

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.0

## Table of Contents

[**CS 230 Project Software Design Template** 1](#_8yll8we3g9gk)

[**Table of Contents 2**](#_bigquqixolb6)

[**Document Revision History 2**](#_dw177wf6uc7o)

[**Executive Summary 3**](#_dvqgwb35dmo2)

[**Requirements 3**](#_16l1xgm63xiq)

[**Design Constraints 3**](#_8q9p7erj6kw8)

[**System Architecture View 3**](#_v3lbl36uymfp)

[**Domain Model 3**](#_8y70caddweq5)

[**Evaluation 4**](#_rskwclmig4s4)

[**Recommendations 5**](#_1sldlvtpnlq5)

## [Document Revision History](#_318ja9xg9jyt)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 06/21/25 | Trista Jacobs | I added the parts for Project Three and cleaned up the extra recommendations section at the end. |

**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](#_21gs3hve8vjy)

The Gaming Room is expanding their Android-only game, "Draw It or Lose It," into a cross-platform, web-based application. This software design document presents a solution that addresses system architecture, platform compatibility, memory and storage management, and distributed system considerations to ensure secure and scalable deployment.

## Requirements

*The Gaming Room seeks to expand their Android-only game "Draw It or Lose It" into a web-based, cross-platform application. This expansion requires a unified and scalable software design that maintains game state integrity, supports multiple teams and players, and ensures name uniqueness for games and teams. Our solution leverages object-oriented principles and design patterns such as Singleton and Iterator to manage game entities, enforce uniqueness, and ensure a single game instance in memory. This design provides a maintainable foundation for scaling the game to web and desktop platforms.*

## [Design Constraints](#_whc7661lxuef)

Developing the game as a web-based distributed system presents constraints such as:

* **Single Instance Enforcement:** Only one instance of the game should exist in memory. This is managed through the Singleton pattern in GameService.
* **Unique Identifiers:** Games, teams, and players must have unique names and IDs. This affects storage, retrieval, and UI validation logic.
* **Concurrency and Synchronization:** In a web-based environment, race conditions must be handled, especially when multiple users interact with game state simultaneously.
* **Cross-platform Accessibility:** The app must be deployable across different OSes (Windows, macOS, Linux) and browsers, requiring platform-agnostic technologies like Java and web APIs.
* **Scalability and Performance:** The system must support a growing number of concurrent users without degrading performance.

## [System Architecture View](#_jm0u3j65ug0r)

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](#_4z3cvuaqhz16)

The UML diagram showcases a domain model with clear object-oriented structure:

* The Entity class serves as a base class that encapsulates common properties (id, name) and behavior for Game, Team, and Player, promoting code reuse through inheritance.
* The GameService class implements the Singleton pattern, ensuring only one game service instance exists.
* The Game, Team, and Player classes all extend Entity, adhering to inheritance and polymorphism principles.
* Aggregation is used to model relationships:  
  + Game has a list of Team
  + Team has a list of Player
  + GameService manages a list of Game
* The addGame, addTeam, and addPlayer methods utilize the Iterator pattern to enforce name uniqueness before creation.

This design aligns with key OOP principles like encapsulation, inheritance, and modularity, fulfilling the requirements for unique names, single instances, and scalable structure.

**"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](#_xfvqtm9h07po)

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 offers robust development tools (e.g., Xcode, IntelliJ) and UNIX-based architecture compatible with Java-based servers. However, macOS servers are less common in production environments and may be costlier to maintain. | Linux is the preferred OS for web servers due to its open-source nature, stability, performance, and strong community support. It is highly compatible with Java applications, making it ideal for hosting this web-based game. | Windows servers support Java but typically require more resources. Licensing costs and system overhead are higher compared to Linux. However, Windows may be beneficial if Active Directory or other Microsoft services are used. | Mobile devices are not suited for hosting server-side applications but will function as clients that interact with the backend server. |
| **Client Side** | Development on macOS allows easy deployment to iOS platforms and web testing. However, it may require more time or emulators to test compatibility with non-Apple platforms. | Linux development is efficient and cost-effective for cross-platform tools (e.g., Java, web tech). However, it lacks robust tools for testing iOS. | Windows is widely used and supports a vast array of development tools. Testing web clients and desktop apps is straightforward, but Mac/iOS testing requires additional setup. | Mobile development introduces complexities like screen size variability and platform-specific UI. Testing requires emulators or multiple devices, increasing development time. |
| **Development Tools** | Java (Eclipse, IntelliJ IDEA), Maven, Git, and terminal tools are widely used. Mac also supports Docker for containerization. | Same tools as Mac; preferred for backend and server-side development. Excellent support for Jenkins, Git, and containerization with Docker. | Windows supports Eclipse, IntelliJ, Visual Studio, and WSL (Windows Subsystem for Linux) for running Linux-like environments. | Development uses Android Studio (for Android) or Xcode (for iOS). Cross-platform frameworks like Flutter or React Native can reduce workload. |

## 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 a good choice for the server because it’s free, fast, and works well with Java. It’s used a lot for hosting games and websites. It also works with tools like Docker, which makes it easier to manage and update the game.
2. **Operating Systems Architectures**: Linux uses something called a monolithic kernel, which just means it runs everything together for better speed. It can also load pieces only when they’re needed, which helps the system run better. It supports running different parts in separate areas, so one crash won’t mess everything up.
3. **Storage Management**: For game data like players and teams, using a database like MySQL or PostgreSQL works well. It keeps data organized and safe, even with a lot of users. Later, if the game grows, Linux can also use things like network storage or the cloud to handle more data.
4. **Memory Management**: Linux handles memory using virtual memory and paging. The Java Virtual Machine helps clean up memory the game doesn’t need anymore. The Singleton pattern also helps save memory by making sure there’s only one copy of the main game service running.
5. **Distributed Systems and Networks**: The game will use APIs so it can work across different devices like phones, computers, and tablets. It sends and receives data through the internet using regular networking (TCP/IP). Using things like cloud servers and load balancers will help keep the game running if one part goes down.
6. **Security**: All data will be sent using HTTPS so it’s encrypted. Players will log in with secure tokens like JWT so their accounts stay protected. Linux also has tools like SELinux to keep the server safe. Any info users type in will be checked so it can’t be used to break the system.