

Draw It or Lost It

# **CS 230 Project Software Design**

Version 1.2

## Table of Contents

[**CS 230 Project Software Design** 1](#_Toc205128184)

[Table of Contents 2](#_Toc205128185)

[Document Revision History 2](#_Toc205128186)

[Executive Summary 3](#_Toc205128187)

[Requirements 3](#_Toc205128188)

[Design Constraints 3](#_Toc205128189)

[System Architecture View 4](#_Toc205128190)

[Domain Model 4](#_Toc205128191)

[Evaluation 6](#_Toc205128192)

[Recommendations 9](#_Toc205128193)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 07/20/2025 | Rebecca Davis | Initial design document |
| 1.1 | 08/03/2025 | Rebecca Davis | Updated Evaluation Section |
| 1.2 | 08/17/2025 | Rebecca Davis | Updated Recommendations Section |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room has requested development support for a web-based team guessing game called *Draw It or Lose It*. The game is inspired by the classic game *Win, Lose or Draw*. The *Draw It or Lose It* game relies on timed visual clues from a stock image library to help teams guess phrases, titles, or things. The goal is to deliver a scalable, intuitive, and engaging web-based version of The Gaming Room’s Android app that supports timed rounds, team-based competition, and unique player and game identities.

The Gaming Room team has limited experience setting up development environments and needs guidance on how to build, test and deploy the overall system effectively.

## Requirements

Business Requirements

* Support one or more teams per game
* Allow multiple players per team
* Validate uniqueness of game and team names
* Display a series of images at fixed intervals during each round
* Enable a 15-second bonus guessing opportunity for other teams after time expiration
* Maintain game state, timing, and round progression
* Provide administrative utilities for game creation and environment setup

Technical Requirements

* Web-based responsive UI for desktop and mobile devices
* Real-time updates for gameplay progression
* Stateless server-side handling with synchronized round timing
* Secure access control for player login and team selection
* Development tools and deployment environments need to be streamlined and clearly documented for Gaming Room staff.
* The system should allow for future enhancements and increasing traffic loads without performance degradation.

## [Design Constraints](#_2et92p0)

* Only one active game at a time, enforcing each game to be unique by using a unique game identifier and in-memory singleton instance
* Game and team names must be checked against current session memory or using a database to avoid duplication
* Use a containerized environment for both development and hosting for a streamlined launch
* Preloaded stock images must be optimized for web delivery. Utilize a geographically distributed network for these files. Images should also be compressed for faster delivery to clients.
* Adopt Agile best practices to be utilized with requirements analysis, development, testing, and deployment.

## [System Architecture View](#_ilbxbyevv6b6)

## [Domain Model](#_8h2ehzxfam4o)

At the foundation is the Entity base class, which encapsulates the shared attributes id and name. This promotes code reuse and simplifies management of unique identification across the system. The Game, Team, and Player classes all inherit from Entity, reinforcing their role as distinct attributes and methods, yet structurally similar entities within the game. The Game class handles a single instance of play, managing one to many Teams. Each Team holds one to many Players, which captures the game's layered relationships and supporting the requirement for multi-team, multi-player participation.

To ensure only one game instance resides in memory at a time, the design features a GameService class implemented as a singleton. In the GameService class the getInstance() method is used to get the singleton instance. The GameService class provides methods for creating games, assigning teams and players, and managing globally unique identifiers. The GameService holds one to many Games. The singleton pattern implemented in the GameService fulfills the client’s constraint of a single active game session, which offers a secure way to maintain game state. The SingletonTester class is used to validate the singleton behavior of the GameService class is correctly enforced during development. The ProgramDriver class serves as the entry point, containing the main() method that orchestrates system startup and gameplay initiation. The ProgramDriver class also utilizes the SingletonTester class to validate the singleton behavior and game state.

