

Draw It or Lose It Software Application

# **CS 230 Project Software Design Template**

Version 1.0

## Table of Contents

[**CS 230 Project Software Design Template** 1](#_heading=h.2tacjchlqogt)

[**Table of Contents 2**](#_heading=h.2nmc3jxvwaea)

[**Document Revision History 2**](#_heading=h.c2zfi8qxw2qd)

[**Executive Summary 3**](#_heading=h.xbrwwvi0urs2)

[**Requirements 3**](#_heading=h.7puqnj57jjek)

[**Design Constraints**](#_heading=h.kda6pgrpaw54) **4**

[**System Architecture View**](#_heading=h.1il7ndv4u4hh) **5**

[**Domain Model**](#_heading=h.yg2nc5xage1l) **5**

[**Evaluation**](#_heading=h.w7plx4prlgr) **7**

[**Recommendations**](#_heading=h.bo6vcmf9g57m) **9**

## [Document Revision History](#_heading=h.ilu5iz139dwb)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 03/24/2025 | Malakai Magharing | Initial draft of software design document for “Draw It or Lose It”. |
| 1.1 | 04/06/2025 | Malakai  Magharing | Revisions made to the Evaluation table. |
| 1.2 | 4/20/2025 | Malakai Magharing | Reviewed recommendations section and finalized Software Design Document for “Draw It or Lose It”. |

## [Executive Summary](#_heading=h.71z861g7lesq)

The Gaming Room currently has a successful Android-based application called **Draw It or Lose It**, which allows teams to compete in guessing puzzles. They want to expand this game into a **web-based, distributed environment**, enabling multiple teams to play from a variety of devices.

The application must guarantee that:

1. Only **one** instance of the game service exists in memory at any time (via a **Singleton** design pattern).
2. The system can manage **multiple games**, each containing one or more **teams**, with each team having multiple **players**.
3. **Unique** names for games, teams, and players must be ensured.
4. The application is structured with a **domain model** that effectively captures relationships and enforces constraints (e.g., team uniqueness within a game, player uniqueness within a team).

Given these requirements, we propose an **object-oriented** architecture in Java that relies on widely used patterns (Singleton for GameService, iteration logic to enforce name uniqueness). This document outlines the design constraints, a domain model overview, and an evaluation of various operating platforms. Finally, we present recommendations for The Gaming Room to move forward confidently with the new web-based solution.

## Requirements

**Business Requirements**

* The Gaming Room wants to reach a broader audience by making the game available on platforms beyond Android.
* The user experience should remain consistent: multiple teams, four rounds, guess-based scoring, etc.

**Technical Requirements**

* The application must be **web-based** and eventually support **multiple platforms** (Windows, macOS, Linux, mobile).
* Only **one** instance of the game service should exist at runtime (Singleton).
* **Unique IDs** for Game, Team, and Player.
* Names (Game, Team, Player) must be **unique** to prevent conflicts.
* Code must follow good **OOP** and **best practices** for maintainability.

## [Design Constraints](#_heading=h.gw9l7n1jcka8)

Developing a game application in a **web-based distributed environment** introduces specific constraints:

1. **Single Instance in Memory**
   * By using a **Singleton** (GameService), the application ensures consistent state management across all connected clients.
   * In a distributed environment, scaling this pattern beyond a single JVM or server requires additional design considerations (e.g., caching, synchronization).
2. **Name Uniqueness**
   * Games, teams, and players must be searchable and validated to ensure no duplicates.
   * This introduces constraints in how we store and retrieve data (lists, maps, or databases), and how we iterate to confirm uniqueness efficiently.
3. **Scalability and Concurrency**
   * Multiple users could join or create teams simultaneously, so the data structures need to handle concurrent requests.
   * Potential concurrency hazards (e.g., race conditions, data inconsistencies) need to be mitigated if the system scales up.
4. **Extensibility**
   * The design must be flexible enough to add new features (like scoring, leaderboards, time-based events) without major refactoring.
   * Using inheritance (Entity as a base class) and recognized design patterns helps meet that need.
5. **Distribution and Network Latency**
   * In a future multi-server architecture, calls between distributed nodes might introduce latency or partial failures, which the design must handle gracefully.
   * The current local design is an initial step toward that distributed model.

