

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 | 8/16/2025 | Reice Morgan | Initial Draft for client review |

**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 is looking to expand its popular game, *Draw It or Lose It*, beyond just Android users. They want a version of the game that can run on multiple devices and platforms through the web. Right now, the game is only available as an Android app, which limits its reach and player base.

To solve this, we’re proposing a web-based version of the game that uses Java for the back end and HTML5 technologies for the front end. This approach will let players access the game on phones, tablets, and computers without needing to install anything. We’re using design patterns like Singleton to make sure there’s only one active game manager at a time, and Iterator to keep things organized when working with lists of games, teams, and players. These choices will help us build a system that’s efficient, scalable, and easy to maintain.

## Requirements

**Business Requirements:**

* The game needs to work on different platforms (Windows, Mac, Linux, and mobile devices).
* Game names and team names must be unique so that players can’t accidentally use the same name twice.
* Teams must have multiple players.
* Players should be able to join teams within a game.
* Only one game service should run at a time.

**Technical Requirements:**

* Use object-oriented programming with Java.
* Use Singleton to ensure one game manager.
* Use Iterator to search lists of games, teams, and players.
* All entities should have unique IDs and names.
* Build the system in a way that supports web-based deployment.

## [Design Constraints](#_2et92p0)

Since this version of the game is being built as a web-based application, there are a few constraints that impact how we design and develop it.

First, it has to work across different operating systems and devices, which means we need to stick to standard web technologies that work in all modern browsers. We also need to keep things lightweight for mobile users, who may have slower internet connections or limited data plans.

Security is another key factor—we’ll need to use HTTPS and make sure user information is handled safely. On the development side, we must ensure the game and team names are unique. This requires careful searching of existing collections using iterators before adding anything new.

Lastly, since we’re only allowing one active instance of the game service, we’ll use the Singleton pattern to make sure only one version of that class exists in memory at any time.

## [System Architecture View](#_ilbxbyevv6b6)

For this project, the application is set up using three-layer architecture:

1. **Presentation Layer (Front End):** This is the web interface that users interact with, which will be built using HTML, CSS, and JavaScript.
2. **Application Layer (Back End):** This is the logic layer written in Java, where all the rules of the game, team management, and player tracking take place.
3. **Data Layer:** Right now, the data is held in memory using Java collections, but this can easily be replaced with a database if needed in the future.

This setup makes it easy to separate the visual part of the game from the logic and data, which helps with testing, updates, and scaling later on.

## [Domain Model](#_8h2ehzxfam4o)

**The domain model is designed to reflect the structure of the game. At the base, we have an Entity class that holds shared attributes like id and name. This allows other classes like Game, Team, and Player to inherit from it instead of repeating the same code.**

**GameService is where the actual games are stored. It's a Singleton, which means only one instance of it can be created. This ensures we don’t accidentally create multiple versions of the game manager.**

**Each Game contains a list of Teams, and each Team contains a list of Players. When we add a game, team, or player, we use an iterator to check if the name already exists, so we don’t create duplicates. This design follows object-oriented principles like encapsulation, inheritance, and composition. It keeps the code organized and makes it easier to manage the game structure as it grows.**

**"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)

Using your experience to evaluate the characteristics, advantages, and weaknesses of each operating platform (Linux, Mac, and Windows) as well as mobile devices, consider the requirements outlined below and articulate your findings for each. As you complete the table, keep in mind your client’s requirements and look at the situation holistically, as it all has to work together.

In each cell, remove the bracketed prompt and write your own paragraph response covering the indicated information.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Great for development and media editing but not commonly used in production hosting. More expensive. | Very reliable for server hosting; open-source and highly customizable. Often used in cloud environments. | Works well but can be more expensive to license. Slightly more maintenance heavy. | It is not ideal for hosting servers but could handle edge processing or run basic APIs in limited use cases. |
| **Client Side** | Safari sometimes needs extra testing but overall works well with modern browsers. | Works well for web apps; supports all major browsers. Low cost. | Good browser support. Windows users make up a large portion of the audience. | Web apps must be responsive and optimized for different screen sizes and limited network speeds. |
| **Development Tools** | Xcode, IntelliJ, and other IDEs available. Great for developers used to Apple’s environment. | Open-source tools like VS Code and Eclipse work well. Command-line tools are powerful. | Full support for Java and web tools like Visual Studio, IntelliJ, and Docker Desktop. | You can use tools like Android Studio and Chrome DevTools. Front-end testing requires simulators or real devices. |

## 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 is the best choice for deploying the backend. It's free, highly stable, and commonly used in cloud services. It also has great community support and works well with Docker containers and web servers.

1. **Operating Systems Architectures**:

A Linux-based, 64-bit operating system architecture will support the game service efficiently. It allows us to containerize the application and scale horizontally if needed.

1. **Storage Management**

**Right now, we're using in-memory storage with Java collections. But if the app expands, we recommend using a relational database like PostgreSQL to store game, team, and player data reliably.**

1. **Memory Management**:

The Java Virtual Machine (JVM) handles memory automatically using garbage collection. As long as we avoid memory leaks and manage our object lifecycles properly, this system should handle memory well for the game.

1. **Distributed Systems and Networks**:

The game can use web services to allow communication between users on different devices. For real-time updates (like showing drawings in progress), we could use WebSockets or Server-Sent Events to push updates from the server to all connected clients. If a server goes down, we can recover game state from a database or in-memory cache.

1. **Security**

We’ll secure all connections with HTTPS and encrypt any sensitive data. Unique user IDs and secure

password handling (like using salted hashing) will protect players. We’ll also validate all user input to

avoid injection attacks or bugs that could lead to exploits.