

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.2

## 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 | 07/20/2025 | Anthony Arnett | Initial Draft |
| 1.1 | 08/03/2025 | Anthony Arnett | Added evaluation and recommendations |
| 1.2 | 08/15/2025 | Anthony Arnett | Proofread and final evaluation before submission |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room seeks to develop a web-based version of its drawing game, *Draw It or Lose It*. This version uses a dynamic image-rendering system from a stock drawing library to convey clues. A typical game consists of four timed rounds where teams must guess the puzzle as the image progressively disappears.

To ensure scalability and performance in a web-based environment, the application is built using Java with object-oriented principles and key software design patterns. The solution needs to support unique team and game names, restrict the game to a single active instance, and allow team and player management within that instance. The Singleton and Iterator patterns are crucial to ensure system integrity, efficiency, and proper resource management.

## Requirements

*Functional Requirements*

* A game can hold one or more teams.
* Each team can hold multiple players.
* Game and team names must be unique.
* Only one live game can exist at a time.
* Must be able to create, find, and list teams and players

*Non-Functional Requirements*

* Runs in a web browser on desktop and mobile.
* Sketches updates smoothly and reach full detail at thirty seconds.
* Inputs are checked for safety.
* Structure is simple enough to add new features later.

## [Design Constraints](#_2et92p0)

We’ll use the Singleton pattern to make sure only one GameService exists. That way, no two games accidentally run at the same time. We also need to check each new name against the existing list before it is added, ensuring unique names. The game will need to be platform-neutral, ensuring it works on Windows, macOS, Linux, and mobile. Drawings are revealed gradually, so timing has to be perfect. We will need accurate timers that display accurately to all users at the same time.

## [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 *Entity* class is the base. It holds an ID and name. *Game, Team,* and *Player* all extend from it. A *Game* can have multiple *Teams*, and each *Team* has multiple *Players.* *GameService* is a Singleton, it manages the game and handles creating new teams and players.

*Game, Team,* and *Player* all share the *Entity* class structure, keeping things simple and reusable. We use a Singleton pattern to ensure only one *GameService* is active, and an Iterator Pattern to sort through lists of games, teams, and players when adding new ones, making sure names aren’t reused.

**"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 can be used as a server, but it's uncommon in production. It has great developer tooling and Unix compatibility, but Apple hardware is expensive and doesn’t scale well for large deployments. | Linux is the most popular OS for web hosting. It’s open-source, free, and highly customizable, making it ideal for scaling to thousands of users. It's also supported by all major cloud providers. | Windows Server integrates well with Microsoft tools but comes with licensing fees and uses more system resources than Linux. It’s less common for hosting web apps but may be useful if Microsoft integration is needed. | Mobile devices aren’t suitable for server hosting. They’re limited in hardware, power, and lack the infrastructure needed to serve content or manage high traffic. |
| **Client Side** | Safari needs extra testing since it sometimes behaves differently than other browsers. We’ll also need to verify responsiveness across various Mac screen sizes and ensure good performance. | Linux desktops are less common among general users, but common among developers. We need to make sure our browser-based app works well in Firefox and Chrome with good fallbacks. | Windows is the most common desktop OS, so performance and compatibility must be solid. Chrome and Edge are most used, so we need to test for small layout or accessibility issues across browsers. | Responsive design is crucial. We’ll need to test different screen sizes, support touch input, and monitor performance on low-end mobile devices. Browser compatibility and usability across Android and iOS are essential. |
| **Development Tools** | Xcode is needed for native iOS/macOS apps, but not required here. Macs support VS Code, IntelliJ, and terminal tools. Being Unix-based, they handle Linux-style workflows easily. No special licensing is needed for web development. | Linux supports nearly all development tools, compilers, and interpreters. Development is fast, customizable, and mostly free. Great for backend and scripting. | Windows supports major IDEs like Visual Studio, IntelliJ, and Eclipse. It works well for both frontend and backend work. Some tools may have licensing costs depending on the edition. | We’ll use mobile devices for testing responsiveness and user experience, but development is best done with emulators and simulators to avoid unnecessary device costs. |

## 

## Recommendations

1. **Operating Platform**: Linux is the best choice for hosting Draw It or Lose It. It’s free, reliable, and already standard across most cloud providers. It scales well, supports all the necessary tools, and won’t cost the client anything in licensing. For players, the game should run in the browser so it works on any device without needing to build native apps.
2. **Operating Systems Architectures**: We’ll use a standard Linux web server setup, probably something like Ubuntu or Debian, with a Java backend served through Tomcat or another lightweight server. The app will be accessed through the browser using HTML, CSS, and JavaScript, with all logic and game state handled on the server. This setup keeps everything centralized, easy to test, and simple to update.
3. **Storage Management**: We will store data in a relational database like PostgreSQL or MySQL. These are stable, widely supported, and scale well. Image files and other assets can live in cloud storage, giving us flexibility and fast load times.
4. **Memory Management**: Java handles memory automatically with garbage collection, but we’ll still design carefully to avoid memory issues. Using Singleton for the GameService helps control how much memory is used, and clearing out old data after each game round keeps memory usage low.
5. **Distributed Systems and Networks**: The server and client will communicate using REST APIs or WebSockets for real-time updates. Since it’s browser-based, users on different platforms can connect easily. We’ll make sure everything is stateless and cloud-ready so we can scale quickly. To avoid issues with outages, we’ll add caching, retries, and load balancing as needed.
6. **Security**: We’ll secure all traffic using HTTPS and sanitize all user inputs to prevent injection attacks. Sessions will be managed using secure tokens. On the server, we’ll isolate services using containers, keep everything up to date, and use firewalls and port restrictions. If we collect any personal data, we’ll follow basic privacy best practices from the start.