## [System Architecture View](#_heading=h.spdjdhpy22my)

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](#_heading=h.28vyi614z3b8)

The **Domain Model** for *Draw It or Lose It* includes the following main classes:

1. **Entity**
   * A **base class** containing an id and a name, from which Game, Team, and Player all inherit.
   * Enforces consistency and prevents code duplication.
2. **Game** (extends Entity)  
   * Represents a distinct game session.
   * Contains a list of Team objects.
   * Enforces **unique** team names within the same game.
   * Common behaviors include methods like addTeam().
3. **Team** (extends Entity)  
   * Represents a specific team within a Game.
   * Maintains a list of Player objects.
   * Ensures **unique** player names within its team.
   * Methods include addPlayer().
4. **Player** (extends Entity)  
   * Individual players, each with a unique ID and name.
   * Could be extended further to track scores or personal stats.
5. **GameService** (Singleton)  
   * Coordinates the creation and retrieval of games, teams, and players.
   * Manages unique ID counters for each type of object.
   * Ensures there is only **one** instance of GameService in memory to maintain global state consistently.

### Object-Oriented Principles Demonstrated

* **Inheritance**: Game, Team, and Player all extend the common Entity class.
* **Encapsulation**: Each class controls its own internal lists and data.
* **Polymorphism**: Each subclass can implement methods (like toString()) differently.
* **Design Patterns**:  
  + **Singleton**: Restricts GameService to a single instance.
  + **Iterator**: Used implicitly via loops or streams to find existing names and prevent duplicates.

**"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](#_heading=h.uk8a425j78dt)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | **Deployment**: macOS servers are technically possible but less common in commercial data centers. It can run standard web servers (Apache, Nginx, Tomcat, Jetty).  - **Licensing**: Comes with each Apple machine; no additional OS licensing if you already own the hardware, but Apple servers can be more expensive.  - **Scalability**: Generally stable, but large-scale deployments are more often done in Linux-based data centers. | **Deployment**: Linux (Ubuntu, CentOS, Red Hat, etc.) is the most popular choice for hosting web-based applications. It integrates seamlessly with Java-based services, Docker/Kubernetes, and other open-source technologies.  - **Licensing**: Typically free (or low-cost Red Hat subscriptions).  - **Scalability**: Broad community support, highly scalable, widely used in production servers, and cost-effective for large-scale environments. | **Deployment**: Windows Server is a mature platform for hosting web services (.NET, IIS, or even Java-based Tomcat/JBoss).  - **Licensing**: Requires purchasing Windows Server licenses, which can be more expensive at scale. Azure hosting also factors in OS licensing if using Microsoft’s cloud.  - **Scalability**: Well-supported enterprise solutions, though not as cost-efficient as Linux for very large deployments. | **Deployment**: **Not typically** used as a “server OS.” Mobile devices (iOS, Android) function primarily as **clients** for end users. You generally do not host the main server on a phone/tablet. However, you can run small or local servers for testing on mobile.  - **Licensing**: Tied to the device (iOS or Android), no direct OS server license, but app store membership fees apply for distributing apps (e.g., iOS Dev program fees).  - **Scalability**: Not used in large-scale hosting. Primarily a client. |
| **Client Side** | - macOS desktops/laptops can access the game via **modern, responsive web browsers** (Safari, Chrome, Firefox).  -**Cost/Time/Expertise**: Minimal overhead if you create a **single** HTML5/JavaScript front end that is cross-browser. Testing on Safari specifically is important for Mac. | - Linux desktops are less common among general end users, but power users/devs may access via Firefox or Chrome. The **HTML5** approach remains the same. Minimal extra cost beyond ensuring cross-browser compatibility.  - **Expertise**: Typically, your dev team is comfortable with general Java/JavaScript if you are already building for the server side. | - The majority of desktop users run Windows, so ensuring compatibility with **Chrome, Firefox, Edge** is critical.  **-**  **Cost/Time/Expertise**: Typically no extra OS license cost for the end user, but you must test thoroughly across different Windows browsers to ensure a smooth experience. | - The Gaming Room already supports **Android**. Extending to **iOS** or continuing Android support requires HTML5 responsive design or a separate native/hybrid app.  -**Cost/Time/Expertise**: If a **single** responsive front end is used, you minimize extra overhead. However, if you create **native** apps, you need iOS-specific dev (Xcode/Swift) and Android dev (Android Studio/Kotlin) teams, which adds time/cost |
| **Development Tools** | **Languages/Frameworks**: Java, JavaScript, Swift for native Mac apps, or cross-platform frameworks (React, Angular). Eclipse/IntelliJ can run on Mac with no issues.  - **Licensing**: macOS is commercial but included with Apple hardware. Dev tools like Xcode are free, as are Eclipse/IntelliJ Community editions. | **Languages**  **/Frameworks**: Typically Java, JavaScript, C++, Python, etc. Most are open-source. Tools include Eclipse, IntelliJ, VS Code. Linux-based Docker and CI/CD pipelines are standard.  - **Licensing**: Open-source operating system, free dev tools (Eclipse, VS Code, etc.). Minimal overhead. | **Languages**  **/Frameworks**: Java and JavaScript remain cross-platform. Microsoft Visual Studio is often used for .NET but can also support C++ and cross-platform dev. Eclipse/IntelliJ also work on Windows.  - **Licensing**: Windows OS/Server plus possible IDE licensing (although many free editions exist). | **Languages**  **/Frameworks**: Android Studio (Java/Kotlin), Xcode (Swift/Objective-C), or cross-platform frameworks (Flutter, React Native).  - **Licensing**: Typically free dev tools, but publishing on iOS requires an annual fee for Apple Developer Program ($99/year). Google Play has a one-time fee ($25).  - **Impact**: Potentially separate dev teams or skill sets if building fully native apps, but a responsive web-based approach can unify the client code. |

