

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 | 7/14/2025 | Jazzlynn Gonzalez | I added code to make sure no two teams or players have the same name. I used a loop to check the list before adding a new one. This helps keep the game organized and avoids duplicates. I also followed best practices to keep the code clean and easy to understand. |

**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 turn their popular Android game Draw It or Lose It into a web-based version so more people can play it on different devices. Currently, the game needs a better way to keep track of all the games, teams, and players. It also needs to make sure no names get repeated and that everything runs smoothly behind the scenes. To solve this, use object-oriented programming. Use two patterns called Singleton and Iterator. Singleton to make sure there’s only one main GameService running the game. Iterator to loop through lists and check if names already exists. These patterns will help keep everything organized and easy to manage as the game grows.

## Requirements

The client, The Gaming Room, has both business and technical requirements for the game application Draw It or Lose It. From a business perspective, the application must support growth across multiple platforms, including desktop, mobile, and web environments. It should maintain a consistent user experience while scaling to accommodate a growing player base and high-performance demands. The application must also be reliable, secure, and responsive to keep users engaged and maintain the company’s competitive edge. On the technical side, the game needs to manage a large volume of high-resolution images, around 200 files at 8 MB each, which means memory and storage efficiency are critical. The system must also support real-time rendering of images during gameplay, fast load times, and minimal latency. Additionally, secure data transmission between platforms is essential, as is protection of user information, especially when dealing with account authentication or any personal data. The software should be designed with distributed systems in mind to allow communication and synchronization between different instances or platforms of the game.

## [Design Constraints](#_2et92p0)

Some design constraints include:

* Only one GameService allowed. There should only be one GameService in the whole app to keep everything in one place. This helps the app run smoothly and avoids confusion.
* Unique Names and IDs. Every game, team, and player needs a name and ID that no one else has. This keeps things from getting mixed up when people create or join games.
* It has to work on the web. Since the app will run on the internet, it needs to be built in a way that works well on websites and different devices, not just phones.
* It needs to be fast. The game should stay fast even when lots of people are playing at the same time. So writing a code that can handle big lists and run efficiently.
* No saved data yet. The app doesn’t save anything when it closes. Everything is stored in memory, which is ok for now, but will need to be improved later.

## [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 UML class diagram for The Gaming Room application outlines a clear, structured design for managing games, teams, and players using object-oriented programming principles. At the center of the structure is the Entity class, which serves as a base class for Game, Team, and Player. This demonstrates inheritance, as each of those classes inherits common properties like id and name from Entity, along with methods such as getId(), getName(), and toString(). This setup allows for cleaner, reusable code across different object types that share similar characteristics.

Each Game object can contain multiple Team objects, and each Team can contain multiple Player objects. These aggregation relationships show how complex game structures can be built from simpler parts. The GameService class acts as the central controller, maintaining a list of all games and generating unique IDs for games, players, and teams. It uses the Singleton design pattern, shown through the getInstance() method and private GameService constructor. This ensures that only one instance of the service exists across the application, which helps centralize control and avoid inconsistencies in data handling.

The ProgramDriver class runs the main application logic, while SingletonTester is used to validate the Singleton functionality of GameService. Altogether, this design reflects key object-oriented programming principles including inheritance, encapsulation, aggregation, and singleton usage. These choices make the code scalable, organized, and easy to maintain, an efficient structure for managing a multiplayer game environment like Draw It or Lose It.

**"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** | macOS can run servers like Apache and Nginx. It supports Java, Python, and other common languages. It is not usually used in production servers because licensing can be expensive and Apple hardware is required. | Linux is the most common platform for web hosting. It is fast, secure, and open sourced. It also supports all common web server tools and languages. It is ideal for scaling the application. | Windows supports IIS and other web technologies. It can host .NET and Java apps, but it may require paid licenses. It’s good for enterprise environments, but not as lightweight as Linux. | Mobile devices are not suitable for hosting web apps since they’re clients, not servers. They have limited storage, processing, and battery life making them poor server hosts. |
| **Client Side** | macOS users will access the app through modern browsers like Safari and chrome. Developers must test for visual consistency and compatibility. No major issues if using responsive HTML/CSS. | Linux desktop users will use browsers like firefox for chromium. If the app is built using web standards, it should work well. It may need some testing for UI quirks. | Windows users are the largest desktop market. The app must be tested in edge and chrome. Development tools are widely available. | Mobile clients need responsive design. It must work well in mobile browsers like chrome (Android) and Safari (IOS). Developers must test across many screen sizes and touch inputs. |
| **Development Tools** | macOS supports tools like xcode (for IOS), IntelliJ, Eclipse, and VS Code. Great for front-end and back-end development. Apple licenses may apply. | Linux supports open-source tools like Eclipse, IntelliJ, and vs code. Ideal for java and web dev. No licensing fees. Developers may need more command-line knowledge. | Windows supports many tools: Visual studio, Eclipse, IntelliJ, and others. Easy setup for beginners. Some tools may require licenses. | Mobile dev uses Android studio (Android) and xcode (IOS). Cross-platform tools like react native or flutter can reduce cost/time. Testing on real devices is critical. |

## 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**: I recommend using Linux as the server operating platform. It’s open source, super stable, and already trusted by companies like Google and Facebook. Linux handles high traffic well, and since the game might have tons of players drawing and guessing at once, it needs that kind of strength. It also supports all the popular tools and languages like Java, Python, and Node.js, which gives the development team a lot of flexibility without needing to pay extra for licenses like you would with Windows or macOS.
2. **Operating Systems Architectures**: Linux uses a monolithic kernel architecture, which means everything runs in the same memory space. While that sounds risky, it’s actually fast and efficient, especially with the right access controls in place. The architecture supports multitasking, multithreading, and smooth hardware interactions. All of which are important for a game that needs to load images quickly and track player interactions in real time. Linux also supports virtual memory, dynamic libraries, and modular kernel components, which helps optimize performance on different types of servers.
3. **Storage Management**: I recommend using a hierarchical file system on Linux like EXT4, possibly paired with a cloud-based object storage solution like AWS S3 or Google Cloud Storage to scale. This gives the flexibility to store static image assets on the cloud and keep the game lightweight locally, improving load times and server efficiency.
4. **Memory Management**: To make the game fast and responsive, especially when loading and displaying images at a high rate, Linux’s virtual memory management and paging techniques are crucial. Memory can be managed through segmentation, paging, and swapping. Linux uses demand paging, so it only loads data when it’s needed, saving memory. We can also use caching to keep the most frequently used images or data in RAM, reducing delays. For a smoother user experience, especially during gameplay, images should be preloaded into memory when the game starts then reused as needed.
5. **Distributed Systems and Networks**: To support communication between different platforms (phones, tablets, laptops, etc.) the game should be built using RESTful APIs and hosted on a distributed cloud network. Tools like Docker and Kubernetes will allow us to containerize the game services and scale them across multiple servers. If one node goes down, others can keep the game running without interruption. Using strategies like disk scheduling algorithms and redundancy in network storage ensures that game data is served quickly and safely. Distributed systems can also help players from different parts of the world connect with lower latency.
6. **Security**: We’ll use HTTPS for encrypted communication between players and the server and make sure all user data ( usernames, game progress, etc) is stored securely using hashed passwords and secure authentication tokens. Linux has built-in tools for setting user permissions, encrypting files, and isolating applications using containers. These features help protect the game from unauthorized access or data leaks. In case a server is attacked or goes offline, regular backups and firewall rules can limit damage and help us recover quickly. Managing interrupts, I/O control, and privileged access also helps reduce vulnerabilities in system interactions.