

Draw it or Lose it

# **CS 230 Project Software Design Document**

Version 1.1

## Table of Contents

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

[**Table of Contents 2**](#_30j0zll)

[**Document Revision History 2**](#_3znysh7)

[**Executive Summary 3**](#_2et92p0)

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

[**Design Constraints 3**](#_1t3h5sf)

[**System Architecture View 3**](#_4d34og8)

[**Domain Model 4**](#_2s8eyo1)

[**Evaluation 5**](#_17dp8vu)

[**Recommendations 6**](#_26in1rg)

## [Document Revision History](#_lnxbz9)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 11/17/2024 | Breeanna Hairston | Initial documentation |
| 1.1 | 11/17/2024 | Breeanna Hairston | Revised and added to the evaluation. Added and summarized new/old recommendations. |

## [Executive Summary](#_35nkun2)

The Gaming Room aims to expand its successful Android-based game *Draw It or Lose It* into a web-based, multi-platform application. The game will retain its core mechanics while being accessible across devices, providing a seamless, competitive gaming experience. This design addresses the client's key requirements: enabling multiple teams and players, ensuring unique names for teams and games, and enforcing a single-instance memory model for game management. Utilizing robust software design principles such as inheritance, singleton patterns, and iterative processing, we will create a scalable, efficient solution tailored to the client’s needs.

This document outlines the software architecture, domain model, and constraints for development, ensuring clarity and alignment with The Gaming Room’s goals.

## Requirements

1. *The game must support one or more teams, with each team comprising multiple players.*
2. *Game and team names must be unique to prevent conflicts during gameplay.*
3. *Only one instance of the game can exist in memory at any time.*
4. *Unique identifiers for all entities (game, team, player) must be generated automatically.*

## [Design Constraints](#_1ksv4uv)

1. **Distributed Web-Based Environment**:
   * The game must function across a distributed web architecture. This introduces challenges such as network latency, session management, and state synchronization across platforms.
2. **Concurrency**:
   * Multiple users interacting simultaneously necessitates robust data consistency mechanisms and thread-safe operations.
3. **Platform Independence**:
   * The application must be accessible across various operating systems (Windows, Mac, Linux) and devices (desktop and mobile). Compatibility testing and responsive design will be crucial.
4. **Scalability and Maintainability**:
   * As user traffic grows, the system must handle increased loads without performance degradation. Clean, modular code following object-oriented principles will aid scalability.

Implications: These constraints dictate the need for stateless server architectures, robust backend frameworks (e.g., Java-based Spring Boot), and careful testing to ensure consistent user experiences.

## 

## [System Architecture View](#_44sinio)

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](#_2jxsxqh)

The UML class diagram provides an overview of the key entities and their relationships within the Draw It or Lose It game application system. At the core is the Entity class, which serves as a base superclass for all entities in the application. It defines common attributes such as id and name, ensuring that all subclasses share these properties. This structure enforces consistency across all system entities, including games, teams, and players.

The Game, Team, and Player classes extend the Entity class, establishing an inheritance relationship. This hierarchical design simplifies code maintenance by allowing these subclasses to inherit and reuse the shared attributes and methods defined in Entity. The Game class contains multiple Team instances, and each Team holds multiple Player instances, forming a composition relationship between these classes. This design reflects the real-world structure of the game, where games consist of teams, and teams consist of players.

The GameService class plays a central role in managing the lifecycle of the game application. It has a composition relationship with the Game class, maintaining references to multiple game instances. As a singleton, the GameService class ensures that only one instance exists in memory, using a private constructor and a static access method to enforce this pattern. This guarantees centralized control over game-related operations, such as adding games, teams, and players, and prevents duplicate instances from being created.

The ProgramDriver class serves as the entry point of the application. It interacts with the singleton instance of GameService to initialize and manage game data. Through this class, games, teams, and players are created and manipulated. Additionally, the ProgramDriver class has a dependency on the SingletonTester class, as indicated by the <<uses>> arrow in the diagram, to validate the singleton behavior of GameService.

The UML diagram demonstrates key object-oriented programming (OOP) principles, including:

* Inheritance: The Entity class defines common attributes that are inherited by Game, Team, and Player, reducing redundancy and ensuring uniformity across entities. Subclasses use the super keyword to initialize properties defined in the superclass.
* Encapsulation: The GameService class restricts direct access to its attributes, such as the list of games, by providing controlled methods for interaction. This design ensures data integrity and privacy.
* Abstraction: The classes abstract unnecessary details, focusing only on relevant behaviors and interactions. For example, GameService manages high-level operations like game creation, shielding the underlying implementation details from other parts of the application.

By leveraging these OOP principles, the design ensures modularity, reusability, and ease of maintenance, meeting the software requirements efficiently.