## Recommendations

Finally, we present a concise set of recommendations to guide The Gaming Room in deploying **Draw It or Lose It** in a distributed web environment:

1. **Operating Platform** We recommend hosting the application on **Linux** servers (e.g., Ubuntu or CentOS) due to its **robustness**, **cost-effectiveness**, and **large open-source ecosystem**. Linux is the most common production environment for Java-based web applications. For local development, your team can use Windows, Mac, or Linux interchangeably since Java is cross-platform.
2. **Operating Systems Architectures** Linux typically runs on x86-64 architecture for servers, which is widely supported. Its modularity allows easy deployment in containers (Docker/Kubernetes) for **scalable microservices**. Meanwhile, your developers can continue to use macOS or Windows for coding and unit testing.
3. **Storage Management** For storing game data (teams, players, etc.), we recommend a **relational database** (such as MySQL or PostgreSQL) or a NoSQL solution (like MongoDB) depending on expected scale and data structure. This ensures persistent IDs and unique name checks beyond in-memory lists if the environment scales significantly.
4. **Memory Management** Java automatically handles **garbage collection**. On Linux servers, you can configure JVM heap sizes to optimize performance. A container orchestration platform (like Kubernetes) can manage memory constraints per container to efficiently scale multiple game instances if needed.
5. **Distributed Systems and Networks** For multi-platform communication, **RESTful web services** or **WebSockets** can be implemented to let different devices interact with the central server. Linux servers can be clustered or load-balanced to handle large traffic volumes. Keep in mind potential **network outages**—the application should handle partial connectivity gracefully.
6. **Security**
   * Enforce **HTTPS** for all client–server communication to protect user data in transit.
   * Use proven identity management (e.g., OAuth) if you add user authentication.
   * Host on a secure platform (e.g., Linux with firewalls, routine patching, intrusion detection).
   * Validate input from all client devices to prevent malicious activity, and store only necessary user data to minimize risk.

By following these recommendations, The Gaming Room can confidently expand **Draw It or Lose It** into a **distributed, web-accessible** experience. The chosen **Linux** platform for the production server environment maximizes stability, cost-efficiency, and scalability. Java’s cross-platform nature ensures that the same code can run on Windows or Mac for development and easily be deployed to Linux for production. Security measures—like HTTPS and strict user input validations—provide the protection needed for a growing, internet-based user community.