

Draw It or Lose It

# **CS 230 Project Software Design**

Version 1.2

## Table of Contents

[**CS 230 Project Software Design 1**](#_heading=h.tiw493ia4vx9)

[**Table of Contents 2**](#_heading=h.uesnbe1cg297)

[**Document Revision History 2**](#_heading=h.bx8kao8f1lm7)

[**Executive Summary 3**](#_heading=h.ecfmo3slhelc)

[**Requirements 3**](#_heading=h.ekcr4ncqz3uf)

[**Design Constraints 3**](#_heading=h.50gj1uze9kvh)

[**System Architecture View 4**](#_heading=h.w5k2gflnvznf)

[**Domain Model 4**](#_heading=h.7dcm981kk2n5)

[**Evaluation 6**](#_heading=h.4q9popbtdrvx)

[**Recommendations 8**](#_heading=h.3dy4xhbmxf8)

## [Document Revision History](#_heading=h.l7jrpjaff5jb)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 03/23/25 | Keir Charlton-Molloy | This revision is the initial submission. Completed all required sections and updated design to meet project specifications. |
| 1.1 | 04/04/25 | Keir Charlton-Molloy | Revisions made to the evaluation section |
| 1.2 | 04/20/25 | Keir Charlton-Molloy | Revisions made to the recommendations section |

## 

## [Executive Summary](#_heading=h.3upeo38nht5c)

The Gaming Room aims to expand its successful Android-based game, Draw It or Lose It, into a web-based, distributed environment. The goal is to create an application that operates seamlessly across multiple platforms while preserving the core functionality of the existing game. To achieve this, a client-server model will be implemented, enabling real-time interaction between multiple teams and players.

The proposed solution incorporates industry-standard design patterns to ensure efficiency and scalability. A singleton pattern will be used to maintain a single global game instance in memory, preventing duplication. Additionally, the iterator pattern will enforce unique identifiers for game, team, and player names. The system will be designed for scalability and concurrency, allowing multiple users to interact simultaneously in a distributed environment. Furthermore, an extensible structure will be established to support future enhancements, such as real-time messaging and scoring.

Key considerations moving forward include ensuring enough infrastructure for growth and addressing challenges like server management and network delays. The system will be built to support future hardware upgrades, ensuring long-term compatibility and easy updates.

By meeting these objectives, The Gaming Room will create a software design that will not only accommodate growing user demand but also facilitate seamless gameplay across various platforms while supporting future hardware advancements.

## Requirements

**Business Requirements:**  
The business requirements for the new web-based version of Draw It or Lose It focus on maintaining a single point of truth, ensuring that only one instance of the game runs at any given time. To prevent conflicts, there must be unique names for game, team, and player names. The game must also support team-based gameplay, allowing multiple teams with multiple players in each match. Additionally, the system should be designed with scalability in mind, enabling it to handle growth without performance issues as more players and teams join.

**Technical Requirements:**  
On the technical side, the game must function in a distributed web-based environment that’s accessible across various platforms, including web and mobile. A singleton pattern will be used to enforce a single instance of the GameService class to prevent multiple game instances from running in memory. To ensure name uniqueness, the iterator pattern will be implemented to check for duplicates before adding new entities. The system will also follow an object-oriented approach, utilizing a base Entity class to promote reusability and minimize code duplication.

## [Design Constraints](#_heading=h.899eap46wv34)

For the single-instance constraint, the singleton pattern in GameService prevents duplicate game instances. This ensures that any logic controlling the game state remains consistent across all clients.  
  
For the name uniqueness constraint, the iterator pattern checks existing names for games, teams, and players before creating new ones to avoid naming collisions.  
  
For the distributed environment constraint, because the game is web-based, the system must account for scalability, maintainability, and potential synchronization challenges. Future solutions may involve session management, load balancing, or distributed storage.

For performance and concurrency, multiple requests could add teams or players simultaneously. Proper concurrency controls, like thread-safe patterns and synchronized methods, may be needed to keep data consistent while maintaining performance.  
  
