

The Gaming Room

# **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 | 01/25/25 | Sarah Dowd | Initial Software Design Document |
| 2.0 | 2/21/25 | Sarah Dowd | Evaluation: Server & Client Sides, Development Tools |
| 3.0 | 3/2/25 | Sarah Dowd | System Recommendations |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room has requested a web-based application for their game, "Draw It or Lose It," which expands upon the functionality of the original concept. The application will support multiple teams, each with multiple players, ensuring unique names for games, teams, and players. Only one instance of the game engine will exist at a time, accomplished through a singleton pattern. This design document outlines the requirements, constraints, domain model, and recommendations to guide the development process.

The proposed solution involves implementing object-oriented programming (OOP) principles, such as inheritance, encapsulation, and design patterns, to ensure scalability, maintainability, and robustness. The system will be developed to run in a web-based distributed environment, allowing flexibility for future expansion to other platforms.

### Requirements

1. **Business Requirements**:
   * Allow multiple teams, each with multiple players.
   * Ensure unique names for games, teams, and players.
   * Maintain one instance of the game engine.
2. **Technical Requirements**:
   * Utilize a singleton pattern for game engine management.
   * Enforce uniqueness for names using iterators.
   * Use object-oriented principles to reduce redundancy and improve maintainability.

## [Design Constraints](#_2et92p0)

1. **Distributed Environment**:
   * The game must function in a distributed web-based environment, ensuring compatibility with various operating systems and devices.
   * Implication: Requires robust network communication protocols and data synchronization between clients.
2. **Scalability**:
   * The application must accommodate future expansions, such as additional game types or player capabilities.
   * Implication: The system must be modular, with reusable components.
3. **Performance**:
   * Rendering of game clues must be efficient and complete within 30 seconds per round.
   * Implication: Optimize resource allocation and database queries to avoid delays.
4. **Security**:
   * User data and game state must be protected from unauthorized access.
   * Implication: Implement encryption and secure communication protocols.

## [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 illustrates the relationships between the main classes: Entity, Game, Team, and Player.

* **Entity**:
  + A base class holding common attributes (id and name) for all other classes. It promotes code reuse and consistency.
* **Game**:
  + Represents a game instance. Contains a list of Team objects and enforces unique team names through iterators.
* **Team**:
  + Represents a team within a game. Maintains a list of Player objects and ensures player name uniqueness.
* **Player**:
  + Represents an individual player. Inherits from Entity to reduce redundancy.
* **IdGenerator:**
  + A dedicated utility class, IdGenerator, has been added to centralize the generation of unique identifiers for games, teams, and players. This class provides a synchronized method to ensure thread-safe ID generation and eliminates redundancy in managing separate ID counters across multiple classes. By abstracting this functionality, the system becomes more modular and easier to maintain.

**OOP Principles in Use**:

1. **Inheritance**: Reduces redundancy by consolidating shared attributes in the Entity class.
2. **Encapsulation**: Attributes like id and name are private, with public getters.
3. **Singleton Pattern**: Ensures only one instance of GameService exists at runtime.
4. **Iterator Pattern**: Simplifies traversal of collections to enforce uniqueness.

**"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)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Suitable for development with Xcode and Java. High upfront costs but good integration with Apple ecosystems. | Preferred for hosting due to stability, scalability, and cost-efficiency. | Widely used and compatible with many tools. However, it is more resource-intensive and prone to security risks. | Limited capability for server-side hosting, but useful for testing client-side integration. |
| **Client Side** | Development is streamlined with Xcode for Mac users. Higher costs for hardware and software licenses. | Open-source tools make development cost-efficient. Strong community support for troubleshooting. | Broad developer familiarity but higher licensing fees and slower updates compared to open-source options. | Essential for building native mobile applications (iOS and Android). Requires separate tools like Android Studio and Xcode. |
| **Development Tools** | Java, IntelliJ IDEA, Eclipse. | Java, Eclipse, Vim. | Java, IntelliJ IDEA, Visual Studio. | Android Studio, Xcode for native apps. |

## 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**: Use a Linux-based server for hosting due to its stability, scalability, and cost-effectiveness.
2. **Operating Systems Architectures**: Implement a client-server architecture with a RESTful API to handle communication between the front-end and back-end.
3. **Storage Management**: Use a relational database like MySQL or PostgreSQL to manage games, teams, and players with constraints for unique names and ids.
4. **Memory Management**: Use efficient garbage collection in Java to manage memory. Design data structures to minimize memory overhead.
5. **Distributed Systems and Networks**: Use HTTPS for secure communication between client and server. Employ load balancing for scalability and redundancy.
6. **Security**: Implement encryption (e.g., TLS/SSL) for data transmission. Secure user authentication and data storage with hashing algorithms like bcrypt.