

Draw It or Lose It (Web Edition)

# **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 | 09/21/25 | Kimo Lizama | Initial draft of software design 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)

The Gaming Room wants to bring their game **Draw It or Lose It** to the web so people can play it on more than just Android. The new version needs to let teams and players join games, make sure names don’t repeat, and only keep one copy of the game service running at a time.

The plan is to build a simple Java program that has classes for **Game, Team, and Player**, with a base class called **Entity**. A **GameService** class will handle creating and keeping track of everything. To meet the rules, we’ll use the **Singleton pattern** (so only one GameService exists) and **Iterator pattern** (to check for duplicate names). This setup will make the game work now and also make it easier to add more features later.

## Requirements

* *The game must allow one or more teams.*
* *Each team must allow multiple players.*
* *Game and team names must be unique.*
* *There can only be one instance of the game service in memory.*
* *Each entity (game, team, player) must have a unique ID and a name.*

## [Design Constraints](#_2et92p0)

* **Web-based:** The program will run on a server and people will connect with browsers. This means the server has to manage all the game data.
* **One instance:** Only one GameService can run at once, so we’ll use a Singleton.
* **Unique names:** The system has to check names before adding new games, teams, or players.
* **Concurrency:** More than one person could use the system at the same time, so the code needs to handle that.
* **Portability:** The design should work on different operating systems.
* **Performance:** For now, data is stored in memory, which is fine for a small number of users. Later, we may need a database.

## [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)

* **Entity class:** Holds the id and name.
* **Game class:** Extends Entity, holds a list of teams.
* **Team class:** Extends Entity, holds a list of players.
* **Player class:** Extends Entity.
* **GameService:** Singleton that creates and manages games, teams, and players. It also makes sure names are unique.

**Object-Oriented principles used:**

* **Inheritance:** Game, Team, and Player all extend from Entity.
* **Encapsulation:** Data is private and accessed with getters.
* **Abstraction:** GameService hides the details of how things are stored.
* **Design Patterns:** Singleton (one GameService) and Iterator (to search for duplicates).

**"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** | Good for testing, but not common for real servers. | Best for hosting because it’s reliable and free. | Works but can be more expensive. | Not good for hosting; better as clients. |
| **Client Side** | Players can connect with browsers like Chrome, Firefox, or Safari. | Players can connect with browsers like Chrome, Firefox, or Safari. | Players can connect with browsers like Chrome, Firefox, or Safari. | Important since many players will use phones. Needs testing on both iOS and Android. |
| **Development Tools** | Java for the program  Eclipse or IntelliJ for coding  JUnit for testing  Safari/Chrome browsers for testing  Optional: Docker for easier setup | Java for the program  Eclipse or IntelliJ for coding  JUnit for testing  Chrome/Firefox browsers for testing  Docker and other Linux tools for easier setup | Java for the program  Eclipse, IntelliJ, or Visual Studio Code for coding  JUnit for testing  Edge/Chrome browsers for testing  Optional: Docker (via WSL2) for setup | Java back end connects to mobile browsers (Safari on iOS, Chrome on Android)  Browser developer tools for testing mobile layouts  Optional: Android Studio/Xcode for native testing later |

## Recommendations

1. **Operating Platform**

The best operating system for running *Draw It or Lose It* is **Linux**. It’s free, very reliable, and works great for web servers. Many companies use it because it can handle lots of users at once and rarely crashes. Linux can run tools like Apache Tomcat to host the Java program for the game. It also works well with cloud services like AWS or Google Cloud, which makes it easy to grow when more players join.

1. **Operating System Architecture**

Linux has two main parts — the **user space** and the **kernel**. The kernel is like the brain that controls memory, hardware, and how programs run. The user space is where programs like our game server run. This setup keeps everything organized and safe if something goes wrong. The *Draw It or Lose It* server will run the Java program in the user space, and players will connect to it using their web browsers. The browsers send requests over the internet, and the server responds with the game updates. This makes it easy for people to play on any device that has a browser.

1. **Storage Management**

Right now, the game can use **in-memory lists** to store information, like player names and game data. Later on, it should use a database such as **MySQL** or **PostgreSQL** to keep data safe even when the server restarts. A database will make it easier to check for duplicate names, manage logins, and back up data. Using cloud storage can also help save copies of the game data in case something goes wrong with the main server.

1. **Memory Management**

Java already has **automatic garbage collection**, which means it clears out old data when it’s no longer needed. This helps keep memory from filling up and slowing down the game. The system should also remove old or finished games from memory to stay fast. Caching can be used to keep things that are used often, like player lists, ready to go without loading them every time. Linux also helps by giving each process its own space in memory, so one program doesn’t crash another.

1. **Distributed Systems and Networks**

The game will work using a **client-server model**. Players connect from their browsers, and the server runs the main game program. The two talk to each other through the internet using **HTTPS**, which keeps the data safe. If the server ever goes down, another backup server can take over so players don’t lose connection. This setup makes it easy for the game to run smoothly on many devices at once, even if players are in different places.

1. **Security**

Keeping the players’ information safe is very important. The game should always use **HTTPS** so that all messages between the player and the server are encrypted. Passwords should be **hashed** (turned into unreadable code) before being stored. The server should also check all inputs to make sure hackers can’t send harmful code. Only trusted people should be able to access the database, and the server should use a **firewall** to block attacks. Updating the system regularly will help keep everything secure.

1. **Summary**

Using Linux with Java gives The Gaming Room a stable and affordable setup for *Draw It or Lose It*. It’s safe, easy to grow, and can handle many players. With good memory, storage, and security plans, the game will run smoothly on browsers and stay protected from online threats.