

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 | 05/25/2025 | Sophie Biondolillo | Proposed product architecture |
| 2.0 | 06/08/2025 | Sophie Biondolillo | Expanded on development evaluation |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room wants to produce a web-based version of their proposed game *Draw It or Lose It*, an interactive multiplayer game where teams compete to solve puzzles with rendered images. The project requires a development approach that makes effective software while accommodating for future hardware requirements. To meet the client’s specifications, the application must support multiple teams, generate unique puzzles and enforce unique team names, and ensure that only one instance of the game exists in memory at a given time.

To achieve this, our solution proposes web-based architecture that generates singular identifiers for each game instance, team, and player, preventing duplicate entries. The system will be designed with room to grow in mind, allowing for future improvements while maintaining stability as more users engage with it.

## Requirements

* A database system to store and manage unique identifiers for games, teams, and players.
* API-driven architecture to handle interactions between client-side applications and server-side logic.
* An infrastructure capable of handling concurrent users and maintaining performance integrity.
* UI/UX design that allows ease of use for players and administrators.

## [Design Constraints](#_2et92p0)

* The system must generate and track unique identifiers for game instances, teams, and players to prevent duplication and conflicts. This impacts database design and performance optimization strategies.
* Multiple players interact in real time, so the application must manage concurrent connections and mitigate synchronization issues.
* The system should be designed to support growing numbers of users without lagging. Cloud-based architecture could be implemented to manage the large amounts of data in use.
* To comply with privacy acts, encryption protocols and authentication must be used to keep user information secure.
* Only one game instance should exist in memory at a time. This requires a session handling program to track active games.

## [Domain Model](#_8h2ehzxfam4o)

Class Relationships:

* ProgramDriver: Acts as the entry point of the application.
* SingletonTester: Enforces a singleton pattern, preventing multiple instances of main game aspects.
* Entity: A base class with attributes id and name, initializing shared properties for Game, Team, and Player.
* GameService: Manages all game instances, generating unique identifiers for games, teams, and players while enforcing the single-instance constraint.
* Game: Represents a game session, containing multiple teams competing to image puzzles.
* Team: Represents a group of players competing to solve the puzzle within the concrete time constraints.
* Player: Represents a user assigned to a team.

Object-Oriented Programming Principles:

* Inheritance: This can be seen in the Entity class as it provides a reusable foundation for Game, Team, and Player.
* Encapsulation: Each class contains private attributes with getter and setter methods for data protection and controlled access.
* Singleton Pattern: The GameService class implements the Singleton pattern to make only one instance of the game system exist in memory at a given time. This fulfills the client’s requirement to prevent duplicate game sessions.
* Unique Identifiers: The system assigns unique identifiers (id) to each instance of Game, Team, and Player to prevent conflicts.

**"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** | Mac provides a stable Unix-based environment but has limitations in enterprise-level server hosting. It lacks industry adoption for web hosting due to fewer dedicated server solutions. | Linux is the preferred choice for web hosting, offering flexibility and security. It supports various server technologies like Apache or Docker. | Windows Server supports enterprise environments well historically. It has compatibility with Microsoft-based technologies (IIS, .NET) but is less commonly used for large-scale web applications. | Mobile devices are generally not suitable for hosting server applications due to hardware constraints, battery life issues, and limited processing power. Cloud-based solutions integrated with mobile applications are recommended. |
| **Client Side** | Developing for Mac requires compatibility with Safari and macOS frameworks. Costs are relatively high, and expertise in Apple’s development process is required. | Linux clients are less common, and compatibility with open-source browsers and frameworks like GTK or Qt is necessary. Most Linux users expect flexibility. | Windows remains the most widely used client-side platform. Development should focus on compatibility across multiple versions as they deploy. Its large user base makes cross-platform testing simpler. | Developing for mobile requires consideration of different operating systems (iOS, Android), screen sizes, or touch interfaces. Mobile-friendly UI/UX design is important. |
| **Development Tools** | Mac developers use Xcode, Swift, and cross-platform tools like Node.js, React, or Unity for web-based applications. Popular databases include MySQL and PostgreSQL. Utilizing Mac would require a very specific set of professionals familiar with the tools, as Mac development does not translate well from more popular programming languages, such as Java or Python. Licensing would be costly as well depending on volume, as SQL database access can be in higher demand. | Linux supports many development tools like Python, JavaScript, PHP, MySQL, and Node.js. IDEs such as VS Code and JetBrains tools are widely used. Linux is a top choice for web and backend development. Linux would support the collaboration of a multi-focused group of development professionals, as there are many tools at their disposal and few licensing restrictions against the suite of development tools. | Windows supports .NET, Java, and JavaScript-based development. Microsoft Visual Studio is a dominant IDE. Windows also integrates well with cloud services like Azure. Microsoft has a highly curated set of development UIs, which although can be pricey with licensing costs, would allow for a less experienced team to create an advanced application. | Mobile development requires tools like Android Studio (for Android), Xcode (for iOS), React Native, Flutter, or Unity for cross-platform functionality. Each development tool is highly abstracted for a less experienced team, but backend access costs can add up with different types of mobile devices in play. |

Recommendations

1. **Operating Platform**: I recommend **Linux** as the primary operating platform for hosting *Draw It or Lose It*. Linux is widely used for web applications due to its stability, security, and flexibility. It allows The Gaming Room to expand the game across other environments, including Windows, macOS, and mobile platforms with web-based deployment. Kernel-level virtualization support via KVM allows efficient resource isolation and cost-effective scaling of game server instances.
2. **Operating Systems Architectures**: The Linux platform operates on a multi-user, multi-tasking architecture with strong process isolation and robust memory management. It supports distributed computing and containerized environments using Docker or Kubernetes, making it an ideal choice for reliability. Additionally, Linux's POSIX-compliant system enhances compatibility with diverse applications. The kernel design provides preemptive scheduling and control groups (cgroups) ensure each game process receives predictable CPU and memory resources for consistent player experience.
3. **Storage Management**: A cloud-based storage solution such as Amazon S3, Google Cloud Storage, or Azure Blob Storage is ideal for managing game-related assets, including images, player data, and session history. A relational database (ex. MySQL or PostgreSQL) can handle structured game data, while NoSQL solutions like MongoDB can manage unstructured data.
4. **Memory Management**: Linux employs virtual memory management. This means that the operating system dynamically allocates resources based on application demand. Garbage collection techniques within programming frameworks (ex. JavaScript's V8 engine) help manage memory. This service provides geo-redundant storage, automated backups, encryption at rest, and lifecycle policies to optimize costs, and integration with Kubernetes for seamless containerized deployments.
5. **Distributed Systems and Networks**: For cross-platform communication, the game should use RESTful APIs or WebSockets as real-time data exchanges between clients and the server. A content delivery network (CDN) keeps access low-latency for remote players. Load balancing strategies such as Nginx or HAProxy prevent server overload during peak usage. Adopting a microservices architecture allows independent scaling and updates, while service discovery tools (e.g., Consul or Kubernetes DNS), load balancers like Nginx or HAProxy, and a CDN ensure low-latency, resilient connections even during partial outages.
6. **Security**: Implementing TLS/SSL encryption secures player interactions, while OAuth 2.0 or JWT-based authentication keeps user login safe. Database security such as role-based access control (RBAC) and input validation help prevent SQL injection attacks. Regular testing should be conducted to identify and address vulnerabilities.