

# 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 | 11/15/24 | Jordan Walker | Initial draft created and completed. Added executive summary, design constraints, system architecture, domain model, evaluation, and recommendations. |

**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 take their Android game (Draw It or Lose It) and turn it into a web-based game that works on multiple platforms. To do this, we’re using software design patterns like Singleton and Iterator. The Singleton pattern makes sure there’s only one instance of the game service running at any time, which keeps things organized. The Iterator pattern helps us go through games, teams, and players efficiently and ensures that names are unique. This setup will help the system be scalable, efficient, and easy to work with as more users come on board.

## Requirements

The system needs to:

* Allow multiple games to run, each with one or more teams.
* Ensure each team has multiple players assigned to it.
* Guarantee unique names for games, teams, and players to avoid confusion.
* Use design patterns to make the system efficient and maintainable.
* Ensure only one instance of the game service exists at a time.

## [Design Constraints](#_2et92p0)

The following design constraints need to be addressed for the game application to work in a web-based distributed environment:

1. **Platform Support**: The system must run on multiple devices, such as desktops and mobile devices, meaning it needs to be platform-independent and responsive.
2. **Concurrency**: Many users will interact with the game at the same time, so the application must handle multiple concurrent requests efficiently.
3. **Security**: Since user data will be transmitted online, the system must include secure communication and user authentication.
4. **Scalability**: The system needs to grow over time, accommodating more users, teams, and games without slowing down.
5. **Unique Identifiers**: To prevent naming conflicts, all entities (games, teams, and players) must have unique names and IDs.
6. **Sufficient Storage for Game Assets:** Ensure that there is enough server capacity for storing a large library of images and assets required for the game.
7. **Timing and Throughput Optimization:** Highlight that efficient rendering and transmission of game assets to multiple users simultaneously is crucial.
8. **Load Balancing:** Mention the use of load balancers to evenly distribute traffic across servers, ensuring stability and high availability.
9. **Secure Communication and Authentication Mechanisms:** Include HTTPS for secure data transmission and token-based authentication for session management and user authentication.
10. **Platform Independence:** Reiterate the need for the design to function seamlessly across Linux, macOS, Windows, and mobile platforms, emphasizing a responsive and adaptable interface.

## [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 diagram shows how the classes and objects in the system are related:

1. **Entity Class**:
   * + This base class holds the shared attributes (id and name) for Game, Team, and Player. It simplifies the design and ensures consistency across these classes.
2. **GameService Singleton**:
   * + This class manages the games, teams, and players. By using the Singleton pattern, we ensure only one instance of GameService exists, centralizing the management of these entities.
3. **Relationships**:
   * + GameService contains a list of Game objects. Each Game contains a list of Team objects, and each Team contains a list of Player objects. This hierarchy organizes data logically and makes it easy to access related objects.
4. **Object-Oriented Principles**:
   * + **Encapsulation**: Attributes are private and can only be accessed through public methods.
     + **Inheritance**: Game, Team, and Player inherit common attributes and behaviors from the Entity class.
     + **Polymorphism**: Shared behavior across entities allows for easier management of the system.

**"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** | While macOS is stable, its higher cost and limited usage as a hosting platform can make it less ideal for large-scale deployments. It may face challenges in handling real-time synchronization under heavy traffic compared to Linux. | Linux remains the best option for hosting due to its widespread use in web applications, offering unmatched scalability and real-time synchronization capabilities. It handles concurrent connections effectively, ensuring smooth gameplay for thousands of users simultaneously. | Windows is user-friendly and widely supported, but its higher licensing costs and potential security concerns make it less competitive for hosting large-scale web applications. However, it is a good option for teams already integrated into the Windows ecosystem. | Relies on cloud servers for hosting. Efficient but depends on strong network connections. |
| **Client Side** | Great for development with Apple tools but has high hardware costs. | Cost-effective but requires Linux expertise. Best for experienced developers. | Beginner-friendly with many tools. Licensing costs can add up. | Requires frameworks like React Native or Flutter to support multiple platforms. Testing is key. |
| **Development Tools** | macOS supports Xcode, IntelliJ IDEA, and Eclipse. Ideal for cross-platform projects. | Linux has free tools like Eclipse and IntelliJ. Best for Java and open-source development. | Windows offers tools like Visual Studio and IntelliJ. Simple to use and versatile. | Development uses Android Studio, Xcode, or cross-platform tools like Flutter for versatility. |

## 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 the best choice for hosting because it’s affordable, reliable, and widely used for web apps. For development, tools like Eclipse work great and are cross-platform, so they’ll work on macOS, Linux, and Windows.
2. **Operating Systems Architectures**: A microservices setup is the way to go. It breaks the game into smaller parts, like player management, team management, and game logic, which makes it easier to update and scale as needed.
3. **Storage Management**: Use a relational database like MySQL for handling data like games, teams, and players. Adding something like Redis for quick data access will speed things up a lot.
4. **Memory Management**: Java’s built-in garbage collection is great for managing memory. Caching active game data will also cut down on database calls and keep the app running smoothly.
5. **Distributed Systems and Networks**: RESTful APIs are perfect for handling communication between the front-end and back-end. For real-time updates, WebSocket connections are a solid choice. Load balancers can help spread traffic evenly so nothing slows down.
6. **Security**: Always use HTTPS to keep everything secure. Token-based authentication like OAuth 2.0 will protect user sessions, and sensitive data should be encrypted both while it’s being sent and when it’s stored.
7. **Security and Scalability:** To secure the platform, rate limiting and firewall protection must be implemented to protect against “Denial of Service” attacks. For scalability, the system should adopt containerization and orchestration tools to dynamically manage server resources and scale based on demand.
8. **Performance Testing:** A robust cross-platform testing strategy must be adopted to ensure the application performs consistently across all operating systems and devices. This includes stress tests to evaluate the platform under high user loads.