IIS Manager, SQL Server Management Studio (SSMS), Azure DevOps for CI/CD.  Impact on Team: If the team has .NET expertise, Windows development speeds up productivity. However, maintaining a mixed-OS environment may require a dedicated Windows specialist vs. a Linux specialist. Building IIS deployment pipelines is straightforward with Azure DevOps.  Licensing Costs: Windows Server Standard costs $972 plus CALs ($45/user). SQL Server Developer edition is free for dev/test; production SQL Server Standard is ~$3,586 per 2-core pack. Visual Studio licenses vary ($45–$250/month per user for Professional or Enterprise subscriptions). | For hybrid apps: Ionic (Angular/React support), React Native (JavaScript), Flutter (Dart), or Progressive Web App (PWA) standards (Service Workers, Web App Manifests).  IDEs/Tools: Xcode (for iOS builds), Android Studio (for Android builds), Visual Studio Code (for cross-platform JS/TS development), Fastlane (automation for signing and deployment), Git/GitHub.  Impact on Team: A single cross-platform team can build both iOS and Android shells if using React Native or Flutter—this consolidates staffing needs, but developers must learn native debugging, code-signing workflows, and platform-specific UI guidelines. Hybrid/PWA teams require less specialized mobile expertise but need to handle browser limitations.  Licensing Costs: Xcode is free, but Apple Developer Program is $99/year. Android SDK is free. Paid plugins (e.g., Ionic Pro, FlutterFlow) may incur monthly fees ($29–$199/mo). Cloud-based testing services (BrowserStack) cost $29–$129/mo per user. |

## Recommendations

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

**Operating Platform**: We recommend Ubuntu Server 22.04 LTS on x86-64 hardware. It carries no OS licensing fees, offers native support for Docker and Kubernetes, and benefits from extensive community maintenance. Whether hosted on-premises or in a public cloud like AWS or Google Cloud, it scales horizontally with minimal cost overhead and integrates smoothly with common CI/CD pipelines.

**Operating Systems Architectures:** Ubuntu Server runs a single, monolithic Linux kernel that includes core services and device drivers, yet supports loadable kernel modules for on-the-fly extensions. User processes reside in isolated memory spaces with strict kernel-space separation. System initialization is handled by systemd, which launches services as units, tracks dependencies, and manages parallel startup. The virtual file system (VFS) abstracts diverse storage back ends into a unified hierarchy under “/,” while procfs and sysfs expose kernel data and hardware topology. Interrupt handling, virtual memory management, and the Completely Fair Scheduler coordinate I/O, paging, and CPU time among containers and native services, ensuring responsive performance under load.

1. **Storage Management**:

A hybrid storage approach balances performance, consistency, and resilience:

* Relational Tier: PostgreSQL on XFS-formatted ext4 volumes in RAID-1 offers ACID transactions for game state, leaderboards, and account data.
* Object Tier: S3 or a self-hosted MinIO cluster stores static assets stock images and logs across multiple availability zones for geo-redundancy.
* Shared Files: An NFS server mounts configuration files and aggregated logs into each container for clustering simplicity and centralized backup.

**Memory Management:** Ubuntu’s demand-paging model pulls only active pages into RAM, minimizing the footprint of thousands of concurrent WebSocket sessions. The slab allocator pools kernel objects network buffers and session structs reducing fragmentation and speeding allocation for game logic and chat messages. Cgroups enforce per-container memory caps, so a runaway process cannot exhaust host memory. Transparent Huge Pages accelerate large contiguous memory operations, such as assembling image buffers for clue rendering, cutting page-walk overhead and reducing latency during rapid clue display.

1. **Distributed Systems and Networks:**

We propose a containerized microservices architecture:

* Service Mesh: Containers for auth, game logic, chat, and scoring register with Kubernetes DNS and communicate over an encrypted overlay network (Calico).
* API Gateway: A front-end proxy terminates TLS 1.3, handles WebSocket upgrades for real-time hints, and routes REST calls.
* Resilience Patterns: Each client library uses exponential backoff, retries, and circuit breakers. Liveness and readiness probes trigger automated failover of unhealthy pods.
* Data Consistency: Critical operations like score submissions use idempotent endpoints and versioned payloads. A message broker (Kafka) buffers chat and scoring events, ensuring no data loss during transient outages.
* Network Design: Deploy within a VPC using private subnets for services and public subnets for the gateway. Security groups restrict traffic to necessary ports. Multi-AZ load balancers distribute client connections and detect instance failures.

1. Security:

Protecting user data end-to-end involves multiple layers:

* Authentication: MFA on login, bcrypt-hashed passwords stored in a Vault-backed secret store.
* Encryption: All traffic client to gateway and inter-service is TLS 1.3 with ephemeral key exchange. Block storage is encrypted at rest via dm-crypt LUKS.
* Container Hardening: AppArmor profiles confine each container’s file and network access. Image builds scan with Clair for known vulnerabilities; hosts run OpenSCAP audits.
* Network Security: Web Application Firewall (ModSecurity) filters malicious payloads. Role-based access control in Kubernetes limits administrative actions. Regular penetration tests and automated patching keep the environment up to date.