

<Draw It Or Lose It>

# **CS 230 Project Software Design Template**

Version 1.2

## Table of Contents

[**CS 230 Project Software Design Template** 1](#_syqppjewmpda)

[**Table of Contents 2**](#_ae08w2wfc2tu)

[**Document Revision History 2**](#_gs2onr95t6qz)

[**Executive Summary 3**](#_qk8jwyyfg0p5)

[**Requirements 3**](#_iqgs988su4ps)

[**Design Constraints 3**](#_8mu80xsq86ly)

[**System Architecture View 3**](#_7uydjyyx8hx8)

[**Domain Model 3**](#_mrc2kbgz44u5)

[**Evaluation 4**](#_fhfqfgw4hmb1)

[**Recommendations 5**](#_p8u7u6ftqpi8)

## [Document Revision History](#_mrhnk1tlrrgt)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 09/27/2025 | Tyler Torrey | Adds Executive Summary, Requirements, Constraints, and UML description. |
| 1.1 | 10/5/2025 | Tyler Torrey | Adds per-environment evaluations and recommendations for the implementation of the application. |
| 1.2 | 10/18/2025 | Tyler Torrey | Adjusts recommendations to include baseline, scale-up, mitigation, and risks (/mitigations) where applicable. |

**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](#_g5meaaet6mk4)

Creative Technology Solutions (CTS) will design and deliver a web-based, multi-platform version of *Draw It or Lose It* for The Gaming Room (TGR). The solution will extend the current Android app into a browser-accessible experience while preserving the core gameplay: timed rounds, progressive image reveals, and team-based guessing. CTS will provide an application service that coordinates active games and exposes endpoints for creating games, managing teams and players, and controlling the round flow. Our solution prioritizes low-latency gameplay, name uniqueness, and scalable, observable operations suitable for multiple concurrent games.

## Requirements

The application must support multiple concurrent games, allow one or more teams per game, and assign multiple players to each team. Game and team names must be unique so users can verify availability at creation time. Only one controlling GameService instance should exist per running process to manage in-memory state, while unique identifiers are issued for games, teams, and players. Gameplay timing follows one-minute rounds with a 15-second steal window, and drawings progressively reveal to full by the 30-second mark. The game should prioritize performance and efficiency so that a broad range of devices can participate reliably.

## [Design Constraints](#_2h3sgoqla5wc)

**Technical Constraints**

1. *Web-based, multi-platform delivery*: Must run reliably in modern browsers and be device-agnostic. To support multi-platform use, keep client resource consumption low. Favor server-side logic with a consistent, versioned JSON API and a responsive UI so clients remain lightweight and interoperable. This will also require significant testing, including as many browsers as possible.
2. *Uniqueness and naming*: Game and team names must be unique. Enforce unique constraints in the datastore and perform checks in the API to prevent duplicates under load. Normalize names consistently (e.g., trim, case-insensitive compare) and return clear error responses (such as HTTP 409 - Conflict) so the client can prompt users to choose another name, if their choice isn’t available.
3. *Real-time round control:* Timers and progressive reveals need low-latency updates. We will use server-side authoritative timers and a pushing system (e.g., WebSocket/SSE) for timely events, with graceful fallback if push is unavailable. The web client should derive countdowns from server timestamps to handle drift and apply simple reconnection/resume logic so users rejoin in the correct state after brief network drops.
4. *Sync with existing functionality:* To achieve cross-platform parity, the legacy Android app will need to be moved to the new, versioned JSON API; if an existing API must be retained, we will add a compatibility layer and accept constraints imposed by its data model, rate limits, and auth. If we must reuse the existing API, we will document any gaps (fields, endpoints) and adjust the MVP accordingly. Presumably, using the legacy API could also be an option. This decision will be finalized during discovery.

**Business Constraints**

1. *Budget / Timeline:* As the hardware requirements arise and software application decisions are completed, there may be increased testing requirements, development fixes, or similar. Any arising factors may cause impacts to the budget and deadline of the project.
2. *MVP Definition*: We should have a clear expectation of what the minimum viable product is, and what can be added to ship later. How much traffic do we expect on the application, and how much scalability is required in the first iteration?

Risks: browser compatibility breadth and API migration complexity may extend testing effort and require scope trade-offs.

## [System Architecture View](#_ujjgmpahldle)

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](#_4ty3b0t1s6cp)

The UML class diagram for The Gaming Room illustrates how the system manages games, teams, and players through a structured object-oriented design. GameService maintains collections of games and ID counters, associating with multiple Game objects through a one-to-many relationship. Each Game contains multiple Team objects, and each Team contains multiple Player objects, reflecting a hierarchical structure. The Entity class defines shared attributes (id and name) and is inherited by Game, Team, and Player, allowing these classes to reuse common properties and methods.

Several object-oriented programming principles are demonstrated in this design. Inheritance is used through the Entity superclass to reduce redundancy and promote code reuse. Encapsulation ensures that each class manages its own data and exposes only necessary operations through methods. Association and aggregation model real-world relationships between games, teams, and players, making the structure intuitive and scalable. Finally, the singleton pattern applied in GameService centralizes control of game state, ensuring consistency and efficient resource management across the system.