**"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](#_z337ya)

| **Aspect** | **Linux** | **Mac** | **Windows** | **Mobile Platforms (iOS & Android)** |
| --- | --- | --- | --- | --- |
| **Server Side** | Offers a cost-effective, robust server-based deployment with open-source advantages. Ideal for hosting due to stability and configurability. No licensing costs; highly customizable. | Can be used for hosting, but higher costs for hardware and limited scalability compared to Linux. Higher hardware costs; macOS licenses are bundled with Apple devices. | Windows Server offers robust hosting capabilities but involves significant licensing costs. Licensing costs for Windows Server can be significant. | Limited as a server; mainly acts as a client-side interface for web applications. Hosting options are minimal for mobile devices. |
| **Client Side** | Open-source frameworks (e.g., GTK, Qt) ensure compatibility with web browsers. | Requires specific development tools (e.g., Xcode for macOS and iOS) for seamless compatibility. | Broad support for web browsers but often needs testing for Edge compatibility. | Development tools like Android Studio and Xcode ensure compatibility. |
| **Development Costs** | Minimal; open-source tools reduce overall expenses. | Moderate; requires Apple hardware and expertise. | High; licensing costs and multiple proprietary tools increase expenses. | Moderate; cross-platform tools (e.g., Flutter) help reduce redundancies. |
| **Development Time** | Slightly longer due to the variety of open-source tools and frameworks available. | Moderate; streamlined for Apple ecosystems. | Moderate; support for popular IDEs ensures efficient development. | Moderate to long; responsive design ensures cross-device compatibility. |
| **Expertise Required** | Developers need familiarity with Linux environments and open-source tools. | Expertise in Xcode and Apple-specific workflows required. | Familiarity with Windows and its proprietary development ecosystem is needed. | Knowledge of mobile development and responsive design techniques is necessary. |
| **Development Tools** | Eclipse, IntelliJ IDEA, Visual Studio Code, open-source libraries (e.g., Flask, Django). | Xcode for macOS and iOS; IntelliJ IDEA; Unity for cross-platform development. | Visual Studio, IntelliJ IDEA, Unity; strong tool ecosystem but licensing required. | Android Studio for Android; Xcode for iOS; Unity or Flutter for cross-platform. |
| **Licensing Costs** | Minimal; most tools and OS distributions are free. | Requires Apple hardware and OS, which adds indirect costs. | Licensing costs for OS and proprietary tools can be substantial. | Tool costs depend on the choice of cross-platform solutions. Flutter and Unity are viable. |
| **Scalability** | High; Linux’s modular nature and containerization tools like Docker ensure scalability. | Moderate; macOS servers are less common and harder to scale compared to Linux. | High; robust scaling options but costs can grow significantly with demand. | Limited scaling as hosting is not a primary function. |
| **Cross-Platform Support** | Supported via modern browsers and responsive design. | Supported via modern browsers, but testing required for Safari. | Supported via modern browsers, with testing needed for Edge and IE compatibility. | Achievable through tools like Flutter, Unity, or React Native. |

## Recommendations

#### Operating Platform

* **Server-Side Deployment**: Recommend Linux for cost-effectiveness, scalability, and stability. Windows Server is a viable but costlier alternative for familiarity.
* **Client-Side Compatibility**: Focus on web-based deployment with cross-platform frameworks like Unity or Flutter for broad compatibility.

#### Operating Systems Architectures

* **Cloud Architecture**: Utilize AWS or Azure for a microservices architecture, dividing the application into manageable services like player authentication and game logic.
* **Client-Server Model**: Employ lightweight client-side applications for rendering, with heavy computational tasks handled on the server.
* **Cross-Platform Development**: Use Unity for a single codebase deployment across operating systems with architecture-specific optimizations.

#### Storage Management

* **System**: Use cloud-based storage with services like Amazon RDS or Azure SQL Database for structured data and AWS S3/Azure Blob Storage for unstructured data.
* **Features**: Leverage automated backups, scalability, and regional replication for data availability and disaster recovery.

#### Memory Management

* **Techniques**:
  + **Caching**: Implement AWS Elasticache or Azure Cache for Redis for quick data retrieval.
  + **Garbage Collection**: Unity's automatic garbage collection prevents memory leaks.
  + **Virtual Memory**: Cloud servers extend physical RAM with virtual memory for handling large datasets.

#### Distributed Systems and Networks

* **Distributed Software**: Use RESTful or GraphQL APIs for seamless communication across platforms.
* **Networking**: Employ AWS VPC or Azure Virtual Network for secure connectivity.
* **Dependencies**: Implement load balancers for traffic distribution and redundancy mechanisms to mitigate outages.

#### Security

* **Encryption**: Secure data with end-to-end encryption (SSL/TLS) and encrypt sensitive data at rest.
* **Authentication**: Use OAuth2.0 and two-factor authentication (2FA).
* **Firewall**: Protect against threats with AWS WAF or Azure Firewall.
* **Secure APIs**: Use managed gateways for API authentication and attack mitigation.
* **Audits**: Conduct regular vulnerability assessments and penetration testing.