

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.0

## Table of Contents

[**CS 230 Project Software Design Template** 1](#_Toc94723243)

[Table of Contents 2](#_Toc94723244)

[Document Revision History 2](#_Toc94723245)

[Executive Summary 3](#_Toc94723246)

[Design Constraints 3](#_Toc94723247)

[System Architecture View 3](#_Toc94723248)

[Domain Model 3](#_Toc94723249)

[Evaluation 4](#_Toc94723250)

[Recommendations 7](#_Toc94723251)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | May 23, 2025 | Anthony Moran | First Draft |
| 1.1 | June 7, 2025 | Anthony Moran | Second Draft-Updated development requirements for all Operating  Systems  Updated development requirements for all Operating  Systems  Second Draft |
| 1.2 | June 20, 2025 | Anthony Moran | FFFSDFASFSD  Final Draft |

## [Executive Summary](#_sbfa50wo7nsh)

CTS has been commissioned by The Gaming Room to develop a web-based version of their existing Android game, *Draw It or Lose It*. The new platform must support multiple teams, each with several players, while ensuring uniqueness for each game session, team, and player.

To manage this complexity, the singleton pattern is applied to limit the instantiation of key game objects, preventing multiple concurrent versions. Additionally, the iterator pattern ensures the uniqueness of teams and players, avoiding duplication or naming conflicts.

## [Design Constraints](#_2et92p0)

Since the game is already available on Android, the technology stack for the web version must align with existing systems. Java has been chosen for backend development due to its compatibility with both Android and web platforms, streamlining the transition. Any current APIs used in the Android version will be reviewed and updated to support web interaction where necessary. Any existing APIs serving the Android platform need to be reviewed or extended for mobile usage.

## [System Architecture View](#_ilbxbyevv6b6)

While no detailed architecture deliverables are required for this phase, it is important to note that future development may require clear documentation of the system's logical and physical components. This includes communication pathways, storage models, and service tiering.

## [Domain Model](#_8h2ehzxfam4o)

The UML for the proposed design is shown below.

**"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.**

The proposed application is structured around a central driver class responsible for initiating games, teams, and players. Object creation is handled by the Game Service class, which follows the singleton design to ensure only one active instance at any time.

By making the constructor private, GameService restricts direct instantiation. Instead, a static method getInstance() controls access. This method checks if an instance already exists before creating a new one.

Games are added through the addGame() method, which uses the iterator pattern to prevent duplicate names. Each new game is stored in a list of games. Similarly, teams are created using addTeam(), with duplicate names filtered out before being added to the game's team list. Players are added to teams using addPlayer() with the same validation mechanism.

The Game, Team, and Player classes all inherit from the Entity superclass, which includes protected id and name attributes. The default constructor is also protected, ensuring all instances are created with meaningful data via overloaded constructors.

The design demonstrates multiple object-oriented principles:

* **Inheritance and Polymorphism** through subclassing and constructor overloading.
* **Encapsulation and Abstraction** by exposing high-level methods like addTeam() to the user, while hiding the internal instantiation logic.

## [Evaluation](#_2o15spng8stw)

There are a few potential targets for development, e.g., Macintosh, Windows, Linux, mobile platforms, for either hosting/serving or acting as the client. The following table discusses the strengths and weakness of each platform.

It should b pointed out that the server and client choices are not linked. That is, if Linux was chosen as the server OS windows may still be the preferred option for the client.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | macOS Server is an option, but costly and not widely hosted. | Most cost-effective and widely supported server OS. Open-source nature reduces license fees. | Familiar GUI environment and office tool compatibility. High licensing costs. | Not ideal for hosting due to hardware limitations. |
| **Client Side** | Development requires macOS hardware and XCode. Lower user market share (~16%). | Offers flexibility in programming languages (Java, Python, C/C++). Limited client market presence. | Broad market penetration (75%). Development is straightforward using C# and .NET. | Ideal for client apps. Android (Java) and iOS (SWIFT) are straightforward to develop for. |
| **Development Tools** | Mac use Objective-C and SWIFT for development languages. | Linux development may take the form of C/C++, Java, or Python. | Windows is primarily developed using C# and primarily .NET.  Microsoft’s Visual Studio is an immensely popular IDE and offers many plugins and integration options, e.g., Jenkins, Test Complete, etc. | Android SDK is Java based, and the most widely used Android IDE is Android Studio which is developed by Google as the official development tool. Android Studio is free to download. |

## Recommendations

**1. Operating Platform**  
Linux is recommended for the backend server due to its lower cost, broad compatibility, and strong security. The frontend can remain flexible, allowing development in platform-specific languages like Java for Android or SWIFT for iOS. The client and server components can communicate using APIs, allowing decoupled development.

**2. System Architecture**  
A modern architecture using microservices and containers (via Docker or Kubernetes) is advised to ensure scalability and resilience. Since *Draw It or Lose It* is not latency-sensitive, asynchronous API communication between frontend and backend will suffice. Client-side rendering will reduce server load and improve user experience by caching visuals.

A browser-based game using a Progressive Web App (PWA) approach would maximize accessibility across platforms without requiring OS-specific installations.

**3. Storage Management**  
Unless proprietary infrastructure is desired, storage can be handled by cloud-native services. With most game logic offloaded to clients, either HDDs or SSDs should suffice, especially with content caching mechanisms in place.

**4. Memory Management**  
Linux employs efficient virtual memory and demand paging via the LRU algorithm, reducing unnecessary memory consumption. On Android, the ART and Dalvik VM handle memory through paging and mmap. iOS now supports ARC, automating memory management through compile-time reference counting.

Server-side RAM requirements will remain low thanks to frontend rendering, and modern containerized systems will allow scaling memory usage with user load.

**5. Distributed Systems & Networking**  
Using a cloud-native approach enhances uptime and allows for seamless load balancing. RESTful APIs will facilitate asynchronous communication between the server and clients across different platforms. This architecture supports resilience and adaptability in multi-region deployments.

**6. Security Considerations**  
Security should include role-based access control (RBAC), limiting users to actions appropriate to their roles (e.g., game creation, team management). Administrative access will be restricted.

All API interactions will be encrypted using TLS 1.2+ with 128-bit SHA-256 keys. Entrust certificates will secure server communications. A robust firewall setup should also be implemented to guard against external threats and ensure compliance with industry best practices.