

*Draw It or Lose It*

# **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 | 09-26-2025 | Elle Ward | Evaluations |

**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 has asked for a web-based version of their Android game Draw It or Lose It. The new system must support multiple platforms, enforce uniqueness for game and team names, and guarantee that only one game service instance exists in memory. To meet these requirements, the design uses object-oriented principles and industry-standard design patterns. The singleton pattern ensures centralized game service management, and the iterator pattern is used to validate uniqueness for games, teams, and players. This approach provides a maintainable foundation for scaling the game across distributed environments and multiple platforms.

## Requirements

* Support multiple teams and players per game.
* Enforce unique names for games and teams.
* Only one instance of GameService in memory.
* Web-based distributed application with multi-platform access.
* Object-oriented design with reusable, maintainable code.

## [Design Constraints](#_2et92p0)

The design introduces several constraints:

* Concurrency: Multiple players may attempt to create teams or games simultaneously. Synchronization and validation are required.
* Scalability: The design must scale with increased users without major changes.
* Cross-Platform Support: The application must support Windows, Mac, Linux, and mobile clients.
* Resource Management: Singleton implementation ensures only one service instance, reducing memory overhead.

## [System Architecture View](#_ilbxbyevv6b6)

The new version of Draw It or Lose It will be organized into three main parts, each with a clear role:

* Players’ Devices (Client Layer): This is what players will see and use, whether on a phone, tablet, or computer. It includes the game screens, buttons, and the overall user experience.
* Game Engine (Application Layer): This is the “brains” of the system. It lives on secure servers. It manages the rules of the game, keeps track of turns and scores, and makes sure everyone is playing in sync, no matter what device they are on.
* Data Storage (Database Layer): This is where important information is saved, such as player accounts, team names, and game history. It ensures that progress is not lost and that names stay unique.

This approach means the Gaming Room staff won’t need to worry about how different devices connect because the architecture is designed to handle that automatically.

## [Domain Model](#_8h2ehzxfam4o)

The UML diagram shows the structure of the game system and how its parts connect. At the top is the Entity class, which provides common fields (id and name) that are shared by all other classes. This prevents duplication and makes sure each object in the system has a unique identifier.

From there, the Game, Team, and Player classes inherit from Entity. A Game contains one or more Teams, and each Team contains one or more Players. This hierarchy matches the business rules: players must belong to a team, and teams must belong to a game.

The GameService class manages all games. It is a singleton, meaning only one instance can exist at any given time. This prevents conflicts and ensures centralized control. The class also uses iterators to check for duplicate names when adding new games, teams, or players.

Finally, the ProgramDriver class serves as the entry point for running the application, while SingletonTester verifies that the singleton behavior is working correctly.

This design demonstrates several object-oriented principles:

* Inheritance: Shared fields in Entity reduce redundancy across classes.
* Encapsulation: Private fields, with controlled access through methods.
* Abstraction: The Entity class generalizes the behavior of all game objects.
* Polymorphism: Each class overrides toString() to provide specific output.

