

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)**-5**

[**Evaluation 6**](#_Toc115077325)

[**Recommendations 7**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 05/25/2025 | Phill Nunez | Software Design Document Initial Set-up |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room is working to develop a web-based version of their multiplayer guessing game, *Draw It or Lose It*, inspired by the 1980s show *Win, Lose or Draw*. The goal is to design an efficient and scalable software solution that enables multiple teams and players to compete online in a timed, drawing based guessing game.

To meet business and technical goals, I propose building a distributed, multi-tiered web application using a central game engine that manages game logic, timing, and team/player data. A singleton pattern will be utilized to make sure there is a single active instance of a game. This solution will support cross-platform accessibility while maintaining uniqueness of team and player names while also leveraging scalable storage and network architecture.

## Requirements

* The game needs to support multiple teams, with each team comprising of multiple players
* Game, Team, and Player names must be unique to prevent duplicate entries
* Only one instance of the game is allowed in memory (singleton enforcement)
* The system must be web-based
* The game operated in timed rounds with clues revealed progressively
* Teams must submit guesses within time constraints
* Platform scalability and cross-platform support are critical
* Security and user data protection is essential

## [Design Constraints](#_2et92p0)

* Singleton Requirement: Only one active game instance can be in memory
* Web-Based Access: the game must be accessible via a browser and mobile web interface
* Distributed Architecture: The system must manage state consistency across multiple clients
* Uniqueness Validation: Game, Team, and Player names must be checked against existing entries
* Time Synchronization: Game logic must enforce strict timing constraints for rounds and guessing windows

## [System Architecture View](#_ilbxbyevv6b6)

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](#_8h2ehzxfam4o)

Program Driver

* Contains a single method: +main()
* This class serves as the application entry point and coordinates the creation of games, teams, and players

Singleton Tester

* Contains: +testSingleton() method
* Used to verify the singleton behavior of the GameService class

Relationship

* A <<uses>> dependency is shown from ProgramDriver to SingletonTester which indicates that the main method makes use of the testSingleton() function

Entity Class (Superclass)

* Attributes:
  + -id: long
  + -name: String
* Constructors and Methods:
  + -Entity()
  + +Entity(id: long, name: String)
  + +getId(): long
  + +getName(): String
  + +toString(): String
* Purpose:
  + Acts as a base class for Game, Team, and Player
* Inheritance:
  + Game, Team, and Player each inherit from this class

GameService Class(Singleton)

* Attributes:
  + -games: List<Game>
  + -nextGameId: long
  + -nextPlayerId: long
  + -nextTeamId: long
  + -service: GameService (Singleton Instance)
* Constructor:
  + -GameService()
* Methods:
  + +getInstance(): GameService – returns the singleton instance
  + +addGame(name: String): Game
  + +getGame(id: long): Game
  + +getGame(name: String): Game
  + +getGameCount(): int
  + +getNextPlayerId(): long
  + +getNextTeamId(): long
* Relationships:
  + 0..\* association with Game indicates that it can manage none or multiple game instances

Game Class

* Attributes:
  + -teams: List<Team>
* Constructor:
  + +Game(id: long, name: String)
* Methods:
  + +addTeam(name: String): Team
  + toString(): String
* Relationships:
  + 0..\* association with Team indicates that a game can contain none or multiple teams
  + Inherits from the Entity Class

Team Class

* Attributes:
  + -players: List<Player>
* Constructor:
  + +Team(id: long, name: String)
* Methods:
  + +addPlayer(name: String): Player
  + toString(): String
* Relationships:
  + 0..\* association with Player indicates that a team can have none or multiple players
  + Inherits from the Entity Class

Player Class

* Constructor:
  + +Player(id: long, name: String)
* Methods:
  + toString(): String
* Inheritance:
  + Inherits from the Entity Class

**"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** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | macOS can be used for server development but it is not common in production. Limited server deployment tools, but they are great for local testing. Ther is a higher licensing cost if scaled for production | Most widely used for web servers. Lightweight, secure, supports all major backend frameworks. No licensing fee is a big upside. | Windows servers support enterprise tools but have higher resource overhead support, not as flexible as Linux for modern web apps. Requires Windows Server licensing and Client Access Licensing which increase cost. | Not suitable for full server hosting. Typically acts as clients. Back end must be hosted on external servers. |
| **Client Side** | Requires cross-browser testing. Fewer compatibility issues with modern browsers. Moderate development cost | Most desktop users do not utilize Linux, so testing on Linux desktops is less common. Development cost is low but there is limited audience | Most common platform. Highest compatibility testing requirements. Widely supported | Requires responsive design and device specific testing(Android/iOS). High Fragmentation means more effort, but critical or more user engagement |
| **Development Tools** | Xcode, IntelliJ, Eclipse, and Docker  Java, JavaScript, Spring Boot, Node.js. Mac hardware is required which increase cost. | VS Code, Eclipse, Docker, Node.js. Again no licensing cost. | Visual Studio, IntelliJ, Eclipse, Node.js. Widespread tool availability. Some developer tools may incur licensing fees. | Android Studio Code, Xcode, React Native, Flutter. Java/Kotlin (Android), Swift (iOS), JavaScript (Cross-Platform)  iOS requires a $99 per year developer program membership, Google Play Developer is a one-time $25 fee. |

## 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**: Linux – it offers the most cost-effective, secure, and scalable server environment for deploying distributed web applications. Ideal for hosting APIs and real-time services. It supports multi-core systems, process isolation, and high concurrency. Linux provides the capability to be compatible with cloud providers and support for containerization, making it ideal for further expansions.
2. **Operating Systems Architectures**: Linux provides a modular and stable kernel with support for multi-user environments, process isolation, and system security. This makes it ideal for a multi-threaded, event-driven game engine
3. **Storage Management**: Use a relational database system such as MySQL. This database supports transactional consistency, indexing for fast lookups, and is compatible with distributed deployments.
4. **Memory Management**: Linux utilizes virtual memory with paging and caching and supports Java’s garbage collection. The Singleton design ensures that only one Game instance is in memory, thereby minimizing memory usage. Node.js memory settings can be utilized as needed.
5. **Distributed Systems and Networks**: Utilize WebSockets for real-time gameplay to facilitate communication between the server and clients. Load balancers and reverse proxies can distribute traffic. Use stateless architecture to allow horizontal scaling. Handle outages using retry logic and fallback messaging queues.
6. **Security**: Use HTTPS for all communication, implementing OAuth2 for authentication. Validate inputs on the server-side to avoid injection. Encrypt sensitive data at rest using Advanced Encryption Standard and in transit with Transport Layer Security. Employe access control lists to prevent unauthorized access.