

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 4**](#_Toc115077322)

[**System Architecture View 4**](#_Toc115077323)

[**Domain Model 4**](#_Toc115077324)

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

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

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | <mm/dd/yy> | Justin Hartwick | Created the initial software design document for *Draw It or Lose It*, outlining the game's core functionality, unique entity relationships, and implementation of software design patterns. Added sections covering the executive summary, design constraints, and architecture considerations tailored for a web-based environment. |

**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, a client of Creative Technology Solutions (CTS), is planning to expand their game *Draw It or Lose It* beyond its current Android application. The goal is to transition the game into a web-based platform that supports multiple operating systems while maintaining efficiency and scalability. This software must facilitate team-based gameplay, ensure unique game and team names, and provide a seamless experience across devices.

To achieve this, the system is designed with object-oriented principles and utilizes software design patterns for optimized performance. A singleton pattern is implemented to manage game instances, ensuring only one instance of GameService exists at a time. The iterator pattern is used to efficiently handle game, team, and player entities, maintaining uniqueness while optimizing performance.

This document details the technical architecture, design constraints, and structural considerations necessary to create a scalable, maintainable, and high-performance game application that meets The Gaming Room’s objectives.

## Requirements

Business Requirements

1. The platform must support multiplayer functionality, allowing users to join team-based game sessions.
2. Players must be able to interact across multiple operating systems, including Windows, Mac, and mobile devices.
3. The system must prevent duplicating game and team names, ensuring each session is uniquely identifiable.
4. The application should be designed for long-term scalability, allowing seamless future expansions.
5. User experience is a priority—the interface must be intuitive, ensuring smooth navigation for all players.
6. Administrative tools should allow for effortless game moderation and session management.

Technical Requirements

1. Game and Entity Management:
   * Each game, team, and player should have an automatically assigned unique identifier.
   * The system must implement a singleton pattern to prevent multiple instances of the GameService, ensuring centralized control.
2. Cross-Platform Compatibility:
   * The application must be web-based, allowing access from desktops and mobile devices.
   * Compatibility must be ensured for major browsers, preventing platform-specific issues.
3. Performance Optimization:
   * The system must be built to handle simultaneous players without performance drops.
   * Database queries and entity retrieval should be optimized for efficient data management.
4. Security Measures:
   * Authentication mechanisms should be enforced to prevent unauthorized game access.
   * Data validation and sanitization must be implemented to counteract security vulnerabilities.
   * Secure storage of game session data is necessary to prevent manipulation or data loss.

## [Design Constraints](#_2et92p0)

1. Web-Based Distributed Environment

The application must be designed as a web-based system, enabling accessibility across different platforms. This introduces several key constraints:

* Cross-platform compatibility (Windows, macOS, Linux, and mobile browsers)
* Cloud-based hosting for scalability and remote access
* Efficient database integration to manage game sessions, teams, and players effectively

2. Unique Identifiers for Game, Team, and Player Instances

To prevent duplication, all entities (games, teams, and players) must have unique identifiers. This is enforced through:

* A central Entity class, which assigns **entityId** and **entityName** attributes to all game-related objects.
* **GameService** ensures uniqueness by checking existing names before creating new entities.
* Iterators in **addGame**(), **addTeam**(), and **addPlayer**() efficiently search for existing entities.

3. Performance & Scalability

As the player base expands, the game must maintain optimal performance. To achieve this:

* Data retrieval and storage are optimized through structured object-oriented programming.
* Memory efficiency is maintained by enforcing the singleton pattern for **GameService**.
* The system is designed to handle multiple concurrent users through well-structured request handling.

4. Security & Data Protection

Security is critical in a web-based game. The system will implement:

* Authentication and session controls to prevent unauthorized access.
* Input validation and data sanitization to mitigate injection attacks.
* Secure storage and encrypted communication to protect game data integrity.

## [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)

The UML class diagram illustrates the structure and relationships between entities in *Draw It or Lose It*. The design follows object-oriented programming (OOP) principles, ensuring modularity, scalability, and maintainability.

**Class Relationships:**

1. **Entity Class** (Base Class):
   * Acts as a parent class for Game, Team, and Player.
   * Provides shared attributes (**entityId**, **entityName**) and methods (**getEntityId**(), **getEntityName**(), **toString**()).
2. **GameService (Singleton Class)**:
   * Manages all games, teams, and players.
   * Implements singleton pattern to ensure only one instance exists.
   * Uses the iterator pattern to enforce uniqueness.
3. **Game Class**:
   * Represents an individual game session.
   * Holds a list of teams (teams: List<Team>).
   * Provides methods to add teams and retrieve details.
4. **Team Class**:
   * Represents a team in a game.
   * Holds a list of players (players: List<Player>).
   * Provides methods to add players and retrieve details.
5. **Player Class**:
   * Represents an individual participant.
   * Inherits attributes and methods from Entity.
6. **ProgramDriver (Main Entry Point)**:
   * Initializes **GameService**.
   * Adds test data (games, teams, players) to verify functionality.
7. **SingletonTester Class**:
   * Verifies that only one instance of **GameService** exists.

