

<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

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | <07/18/2025> | <Matthew Biletnikoff> | <Initial draft revised to complete all required sections, including Executive Summary, Design Constraints, and Domain Model. Integrated feedback to ensure alignment with client requirements and accurately described the UML class diagram. Formatting adjustments made to ensure clarity and professional presentation for submission.> |

**Instructions**

**Document Revision History**

## Executive Summary

<The Gaming Room has requested support in expanding their Android-only game, *Draw It or Lose It*, into a web-based platform accessible from multiple operating systems and devices. As a consultant from Creative Technology Solutions, I have reviewed the requirements and designed a solution that implements software design patterns such as singleton and iterator to manage unique game, team, and player instances efficiently. This software design will allow The Gaming Room to maintain a consistent game experience across platforms while improving scalability, performance, and security. The system is built with Java and is intended to support future integration with web frontends and cross-platform APIs..>

## Requirements

*<* **Business Requirements**:

* Allow multiple teams and players per game.
* Ensure that game and team names are unique.
* Enable the game to run across multiple platforms.

**Technical Requirements**:

* Only one instance of the game engine should be active in memory (singleton).
* Games, teams, and players must have unique identifiers.
* Maintain security of user data and communication.
* Use Java with platform-independent architecture for scalability.>

## [Design Constraints](#_2et92p0)

* <**Singleton Pattern**: The application must ensure that only one instance of the GameService class exists in memory. This is implemented using the singleton pattern to manage centralized control of game data. It limits memory usage and ensures data integrity.
* **Iterator Pattern**: To guarantee uniqueness of game, team, and player names, iterators are used to search existing entries before new objects are created.
* **Cross-Platform Development**: The application is built in Java for maximum portability. However, testing and debugging on each OS will require additional resources.
* **Web-Based Infrastructure**: The game must be hosted on a server that can serve multiple client types (desktop and mobile). This introduces constraints on networking protocols, data serialization (e.g., JSON/XML), and responsiveness.
* **Security Compliance**: Encryption and secure session management must be enforced to protect user data, which constrains development to frameworks that support modern authentication and encryption practices.>

## [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

<Describe the UML class diagram provided below. Explain how the classes relate to each other. Identify any object-oriented programming principles that are demonstrated in the diagram and how they are used to fulfill the software requirements efficiently.>

**"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.**

The UML class diagram for The Gaming Room application illustrates the structure and relationships among six key classes: Entity, GameService, Game, Team, Player, ProgramDriver, and SingletonTester.

**Class Relationships and Structure**

1. **Entity (Superclass):**  
   The Entity class acts as a base class for Game, Team, and Player. It encapsulates two common attributes:

* id: long
* name: String

This demonstrates the object-oriented principle of inheritance, allowing Game, Team, and Player to reuse common methods like getId(), getName(), and toString().

1. **GameService (Singleton):**  
   The GameService class manages the overall application state, tracking all games, teams, and players.

* It uses the singleton pattern with a private constructor and a static getInstance() method to ensure only one instance exists in memory.
* It includes lists such as games: List<Game> and counters for unique IDs.
* It provides methods like addGame(), getGame(), addTeam(), and addPlayer() to enforce uniqueness and manage objects efficiently.

1. **Game, Team, and Player (Hierarchical Composition):**

* A Game object holds a list of Team objects (teams: List<Team>).
* A Team object holds a list of Player objects (players: List<Player>).
* These relationships are modeled using composition, representing that a game has teams and each team has players.

1. **ProgramDriver and SingletonTester (Utility Classes):**

* ProgramDriver contains the main() method to launch the program.
* SingletonTester demonstrates that GameService is a singleton by testing multiple references to the same instance.

**Object-Oriented Programming (OOP) Principles Demonstrated**

* **Inheritance:**  
  Promotes reuse by allowing Game, Team, and Player to inherit shared attributes and methods from Entity.
* **Encapsulation:**  
  Data members such as lists and ID counters are private and accessed via public methods, protecting the internal state of each class.
* **Abstraction:**  
  Complexity is hidden through clean interfaces such as addGame() and getTeam(), while implementation details are encapsulated.
* **Singleton Pattern:**  
  Ensures that only one GameService instance exists, aligning with the requirement that only one game instance should reside in memory.
* **Composition:**  
  The "has-a" relationships (Game → Team → Player) support modular and scalable design through nested associations.

**Efficiency and Requirement Fulfillment**

This design:

* Enforces unique identifiers for all game-related entities.
* Supports data consistency through centralized management.
* Meets the requirement for single-instance game control using the singleton pattern.
* Models team and player hierarchies clearly through composition.
* Allows future scalability and ease of maintenance through strong OOP principles.

This UML structure provides a clear, logical blueprint for building a web-based version of *Draw It or Lose It*, ready for multi-platform support and future expansion.