**"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](#_5zapbxoegc4h)

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** | Can host via Nginx/Apache/Jetty, but rarely used in production.  Strengths: solid UNIX base; good developer tooling.  Weaknesses: Apple licensing ties OS to Apple hardware; limited cloud/server options; not cost-effective at scale.  Licensing: bundled with Mac hardware (no separate server license). | Industry default for web hosting (on-prem & all clouds).  Strengths: stable, performant, first-class Docker/Kubernetes, huge ecosystem, great $/performance.  Weaknesses: steep technical learning curve, some limitations with language / technology hosting  Licensing: GPL - no per-server licensing fee. | Fully capable (IIS/.NET; Java works too).  Strengths: Native Active Directory/SSO integration; GUI admin tools; strong vendor support.  Weaknesses: license cost, heavy footprint; container story not as lean as Linux.  Licensing: per CPU core; higher TCO than Linux. | Not a server platform (used as clients). n/a for hosting. |
| **Client Side** | Mac clients likely use Safari/Chrome/Firefox;  ensure responsive UI, WebGL performance;  test Safari quirks (e.g., date inputs, video / audio codecs).  Cost/Time/Expertise: front-end web + cross-browser testing.  Some specific expertise recommended for WebKit testing. | Linux clients likely use Chrome/Firefox;  Ensure compatibility with GPU / WebGL / audio codecs / fonts across common distros.  Cost/Time/Expertise: Same responsive app, add a pass on popular distros. | Windows clients likely use Edge/Chrome/Firefox;  Most web functions supported out-of-the-box  Cost/Time/Expertise: same responsive web app; little specialty needed. | Mobile clients use Safari & Chrome;  Touch targets, viewport, virtual keyboard, orientation, network variability; we should consider a PWA (installable web app).  Cost/Time/Expertise: performance tuning; device lab or cloud testing.  iOS vs Android differences need specific testing and tuning. |
| **Development Tools** | VSCode - Typescript/ HTML / CSS for transferable + responsive UI.  VSCode Java Plugin for backend development.  Testing - BrowserStack for Webkit testing- free trials; paid tiers available.  Xcode will be necessary for testing iOS integrations if a mobile app is created (as opposed to mobile web browser). | VSCode - Typescript/ HTML / CSS for transferable + responsive UI.  VSCode Java Plugin for backend development.  Testing - BrowserStack for Webkit / Chromium testing- free trials; paid tiers available. | VSCode - Typescript/ HTML / CSS for transferable + responsive UI.  VSCode Java Plugin for backend development.  Testing - BrowserStack for Webkit / Chromium testing- free trials; paid tiers available. | Mobile browser development should be tested on real devices, and using XCode / Android Studio  BrowserStack for device coverage |

## 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**:

Choose Linux (Ubuntu LTS/Amazon Linux) as the server platform to meet TGR’s needs for multi-platform reach, low latency, and cost efficiency.

* Baseline: a single Linux host runs the app and proxy for an MVP.
* Scale-up: containerize and add hosts (ECS/K8s/VMs) without changing client contracts (HTTPS/WSS).
* Risks: limited in-house Linux ops;
  + mitigation: managed DB, automated patching, and scripted builds (CI/CD).

1. **Operating Systems Architectures**:

Organize the system as isolated processes/containers on Linux: an edge proxy (TLS + WebSocket upgrade), a stateless app (REST + WS), and a database on a separate node/managed service. Use cgroups for per-service CPU/RAM limits, non-root users, and AppArmor/SELinux; prefer non-blocking I/O (epoll) in the proxy/app for steady latency.

* Baseline: all on one host;
* Scale-up: add identical app instances behind the proxy.
* Risks: resource contention;
  + mitigation: limits/quotas and health-based restarts.

1. **Storage Management**:

Match storage to access patterns: game state in PostgreSQL (unique indexes for name uniqueness), and drawings/static assets served by the proxy.

* Baseline: local disk + Linux page cache (simple, fast).
* Scale-up: move assets to S3+CDN (or EFS if shared POSIX is required) with cache-busting via file versioning.
* Risks: bandwidth spikes and stale content;
  + mitigation: CDN caching headers, object versioning, and nightly DB/asset backups with restore drills.

1. **Memory Management**:

Keep gameplay render/decode responsive by controlling server and client memory pressure. Server: bound JVM heap with G1GC, reuse serializers, cap WebSocket buffers, and let Nginx leverage the Linux page cache for assets.

* Risks: GC pauses/leaks and buffer bloat;
  + mitigation: right-size heaps/queues, GC/heap telemetry with alerts, and leak checks in staging.

1. **Distributed Systems and Networks**:

Use REST over HTTPS for CRUD and WebSockets (WSS) for real-time ticks/reveals; fallback to SSE/short-poll when WS is blocked.

* Baseline: one app instance keeps a room->connections map and broadcasts directly.
* Scale-up: multiple instances coordinate via Redis pub/sub (server-to-server fan-out), with idempotent writes, timeouts, retries with jitter, and circuit breakers.
* Risks: mobile drops and partial outages;
  + mitigation: auto-reconnect with backoff, heartbeat/idle timeouts, and state rehydrate on reconnect.

1. **Security**:

Protect user data in transit with TLS (HTTPS/WSS) + HSTS and strict CORS/Origin checks; at rest with disk/database encryption and secret management. Enforce RBAC using Authenticator/Authorizer/Principal and @RolesAllowed; hash passwords with Argon2id; throttle logins; validate and size-limit all inputs/WS frames.

* Risks: XSS/CSRF, brute force, and key leakage;
  + mitigation: CSP + httpOnly/SameSite cookies (or short-lived JWTs), rate limits, secret rotation, and audit/alerts on auth anomalies.