**"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 is functional for small-scale or internal web hosting but is not widely used for enterprise-level deployments. It supports common web technologies such as Node.js, Java, and Nginx, but Apple hardware is required and more expensive than other platforms. Since Apple discontinued most macOS Server features, it is now better suited for development and testing environments rather than production systems. | Linux remains the industry standard for web-based hosting because of its stability, scalability, and open-source flexibility. It supports all major server frameworks and languages, including Node.js, Python, Java, and Go, and offers strong compatibility with cloud providers such as AWS, Azure, and Google Cloud. Its primary strengths are cost efficiency, high performance, and strong security. The main drawback is that it requires some administrative expertise to manage effectively. | Windows Server provides a stable environment for applications built in .NET or for teams using Microsoft technologies. It supports IIS, Node.js, and Java, making it flexible for mixed-language environments. It offers strong security and easy Active Directory integration. However, it involves higher licensing and maintenance costs compared to Linux systems. | Mobile devices are not intended to host server components; they serve only as client endpoints. Instead, applications running on mobile clients rely on remote servers deployed on Linux or Windows platforms. The back-end components are hosted in the cloud, and mobile devices connect to them through APIs. |
| **Client Side** | Supporting Mac users mainly involves ensuring compatibility with Safari and Chrome browsers and maintaining consistent performance and accessibility across different Apple devices. Because modern web standards allow a single responsive design to serve all platforms, development effort remains moderate, and costs are manageable. Expertise in macOS testing tools and UI guidelines improves the overall user experience. | Developing for Linux focuses on compatibility with Firefox and Chrome. Since Linux users typically access web apps through browsers, most compatibility issues can be addressed through standardized web technologies. Costs and development time are like Mac, though testing on multiple Linux distributions can add slight complexity. | Windows clients require testing across multiple browsers, including Edge and Firefox, and support for a wide range of hardware configurations. Additional attention is needed for display scaling, GPU acceleration, and input devices. While development costs are comparable to Mac and Linux, more testing time may be required due to enterprise and hardware variations. | Mobile clients must be designed with a mobile-first approach that focuses on touch-friendly interfaces, fast load times, and cross-platform responsiveness. Testing is needed on both iOS and Android browsers. Implementing Progressive Web App (PWA) features can reduce the need for native development, saving time and cost while maintaining a consistent experience across devices. |
| **Development Tools** | Development on macOS commonly uses JavaScript or TypeScript for front-end frameworks such as React or Vue, with Node.js or Java for back-end services. Tools include Visual Studio Code, WebStorm, IntelliJ IDEA, Docker Desktop, Postman, and Git. This setup provides a complete and modern environment for full-stack web development and testing. | Linux supports the same development stack as Mac but provides additional support for DevOps.  Developers often use VS Code, JetBrains IDEs, Docker or Podman, Kubernetes, and GitHub or GitLab CI/CD pipelines. Linux provides the most efficient environment for continuous integration and deployment due to its strong container and automation support. | Windows supports all major web technologies, including TypeScript, .NET, Java, and Node.js. Visual Studio and Visual Studio Code are the main IDEs, alongside Docker Desktop. Windows is suitable for organizations already using Microsoft products and for teams that need .NET support. | Developers can build mobile clients using the same responsive web app codebase, with PWA functionality for offline access. Native tools such as Android Studio and Xcode are available for development. This allows one development team to manage both web and mobile experiences. |

## 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**: Draw It or Lose It should run on a Linux server, ideally Ubuntu LTS hosted on AWS, Azure, or Google Cloud. Linux is stable, secure, and efficient. Using Docker containers managed by Kubernetes gives the system easy scaling and updates without downtime. This setup supports all client devices—Windows, macOS, Android, and iOS—through a single backend accessed over HTTPS.
2. **Operating Systems Architectures**: The system should follow a three-tier structure: an edge layer for routing and security, an application layer for containerized services, and a data layer for storage. The app layer can run Java (Dropwizard, Spring Boot) or Node.js services managed by Kubernetes. REST handles normal communication, while WebSockets enable real-time features such as round timers and instant updates. The design keeps services isolated, making them easier to update or replace.
3. **Storage Management**: Use multiple storage tools, each for a specific job:
   1. **PostgreSQL** for user data, teams, and game results.
   2. **Redis** for quick in-memory caching of active sessions.
   3. **Cloud object storage** (S3, Azure Blob, or Google Cloud Storage) for image files.
   4. **Kafka or RabbitMQ** for game events and analytics.
4. **Memory Management**: Each container should have strict CPU and memory limits. Java services can use G1 or ZGC garbage collection; Node.js can limit heap size. Only live game data should stay in memory, cleared automatically after each round. Results go straight to PostgreSQL. Connection pools, timeouts, and backpressure prevent crashes when traffic spikes.
5. **Distributed Systems and Networks**: The app uses REST APIs for regular communication and WebSockets for real-time updates. A load balancer routes requests and replaces unhealthy instances. Kafka or RabbitMQ manages communication between services for game events and state updates. Multi-region hosting with replicated databases and global load balancing keeps the game online even during outages.
6. **Security**: All traffic runs through TLS 1.3 with encryption at rest and in transit. OAuth 2.1 or OpenID Connect handles authentication with short-lived JWT tokens. Passwords use Argon2id hashing, and secrets stay in a managed cloud vault. Regular scans and OWASP standards keep the system hardened against attacks.