For extensibility, the design should stay flexible for possible features, such as chat, scoring, authentication, along with potential microservices or third-party integrations.

Overall, by adhering to these constraints, the design will produce a scalable, efficient, and maintainable solution for The Gaming Room’s web-based version of Draw It or Lose It. This helps to ensure a seamless experience for all users as the game grows.

## [System Architecture View](#_heading=h.c9kz3x1l55lt)

*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.7mcfvyx1l4h)

The UML diagram includes ProgramDriver, SingletonTester, Entity, GameService, Game, Team, and Player. Each class plays a specific role and demonstrates core object-oriented principles.

The ProgramDriver class contains main() and serves as the application's entry point. It uses SingletonTester to validate the singleton implementation.

The SingletonTester class defines testSingleton() to confirm that only one instance of GameService exists. It is called by ProgramDriver to verify singleton behavior.

The Entity class serves as a base class with id and name attributes. It includes a long id, a String name, along with constructors and getters/setters. Based on the inheritance OOP principle, Game, Team, and Player extend Entity to avoid duplicating common fields.

The GameService class manages all Game objects, storing them in a list. Its key attributes include games: List<Game>, nextGameId, nextTeamId, nextPlayerId, and a static GameService instance. It provides methods such as getInstance(), addGame(), getGame(), and ID management functions. It has a 0..\* association with Game and follows the singleton OOP principle, ensuring only one global GameService instance manages all games.

The Game class represents a specific instance of Draw It or Lose It. It has a teams: List<Team> attribute and provides methods like addTeam() and toString(). The class inherits from Entity and has 0..\* teams. Following the inheritance and encapsulation OOP principles, it manages its own teams.

The Team class represents a group of players within a game. It has a players: List<Player> attribute and provides methods like addPlayer() and toString(). The class inherits from Entity and has 0..\* players. Following the inheritance and encapsulation OOP principles, each team manages its own players.

The Player class represents an individual participant in a team. It provides the toString() method and inherits from Entity, allowing one team to have multiple players. Following the inheritance OOP principle, it shares the id and name attributes with its parent class.  
  
To further expand on the design, it follows key object-oriented principles to ensure efficiency and maintainability. Inheritance allows Game, Team, and Player to extend Entity, reusing id and name to avoid redundancy. Encapsulation ensures that classes manage their own data internally, such as teams in Game and players in Team. Abstraction is applied through the Entity class, which provides a common identification structure, allowing child classes to focus on their specialized behavior. The singleton pattern is implemented in GameService to enforce a single global instance, ensuring consistent management of game objects. Lastly, composition and aggregation define the application's structure. For example, Game contains Team objects and Team contains Player objects, creating a well-organized and scalable system.

Overall, this design meets the requirements for a single instance of the service, unique entity names, and support for multiple teams and players. The UML structure enables easy scaling and distribution, ensuring Draw It or Lose It can seamlessly expand to web-based and multi-platform environments.