**OOP Principles Demonstrated**

|  |  |
| --- | --- |
| Principle | Application in UML |
| **Encapsulation** | Uses private attributes with public getters and setters. |
| **Inheritance** | Entity is a parent class, reducing redundant code. |
| **Abstraction** | Defines high-level methods like **addGame**() without exposing internal details. |
| **Polymorphism** | **toString**() is overridden in multiple classes. |

**Efficiency in Meeting Software Requirements**

* **Encapsulation** ensures controlled access to class attributes.
* **Inheritance** minimizes duplicate code, making maintenance easier.
* **The singleton pattern** centralizes game management, preventing conflicts.
* **The iterator pattern** optimizes searches for existing game, team, and player names.

**"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)

The evaluation of different operating platforms considers their suitability for hosting *Draw It or Lose It* as a web-based distributed system. Each platform is analyzed in terms of server-side capabilities, client-side considerations, and development tools.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Mac servers are stable and provide strong security features, but they are costly and less commonly used for web hosting. Limited support for enterprise-level hosting compared to Linux and Windows. | Linux is the most widely used server OS due to its reliability, flexibility, and cost-effectiveness. It provides extensive server management capabilities and is ideal for cloud deployments. | Windows Server offers strong enterprise support and is widely used for hosting, particularly for applications that rely on Microsoft technologies such as .NET and MSSQL. However, it requires licensing fees. | Mobile devices are not ideal for hosting a server but can access cloud-based game servers. The performance is limited by device hardware, and connectivity can be unstable. |
| **Client Side** | Mac users benefit from a smooth UI experience and strong security. However, Mac devices are more expensive, which may limit accessibility. Development for Mac requires specific expertise. | Linux users can access web applications, but desktop adoption is lower than Windows or Mac. Linux is primarily used for server-side hosting rather than client-side gaming. | Windows remains the most widely used OS for gaming, ensuring broad compatibility. Development is well-supported with extensive tools and frameworks. | Mobile devices allow for portable gaming but require responsive UI design and adaptive performance optimizations to handle different screen sizes and processing power. |
| **Development Tools** | Xcode and Swift are required for native Mac development, while Java and JavaScript can be used for web-based applications. | Linux supports a variety of development tools such as Eclipse, VS Code, and command-line utilities. Open-source flexibility allows for extensive customization. | Windows supports a wide range of IDEs, including Visual Studio, Eclipse, and JetBrains products. It is well-suited for both web-based and native applications. | Mobile development requires platform-specific tools such as Android Studio for Android devices and Xcode for iOS development. Cross-platform frameworks like React Native and Flutter can be used for multi-device support. |

## 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

Given the need for scalability, security, and cost efficiency, Linux is the recommended operating platform for hosting *Draw It or Lose It*. Linux offers stability, cloud scalability, and strong security features, making it ideal for web-based distributed environments.

1. Operating Systems Architectures

The Linux operating system supports modular architecture, allowing developers to customize environments according to system needs. It is widely supported on cloud infrastructure such as AWS and Azure, ensuring high availability and redundancy.

1. Storage Management

A cloud-based storage solution (e.g., AWS S3, Google Cloud Storage) is recommended for maintaining player data, game progress, and media files. This allows fast access, automatic backups, and scalability as user demand grows.

1. Memory Management

Linux’s memory management system provides efficient resource allocation through virtual memory, caching, and paging. This ensures that the game can scale dynamically based on user load without degrading performance.

1. Distributed Systems and Networks

To ensure cross-platform compatibility and multiplayer synchronization, the game will use a client-server architecture with WebSockets or REST APIs for real-time communication. Load balancing and failover mechanisms will be implemented to handle concurrent users efficiently.

1. Security

Security measures will include:

* User authentication via OAuth 2.0 or JWT tokens to prevent unauthorized access.
* Data encryption in transit (TLS) and at rest (AES encryption) to protect user data.
* Input validation and sanitization to prevent SQL injection and cross-site scripting (XSS) attacks.

**Next Steps**

Moving forward, further refinements will focus on system integration, security enhancements, and user interface optimizations. The completed application will adhere to industry best practices and provide an engaging and seamless multiplayer experience.