Together, these classes demonstrate key object-oriented programming principles:

* Inheritance through the shared Entity base class
* Encapsulation by allowing each class to manage its own data and behavior
  + Team manages its Players
  + Game manages its Teams
  + GameService manages the Games
* Single Responsibility Principle where each class has a well-defined purpose.
  + Game, Team, and Player represent game entities
  + GameService manages the overall game coordination
  + ProgramDriver runs the game
  + SingletonTester runs the testing logic
* Singleton Pattern is implemented in the GameService

**"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)

| **Development Requirements** | **macOS** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | - **Characteristics**:  Unix-based, reliable for lightweight dev servers.  - **Advantages**: Native Docker and UNIX tools, good for prototyping. Strong security and stability.  - **Weaknesses**: Rarely used for production web hosting; limited scalability. Deprecated server features. | - **Characteristics**: Most common choice for web hosting and cloud deployment.  - **Advantages**: Lightweight, scalable, secure, excellent container support. Highly secure and stable.  - **Weaknesses**: Less intuitive for beginners; limited enterprise support and depends on distribution. | - **Characteristics**: Stable for enterprise and .NET applications.  - **Advantages**: Familiar UI and wide user base, extensive software and hardware compatibility, IIS support, SQL Server integration, Docker available via WSL2.  - **Weaknesses**: Higher resource footprint; less common in cloud-native stacks. Resource intensive and frequent reboots required. | - **Characteristics**: Not applicable for hosting.  - **Advantages**: Not applicable for hosting.  - **Weaknesses**: Not applicable for hosting. |

The operating platforms, macOS, Linux, and Windows, support server-based deployment methods, though each deployment method varies. Linux and Windows both offer robust environments for hosting web applications, including containerized deployments using Docker, which enhances portability and scalability across platforms. Linux distributions like Ubuntu or CentOS are especially popular for Docker-based hosting due to their lightweight architecture and zero licensing cost, making them ideal for The Gaming Room’s budget-conscious strategy. Windows Server supports Docker as well but incurs licensing fees, typically starting around $1,000 for the Standard edition plus additional CALs (Client Access Licenses). macOS Server, while technically capable of hosting services, is deprecated and not recommended for scalable web deployment. Mobile platforms, like iOS and Android, are not viable for hosting but can serve as client endpoints. Overall, Linux offers the most cost-effective and flexible solution for server deployment, especially when leveraging Docker containers.

| **Development Requirements** | **macOS** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Client Side** | - **Cost**: Moderate to high. Requires Apple hardware (MacBook or iMac), which is more expensive than typical Windows/Linux machines. Development tools like Safari DevTools, Xcode, Visual Studio Code, Docker Desktop are free.  - **Time & Expertise**: Setup time depends on familiarity with Apple’s ecosystem. Safari-specific quirks may require extra testing. Generally efficient for experienced developers. Requires knowledge of Apple’s Human Interface Guidelines and Safari-specific behaviors. Familiarity with Xcode and Swift is optional but helpful.  - **Considerations**: Safari uses the WebKit engine, which may behave differently from Chromium-based browsers. Testing for layout and JavaScript quirks is essential. Safari DevTools are native options; Visual Studio Code and Chrome are also widely used. Apple’s Human Interface Guidelines (HIG) influence design expectations: clean, minimal, and intuitive interfaces. | - **Cost**: Low. Open-source OS with free development tools like Firefox DevTools, Visual Studio Code, Docker Desktop. No licensing costs.  - **Time & Expertise**: Longer initial setup for less experienced users. Requires manual configuration and testing across distributions. Efficient for seasoned developers. Developers should be comfortable with command-line tools, package managers, and browser testing. Strong debugging and system knowledge is beneficial.  - **Considerations**: Firefox and Chromium-based browsers are dominant, but behavior may vary across distributions and window managers. Visual Studio Code and CLI tools are preferred; debugging often relies on browser dev tools. Developers have full control over environment setup, but must account for distro-specific quirks. Linux allows deep tuning of resource usage, but requires manual configuration. | - **Cost**: Moderate. Most developers already have access to Windows machines. Tools like Visual Studio Code, Edge DevTools, Docker Desktop are free. OS license may be bundled with hardware.  - **Time & Expertise**: Fastest setup due to widespread familiarity. Minimal configuration needed for browser-based development. Basic to intermediate web development skills are sufficient. Familiarity with Windows-specific browser behaviors in Edge are useful.  - **Considerations**: Edge, Chrome, and Firefox are common. Visual Studio Code, and Edge DevTools offer robust debugging and UI inspection. Windows devices vary widely in resolution—ensure your app handles high-DPI displays gracefully. Users expect Fluent Design elements like shadows, animations, and responsive layouts. | - **Cost**: Low to moderate cost depending on devices needed for testing.  - **Time & Expertise**: Requires responsive design and touch optimization.  Expertise in responsive HTML/CSS/JS. Familiarity with mobile UX, testing on emulators and real devices.  - **Considerations**: Use flexible layouts, media queries, and scalable assets to ensure the site adapts to various screen sizes and orientations. Frameworks like Bootstrap, Tailwind CSS, or CSS Grid/Flexbox are essential. Design UI elements for finger-friendly interaction—larger buttons, swipe gestures, and minimal hover reliance. Avoid small tap targets and ensure smooth scrolling. Mobile users often face slower networks and limited device resources. Optimize images, minify CSS/JS, and use lazy loading to reduce initial load time. Avoid excessive animations, background polling, or heavy scripts that drain battery or slow down performance on mobile devices. |