**"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.tls3b8rbovmf)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | macOS supports web servers like Apache and Docker containers, offering strong security and a Unix foundation. However, it's rarely used in production environments due to limited server-grade hardware and higher cost. Not ideal for scalable deployments. | Linux dominates the server market for its flexibility, security, and performance. It supports Apache, Nginx, Node.js, and works seamlessly in cloud environments. It's open-source and free, though it may require advanced expertise for setup and maintenance. | Windows servers integrate well with Microsoft’s ecosystem and offer a user-friendly interface. However, licensing costs, overhead for certain server tasks, and frequent updates can be drawbacks. Still, modern Windows Server versions offer improved security and performance.   |  | | --- | | Mobile devices (iOS, Android) can host limited local servers but aren’t practical for enterprise use. They are useful for testing or demos but face resource, scalability, and platform restrictions, making them unsuitable for large-scale applications.   |  | | --- | |
| **Client Side** | Mac users typically access web apps using Safari, Chrome, or Firefox. Native macOS apps may require Swift or Objective-C expertise. Costs can be higher due to hardware and development tool requirements, though Apple’s environment offers a polished UX and good testing options.   |  | | --- | | Linux users mainly rely on browsers like Firefox or Chrome. Package managers (.deb, .rpm) help distribute native Linux apps, but differences between distributions can make support challenging. Development costs are low, but knowledge of multiple distros may be needed.   |  | | --- | | Windows is widely used, so ensuring compatibility with browsers like Edge, Chrome, and Firefox is important. Desktop apps often need .NET, C#, or C++. While licensing for some tools can add costs, Windows offers extensive documentation and a large developer community, making development easier.   |  | | --- | | Mobile devices are key clients, needing iOS (Swift/Objective-C) or Android (Java/Kotlin) apps. Development costs include platform licenses and specialized UX design. Supporting multiple OS versions and devices adds complexity, but mobile reach is crucial for today’s users.   |  | | --- | |
| **Development Tools** | On Mac, common programming languages include Java, Python, JavaScript, and Swift. Xcode is used for native Mac/iOS apps, while Eclipse, IntelliJ, and VS Code support cross-platform development. Mac's Unix foundation makes tool installation easier, such as using Homebrew, for a smooth development experience.   |  | | --- | | Linux supports most major languages (Java, Python, Ruby, Go, JS) and frameworks. IDEs like Eclipse, IntelliJ, and VS Code work well on Linux. It's great for server-side frameworks (Apache, Nginx, Tomcat), but beginners may find setup more challenging. | Windows works well with Visual Studio, especially for .NET, and supports cross-platform IDEs like Eclipse, IntelliJ, and VS Code. Common languages (Java, Python, JS) run smoothly, and WSL enhances cross-platform support. However, licensing fees for some Microsoft tools can be a consideration.   |  | | --- | | Mobile development relies on Android Studio (Java/Kotlin) and Xcode (Swift/Objective-C). Cross-platform tools like React Native, Flutter, and Xamarin make it easier to build for both platforms, but they can complicate testing. These tools are designed for client apps, not full server environments. |

## Recommendations

1. Operating Platform: The recommended operating platform is Linux, specifically distributions such as Ubuntu or Debian. Linux provides an optimal environment for expansion due to its flexibility, support for various file systems, and efficient management of resources. It offers robust security, community-driven support, and cost-effectiveness due to its open-source nature.
2. Operating System Architectures: Linux architecture is recommended, which prominently features the Virtual File System (VFS) that facilitates interaction between different file system types. VFS abstracts the specifics of local and remote file systems, enabling smooth integration and management of files across diverse platforms. This design supports efficient, modular file operations critical for high-performance gaming environments.
3. Storage Management: ZFS is recommended for storage management. ZFS excels in efficient disk usage, dynamically allocating storage without pre-set constraints. It handles large-scale file systems effectively with minimal metadata overhead. ZFS also includes built-in data integrity features, reducing the likelihood of data corruption through checksums and self-healing mechanisms.
4. Memory Management: Linux uses a method called paging to manage memory. It breaks up memory into small chunks called pages, which are then matched to spaces in physical memory. This helps use memory more efficiently and avoids wasting space. Linux also uses shared libraries, which means different programs can use the same pieces of code instead of each having their own copy. This saves memory and reduces duplication.
5. Distributed Systems and Networks: For cross-platform communication, Distributed File Systems (DFS), such as Network File System (NFS), are recommended. NFS allows for distributed file access across various client and server architectures, improving scalability and reliability. Its stateless protocol minimizes disruptions from network or server outages, enhancing system resiliency and facilitating smoother operations and recovery.
6. Security: Linux employs a sophisticated, permission-based security model with user and group controls to ensure secure access to resources. File and directory permissions can be precisely managed to limit access only to authorized users. Additionally, centralized authentication services like LDAP provide secure, efficient management of user identities across platforms, supporting encrypted credentials and systematic access controls that safeguard user data.

In summary, using Linux for managing resources, along with ZFS for storage and DFS for network sharing, helps Draw It or Lose It run smoothly, grow easily, and stay secure across different systems.