

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 | 09/20/2025 | Chris Grey | Initial draft of software design document for Draw It or Lose It |

**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 project involved designing a game application for The Gaming Room, based on the existing game *Draw It or Lose It*. The main problem was finding a way to manage games, teams, and players so that each one is unique and no duplicates are created. At the same time, the design needed to support running in a distributed, web-based environment where multiple users could interact with the system at the same time.

To solve this, the application was built in Java using a structure that ensures only one central game service is active at any point. This was done with the singleton pattern, which prevents multiple conflicting copies of the service from being created. The iterator pattern was also used to loop through lists of games, teams, and players so the system can check for duplicates before creating new ones. By applying these design patterns, the system keeps data consistent while supporting growth and scalability.

The application also follows object-oriented design by using a base Entity class that holds shared attributes like id and name. Other classes such as Game, Team, and Player inherit from this class. This reduces redundancy and makes the code easier to maintain. The result is a working game framework that is flexible enough to expand and reliable enough for multiple users to access without conflict.

This design will allow The Gaming Room to expand their game reliably across platforms while ensuring performance and consistency for users.

## Requirements

The client’s main business requirement is that the game application must support multiple users playing *Draw It or Lose It* in a web-based environment. Each game, team, and player must have a unique name so that duplicates are not allowed. The system should also allow users to check if a name is already in use before creating a new one.

On the technical side, the application must be designed to work in a distributed setting where multiple users can access the system at the same time. It needs to ensure that only one game service is active to prevent conflicts, and it must keep track of unique identifiers for games, teams, and players. The code should be easy to maintain and extend, following object-oriented principles and design patterns that make the application scalable.

## [Design Constraints](#_2et92p0)

When building the game application in a web-based distributed environment, there are a few key constraints to keep in mind. The singleton pattern is needed so that only one GameService instance runs at a time, keeping IDs consistent for games, teams, and players. The iterator pattern should be used to make sure names stay unique, which avoids duplicate games or teams when multiple users are adding them. Because the system runs in a distributed environment, data has to stay consistent even if many users are connected at once. At the same time, concurrency and potential latency must be managed so multiple users don’t cause conflicts when creating or joining games. The design also needs to scale so it can handle more games, teams, and players as the user base grows.

## [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 shows how the different parts of the game application are connected. At the center of the design is a base class that holds the common attributes id and name. The game, team, and player classes all inherit from this base class, which demonstrates the principle of inheritance. This allows shared features to be defined once and reused, keeping the code organized and easier to maintain. This also demonstrates encapsulation, since the attributes id and name are defined once in the base class and access through the derived classes in a controlled way.

The service class manages the creation of games and assigns unique identifiers for games, teams, and players. It is built as a singleton so that only one instance of the service exists at a time. This ensures that identifiers stay consistent and that the overall state of the game environment is managed in a controlled way. The design also uses the iterator pattern to check for duplicates before adding new games, teams, or players, which prevents name conflicts.

The relationships between the classes are straightforward. A game can contain many teams, and each team can contain many players. These one-to-many relationships show how the system is structured to support multiple levels of organization. Together, the use of inheritance, the singleton pattern, and the iterator pattern supports efficiency and helps meet the requirements for unique names and reliable game management.

**"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** | Macs aren’t really considered for servers in real-world setups. They’re expensive, not scalable, and I’ve rarely seen them used in production. They can technically run servers, but businesses almost never choose them for that purpose. | Linux is strong in server environments, especially for scalability and cost savings since it avoids licensing. I’ve come across plenty of Linux servers, but they do require more technical knowledge and can be harder to support in mixed business environments. | Windows Server is common in businesses that rely on Microsoft. It ties in well with Active Directory and the rest of the Microsoft stack, which makes it easier to manage security and users. Licensing adds cost, but the integration and support make it worthwhile. | Mobile devices don’t act as servers. The main concern is making sure the backend environment is tuned to handle mobile traffic and APIs so apps on iOS and Android connect reliably. |
| **Client Side** | Macs tend to show up in creative or dev-heavy teams. They’re not the main endpoint in most offices, but software needs to be tested for compatibility to avoid issues when those users are involved. | Linux desktops are rare for everyday users, though I’ve seen them with technical staff. The challenge is that support is inconsistent across distributions, and many apps don’t provide Linux versions, which means more testing and workarounds. | Windows is the standard on the client side. Most offices I’ve supported run almost entirely on Windows endpoints, so compatibility and adoption are much smoother. Users are comfortable with it, and it’s easier to manage at scale. | Mobile is a must. People expect apps to run seamlessly on both iOS and Android. That means accounting for screen size differences, OS updates, and cross-platform support. |
| **Development Tools** | Macs are required for iOS builds since you need Xcode. Outside of that, they’re not the go-to for enterprise development, though Java and web tools can run on macOS. | Linux pairs well with open-source tools like Eclipse, IntelliJ, or VS Code. It’s developer-friendly, but it’s less common in enterprise workflows that need standardized support and toolsets. | Visual Studio is widely used and fits well in enterprise environments. Windows also supports Eclipse and IntelliJ for Java development. It’s flexible and lines up with the tools I’ve seen used the most in business settings. | For mobile, Android Studio is used for Android apps and Xcode for iOS. Cross-platform tools like React Native or Flutter help reduce cost and time, but apps still need testing across actual 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**: I recommend Windows Server 2022. It’s widely used in businesses, easy to support, and integrates well with tools that most companies already rely on. It also has good vendor support, which helps when troubleshooting or setting things up.
2. **Operating Systems Architectures**: The Java service can be run as a Windows Service, which makes it easier to manage and keep running in the background. IIS can be used to handle web traffic if needed, which keeps things straightforward since many IT teams are already familiar with it.
3. **Storage Management**: Windows provides solid storage options like NTFS. Regular backups and snapshots can be scheduled to protect game data and make recovery easier if something goes wrong.
4. **Memory Management**: Windows handles Java workloads reliably. With basic monitoring, you can make sure the application has enough memory and continues to run smoothly as more players and teams are added.
5. **Distributed Systems and Networks**: Windows works well in both on-premises and cloud environments. It can support scaling as the application grows, and it’s flexible enough to handle connections across different client types.
6. **Security**: Security should include up-to-date encryption such as TLS 1.2 or higher, solid firewall rules, and routine patching with tools like WSUS or Intune. Active Directory or Azure AD can manage account access, while Windows Defender and BitLocker add protection at the endpoint and storage level. Following the principle of least privilege, accounts should only have the minimum access they need to do their job. This helps cut down on risks if a user account or service is compromised.