To ensure broad compatibility across web browsers and mobile devices, the application development process should prioritize responsive design using flexible layouts and media queries, implement cross-browser testing to catch rendering inconsistencies, and optimize performance for limited mobile resources. Developers should adhere to web standards (HTML5, CSS3, ECMAScript), use progressive enhancement to support older browsers, and ensure touch-friendly interfaces with scalable UI elements. Including proper viewport settings, minimizing dependencies, and leveraging feature detection, available in the third-party library Modernizr, which helps maintain consistent behavior across diverse platforms.

| **Development Requirements** | **macOS** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Development Tools** | - **Languages**: Java, Swift (for iOS testing), JavaScript, Python. Web development using HTML, CSS, JavaScript, TypeScript.  - **IDEs**: Xcode, IntelliJ, Visual Studio Code.  - **Other Tools**: Homebrew, Terminal, Docker Desktop, Git. | - **Languages**: Java, Node.js, Python, C++, C#. Web development using HTML, CSS, JavaScript, TypeScript.  - **IDEs**: Visual Studio Code, IntelliJ, Eclipse, JetBrains suite.  - **Other Tools**: Git Bash, Docker, Docker Desktop, nginx, Apache. | - **Languages**: Java, C#, JavaScript, Python. Web development using HTML, CSS, JavaScript, TypeScript.  - **IDEs**: Visual Studio, Visual Studio Code, IntelliJ.  - **Other Tools**: PowerShell, Git Bash, WSL2, Docker Desktop. | - **Languages**: Kotlin, Java (Android); Swift, Objective-C (iOS); JavaScript (web view).  - **IDEs**: Android Studio, Xcode.  - **Other Tools**: Emulators, ADB, TestFlight, Gradle, WebView testing. |

To build cross-platform web applications deployable on macOS, Linux, Windows, and mobile devices, developers typically use HTML5, CSS3, JavaScript, and optionally TypeScript for frontend logic, alongside backend languages like Node.js, Python, or Java. IDEs such as Visual Studio Code (free), IntelliJ IDEA (paid and free versions), and platform-specific tools like Xcode (macOS) and Android Studio (multi-platform) support development and testing. Supporting diverse platforms requires thorough cross-browser and device testing, often involving tools like Docker, Selenium, and Edge/Chrome/Firefox/Safari DevTools. These technical requirements can impact team structure. Larger projects may need specialized frontend, backend, mobile, and QA teams to manage complexity and ensure compatibility. While many tools are free or open-source, some (like IntelliJ Ultimate or enterprise Docker plans) involve licensing costs, which should be factored into project budgeting.

## Recommendations

**Operating Platform**

Ubuntu Server is the recommended operating platform for *Draw It or Lose It*, offering a stable, secure, and widely supported environment for containerized microservices. Its compatibility with Docker, Kubernetes, and modern CI/CD tooling makes it an ideal foundation for scalable cloud-native deployment. Ubuntu’s long-term support (LTS) releases provide predictable update cycles and enterprise-grade reliability, ensuring consistent performance across development, staging, and production environments.

The game’s backend should be deployed on Ubuntu-based Kubernetes clusters, either self-managed or provisioned via AWS EKS, Azure AKS, or Google Kubernetes Engine. These clusters will host services such as matchmaking, session management, and real-time communication, with traffic routed through NGINX ingress controllers and Cloudflare CDN for global reach. Ubuntu’s extensive package ecosystem and active community support accelerate development and troubleshooting, while its compatibility with monitoring tools ensures full observability across services.

By standardizing on Ubuntu, *Draw It or Lose It* benefits from a mature, secure, and developer-friendly platform that supports rapid iteration and operational resilience.

**Operating Systems Architectures**

Ubuntu’s architecture is well-suited for both server-side orchestration and client-side development workflows. On the server, Ubuntu Server LTS provides a hardened Linux kernel, enabling secure container isolation and resource governance. Its native support for multiple administration toolssimplifying configuration management and bootstrapping across cloud environments.