## Evaluation

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** | < Mac is stable and developer-friendly, but has limited server hosting options compared to Linux.> | < Linux is ideal for hosting due to stability, scalability, and community support.> | < Windows servers are widely used in enterprise but may require licensing fees.> | < Mobile devices are not suitable for server hosting.> |
| **Client Side** | < Mac development requires Xcode and Swift expertise; cross-platform Java GUI libraries help minimize cost.> | < Linux offers flexibility with open-source tools; however, GUI rendering can be inconsistent.> | < Windows provides strong IDE support and easier setup for beginners.> | < Mobile requires use of SDKs (Android Studio, Xcode), and extra time to ensure responsive design.> |
| **Development Tools** | < IntelliJ IDEA or Eclipse for Java, with support for Mac-compatible SDKs.> | < Tools like IntelliJ, Eclipse, and NetBeans are fully supported and free.> | < Visual Studio Code and Eclipse are popular, reliable IDEs.> | < Android Studio (Java/Kotlin), Xcode (Swift), and cross-platform tools like Flutter or React Native.> |

**Server Side Evaluation**

**Linux**

* Characteristics: Robust, scalable with server platforms like Apache or Nginx. Excellent community support and open-source nature.
* Advantages: Free, secure, stable, and highly customizable, ideal for large-scale web hosting.
* Weaknesses & Licensing Costs: Requires advanced technical skills; however, there are no direct licensing costs due to its open-source model**.**

**Mac**

* Characteristics: Limited capabilities for server hosting, typically suited for development environments rather than large deployments.
* Advantages: User-friendly and stable, beneficial for small to medium-scale environments or development purposes.
* Weaknesses & Licensing Costs: Higher hardware and potential licensing costs; less practical for extensive web-server needs.

**Windows**

* Characteristics: Enterprise-grade, user-friendly server solutions with GUI-based administration.
* Advantages: Easy setup, wide support, strong integration with Microsoft services.
* Weaknesses & Licensing Costs: Licensing fees required, increasing costs significantly at larger scales.

**Mobile Devices**

* Characteristics: Not designed or suitable for hosting server applications.
* Advantages: None applicable for server-side hosting.
* Weaknesses & Licensing Costs: Insufficient resources, instability issues, and inappropriate infrastructure for hosting.

**Client Side Evaluation**

**Linux**

* Development Considerations: Requires cross-browser compatibility using HTML5, CSS3, and JavaScript.
* Advantages: Offers flexibility, stability, and an extensive set of open-source development tools.
* Weaknesses: GUI inconsistencies across different Linux distributions require extensive testing.

**Mac**

* Development Considerations: Necessary browser testing, especially Safari compatibility; intuitive graphical UI support.
* Advantages: Strong graphical interface, streamlined responsive web design testing.
* Weaknesses: Higher upfront investment in hardware, limited hardware configurations.

**Windows**

* Development Considerations: Extensive testing across multiple browsers; utilizes IDEs such as Visual Studio and VS Code.
* Advantages: Strong IDE support, broad hardware compatibility, easier learning curve for new developers.
* Weaknesses: Possible additional licensing costs for professional development tools.

**Mobile Devices**

* Development Considerations: Responsive web design critical; specialized frameworks and SDKs required (e.g., Android Studio, Xcode).
* Advantages: Broad user accessibility, widely used standardized development platforms.
* Weaknesses: Complex and extensive testing required due to varying screen sizes and device capabilities.

**Development Tools Evaluation**

**Linux**

* Languages & IDEs: Java, JavaScript; IntelliJ IDEA, Eclipse, NetBeans.
* Impact on Development Team: Requires moderate technical expertise; strong open-source community support.
* Licensing Costs: Minimal to none, predominantly open-source tools.

**Mac**

* Languages & IDEs: Java, JavaScript, Swift; Xcode, IntelliJ IDEA, Eclipse.
* Impact on Development Team: Enhanced workflow efficiency, especially for UI design and testing.
* Licensing Costs: Mostly free tools available; some proprietary tools may incur costs.

**Windows**

* Languages & IDEs: Java, JavaScript, C#; Visual Studio, IntelliJ IDEA.
* Impact on Development Team: Strong development support, easier onboarding for developers.
* Licensing Costs: Community editions free; professional editions and additional tools may require licensing fees.

**Mobile Devices**

* Languages & IDEs: Java/Kotlin (Android Studio), Swift (Xcode), React Native, Flutter.
* Impact on Development Team: High specialization needed; potentially separate teams for Android and iOS.
* Licensing Costs: Generally free IDEs; potential third-party or proprietary licensing fees.

## 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 recommended platform for server-side deployment due to its open-source nature, security, and extensive community support. For client development, Java ensures cross-platform compatibility with Windows, Mac, and mobile systems.
2. **Operating Systems Architectures**: Linux offers modular and stable architectures that are ideal for web servers. Java’s platform-independent architecture enables compiled bytecode to run on the JVM, ensuring consistent behavior across devices.
3. **Storage Management**: The system can use a relational database like MySQL or PostgreSQL. These provide ACID-compliant storage for game sessions, players, and team records. Database abstraction can be achieved through Java’s JDBC API or ORM libraries like Hibernate.
4. **Memory Management**: Java’s built-in garbage collector automatically manages memory allocation and deallocation, reducing memory leaks and ensuring efficient memory use for objects like games, teams, and players during runtime.
5. **Distributed Systems and Networks**: A RESTful API backend can be deployed on a cloud server. It allows communication between web clients and mobile apps over HTTPS. Load balancing, failover protocols, and distributed caching (e.g., Redis) will ensure scalability and minimize downtime.
6. **Security** User sessions and data must be protected using secure HTTPS communication, token-based authentication (e.g., JWT), and encryption of sensitive data at rest and in transit. Role-based access control should be implemented for different game user types.