

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)

[**Evaluation 4**](#_Toc115077325)

[**Recommendations 5**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 5/25/2025 | Matthew Molde | Initial version of document |

**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)

Draw It or Lose It is a multiplayer, web-based game inspired by the game show Win, Lose or Draw, where players must guess what is being illustrated. The catch though is going to be rather than having a person draw clues live, the game will displays preloaded images that appear progressively over 60 seconds. Players must guess the item being illustrated before time runs out.

To ensure scalability and maintainability, the software design employs object-oriented programming and design patterns such as Singleton and Iterator. These ensure that only one instance of the game logic is active in memory, and that game components are properly managed and accessible. The code will also enforce unique names for games, teams, and players to avoid conflicts.

This document outlines the technical requirements and constraints necessary for developing the game, including a domain model based on a UML diagram and a breakdown of object-oriented design principles used in the implementation of the code.

## Requirements

 The application must support one or more teams per game.

 Each team must have multiple players.

 All game, team, and player names must be unique.

 Only one instance of the GameService should exist in memory (singleton).

 Drawings are revealed over 30 seconds, and each round lasts 60 seconds.

 If the guessing team fails, other teams can guess within 15 seconds.

 All game logic should be managed in-memory for now, with potential future web deployment.

## [Design Constraints](#_2et92p0)

 Singleton Pattern: Only one GameService instance can exist, centralizing the logic and data handling for each game. This restricts how instances are created and accessed.

 Unique Naming: The application must prevent duplicate names across games, teams, and players. This imposes a constraint on how items are added, making a search and validation mechanism before object creation necessary.

 In-Memory Storage: Since there is no persistent database yet, we must use certain code like List to manage objects, which also affects scalability and memory.

 Iterative Operations: Searching through collections/lists for validation introduces complexity that must be managed with carefully using an iterator pattern.

## [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 domain model, represented in the UML diagram, defines the core entities and their relationships. Entity or the base class contains id and name. This is the superclass for Game, Team, and Player which demonstrates inheritance and reduces redundancy. The GameService which is a Singleton pattern manages the game creation and access. It holds List<Game> and counters for the IDs. It enforces the Singleton pattern and encapsulation by controlling access to the game objects.

The Game class contains the List<Team> attribute which supports adding teams and printing information. The Team class contains the List<Player> attribute and supports adding players, while the Player class represents the individual players.

Finally for verification we have the ProgramDriver class which is the entry point for running the game and the SingletonTester which verifies that only one instance of GameService is being created.

Object-Oriented Design Principles Used include Encapsulation, Inheritance, and Polymorphism. Each class is handling its own data and provides public methods for our Encapsulation principle. For inheritance our common attributes are being abstracted into the Entity superclass. Polymorphism is being practiced by making sure all entities can override toString() or they share behaviors from Entity.

The Composition is that games contain teams and teams contain players with the design patterns being that the Singleton pattern ensures only one GameService instance and the iterator helps validate name uniqueness when adding games teams or players.

**"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 provides a stable environment, but its higher licensing costs and limited industry use for hosting web applications reduce its practicality. | Linux is the industry standard for hosting web apps. Its scalable cheap and widely used and adopted. | While Windows has much more support than Mac the higher costs and more vulnerable state makes it less enticing than Linux. | Mobile platforms are not suitable for hosting due to hardware constraints, operating system limitations, and lack of traditional server support. As well as hardware, and OS limitations. |
| **Client Side** | Developing for Mac clients involves targeting Safari and ensuring compatibility with macOS behaviors. Cost is higher than open source linux and Mac expertise is a must. | Linux clients require support for browsers like Firefox and Chrome. Client-side development on Linux is cost-effective and well-documented, but market share is smaller compared to Windows/Mac. | Most widely used client OS. Broad hardware and browser compatibility. Requires extensive testing due to version variability but has largest market share. | Client development for mobile requires native or hybrid frameworks. It adds complexity but is vital for user accessibility. Cost and development time are higher due to platform diversity. |
| **Development Tools** | Xcode, IntelliJ IDEA, and Eclipse are commonly used on Mac. Java, JavaFX, and Gradle/Maven are all well-supported. | Linux supports most Java IDEs like Eclipse, and IntelliJ. It provides native terminal access, package managers and scripting capabilities. It is ideal for backend Java development. | Visual Studio Code, IntelliJ, Eclipse, and NetBeans are all fully supported. The OS has strong integration with Docker, Java SDKs, and build tools. | Development for mobile requires Android Studio and Xcode or cross-platform frameworks like Flutter. Tools are specialized and often require device emulators. |

**Recommendations**

1. **Operating Platform**: Linux would be my recommended platform for The Gaming Room’s hosting needs. It is reliable, scalable, cost-effective, and the standard for deploying Java-based web applications. It works seamlessly with containers like Docker and cloud infrastructure, providing the flexibility to scale the game as player count grows.
2. **Operating Systems Architectures**: Operating system architectures vary in how they manage processes, memory, and I/O. Windows uses a hybrid kernel model, macOS builds on a UNIX-like microkernel, while Linux uses a monolithic kernel. For hosting Draw It or Lose It, a 64-bit Linux architecture is what I would consider ideal. It allows tight integration with system calls and memory management, has efficient multi-threading for concurrent game sessions, and seamless compatibility with Java EE frameworks.
3. **Storage Management**: Linux supports ext4, a file system that tracks changes to metadata and ensures data integrity during crashes or failures. This reliability is key when storing user progress or team data persistently in MySQL databases. The versatility of Linux also allows integration with network file systems for data sharing between servers. A relational database like MySQL should also be integrated for persistent storage of game sessions, user data, and teams. MySQL also offers robust indexing and transaction support.
4. **Memory Management**: The Java Virtual Machine on Linux uses garbage collection to manage memory for us. Developers can fine-tune this memory usage using JVM flags. This prevents memory leaks and improves performance which will be particularly important as game sessions and players increase. Beyond just JVM, Linux uses virtual memory with paging and demand loading, meaning inactive game data can be temporarily swapped to disk to preserve RAM.
5. **Distributed Systems and Networks**: The application can support cross-platform communication through RESTful APIs over HTTP/HTTPS. Load balancers and health checks can be added to ensure high availability and fault tolerance in case of connectivity issues or partial outages.
6. **Security**: Security should include HTTPS for all network communication. User data must be encrypted at rest and in transit. The Linux platform supports firewall tools like iptables and integration with secure authentication systems such as LDAP. Updates and vulnerability scans should be regularly performed as with all applications and user authorization can be enforced through role-based access control.