Each node in the Kubernetes cluster should run a minimal Ubuntu image optimized for container workloads, reducing overhead and improving boot times. Ubuntu offers lightweight alternatives for edge deployments or local testing. For enhanced security, Ubuntu supports unattended upgrades, rootless containers, and read-only file systems, minimizing an attack surface and enforcing immutability.

On the client side, while Ubuntu is not the primary OS, its desktop variant can serve as a robust development environment. This ensures consistency in tooling and build pipelines across the engineering team.

**Storage Management**

Ubuntu provides a flexible and secure foundation for managing both structured and unstructured data in *Draw It or Lose It*. The backend should use PostgreSQL running on Ubuntu for transactional data such as user profiles, game history, and analytics. Ubuntu’s native support for ZFS, LVM, and ext4 file systems allows for fine-grained control over volume management, snapshots, and performance tuning.

Unstructured assets, such as drawing images, avatars, and chat logs, should be stored in cloud-based object storage, with Ubuntu handling the integration. The Redis caching layer, also running on Ubuntu, can accelerate access to frequently used data and reduce database load.

To ensure durability and compliance, Ubuntu supports automated backup scripts, archival, and encryption at rest. Storage monitoring can be implemented, providing real-time visibility into disk health and throughput. These capabilities make Ubuntu a reliable and extensible platform for managing game data at scale.

**Memory Management**

Ubuntu’s memory management capabilities are well-suited for both high-performance server workloads and resource-constrained environments. On the backend, services should run in containers with defined memory limits and OOM (Out-of-Memory) policies, leveraging Ubuntu’s support to enforce stability. Tools can be used to provide granular insights into memory usage, enabling proactive optimization.

The in-memory data stores Redis should be deployed on Ubuntu nodes to manage data like active sessions, game timers, and matchmaking queues. These services benefit from Ubuntu’s low-latency networking stack and predictable memory allocation behavior. For stateless microservices, memory usage can be minimized through streaming architectures, buffer pooling, and garbage collection tuning in languages like Go or Node.js.

On the client side, while Ubuntu is not the target OS, its desktop environment can be used for profiling and debugging memory-intensive features. This ensures that memory optimization strategies are validated across platforms before deployment. By leveraging Ubuntu’s mature memory management ecosystem, *Draw It or Lose It* can maintain responsiveness and reliability under varying load conditions.

**Distributed Systems and Networks**

To enable seamless communication across platforms, *Draw It or Lose It* should adopt a microservices architecture deployed via Docker containers and orchestrated with Kubernetes. Each core function, such as game logic, user authentication, session management, and leaderboard tracking, can run as an independent service, allowing for modular development and horizontal scaling. These services will communicate through RESTful APIs for standard interactions and WebSockets for real-time gameplay, ensuring low-latency drawing and guessing experiences.

A service discovery mechanism like Consul or Kubernetes-native DNS will allow services to locate each other dynamically, even as containers scale or shift. To maintain reliability, the system should implement load balancing, health checks, and auto-scaling policies to handle traffic spikes and recover from outages.

Network resilience is critical. The system should include retry logic, circuit breakers, and timeout strategies to gracefully handle intermittent connectivity issues. For distributed data consistency, replication and eventual consistency models can be used, especially in multi-region deployments. These strategies ensure that *Draw It or Lose It* remains responsive and synchronized across devices, even in the face of partial failures or network disruptions.

**Security**

Security is a foundational requirement for *Draw It or Lose It*. All data in transit should be encrypted using TLS 1.2+, while sensitive data at rest, such as passwords, session tokens, and personal information, should be protected using AES-256 encryption. The system should enforce OAuth 2.0 or OpenID Connect for secure, token-based authentication across web, mobile, and console clients.

Role-Based Access Control (RBAC) should be implemented to restrict access to sensitive operations based on user roles. Input validation and output sanitization must be applied consistently to prevent SQL injection, cross-site scripting (XSS), and other common vulnerabilities. Rate limiting, IP throttling, and CAPTCHA mechanisms can help mitigate brute-force and denial-of-service attacks.

For secure inter-service communication, internal APIs should be protected using mutual TLS or API keys managed through a centralized secrets manager like Vault or AWS Secrets Manager. All authentication and authorization events should be logged and monitored, enabling real-time threat detection and forensic analysis. Regular penetration testing, code audits, and dependency scanning should be part of the development lifecycle to ensure ongoing security compliance.