

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 | <mm/dd/yy> | Cameron DeShong | <Brief description of changes in this revision> |

**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 aims to develop a web-based version of the game "Draw It or Lose It," a team-based drawing and guessing game that provides an interactive and engaging experience for its users. The game involves four rounds of play, with each round lasting one minute. Drawings serve as visual clues to help teams guess puzzles within the time limit. To meet client requirements, the application must support one or more teams with multiple players per team, enforce unique names for games and teams, and ensure that only one instance of the game exists in memory at any given time. Additionally, the game must leverage a web-based, distributed environment to accommodate a broad user base and scalable gameplay.

## Requirements

*<* Please note: While this section is not being assessed, it will support your outline of the design constraints below. *In your summary, identify each of the client’s business and technical requirements in a clear and concise manner.>*

## [Design Constraints](#_2et92p0)

1. **Web-Based Distributed Environment**
   * **Constraint**: The game must operate in a distributed, web-based environment to ensure accessibility across various devices and platforms.
   * **Implications**:
     + The application must be developed using web technologies, such as HTML, CSS, JavaScript, and a backend framework (e.g., Node.js, Python/Django, or Java/Spring Boot).
     + Cloud-based hosting services (e.g., AWS, Azure, or Google Cloud) should be considered for scalability and availability.
     + APIs should be implemented to support communication between the frontend and backend.
2. **Unique Identifiers for Games, Teams, and Players**
   * **Constraint**: Each game, team, and player must have a unique identifier to prevent duplication and conflicts.
   * **Implications**:
     + A database management system (e.g., MySQL, PostgreSQL, or MongoDB) must be used to store and manage data.
     + Validation logic will be required to ensure uniqueness when creating games, teams, or players.
     + Consideration should be given to using UUIDs (Universally Unique Identifiers) for entities to ensure global uniqueness.
3. **Single Instance of the Game in Memory**
   * **Constraint**: Only one instance of the game can exist in memory at any given time.
   * **Implications**:
     + Implement the Singleton design pattern for the game instance to enforce this constraint.
     + Use server-side logic to maintain a single active game session while allowing multiple teams and players to connect.
     + Ensure thread safety and concurrency management in the backend to handle simultaneous requests.
4. **Scalability and Performance**
   * **Constraint**: The application must support multiple teams and players while maintaining optimal performance.
   * **Implications**:
     + Optimize database queries and indexing to handle large volumes of data efficiently.
     + Implement caching mechanisms (e.g., Redis or Memcached) to reduce server load and improve response times.
     + Use load balancers to distribute traffic evenly across server instances.
5. **Cross-Platform Compatibility**
   * **Constraint**: The game must be accessible on various devices, including desktops, tablets, and smartphones.
   * **Implications**:
     + Employ responsive web design principles to ensure a consistent user experience across devices.
     + Test the application on multiple browsers and screen sizes to identify and resolve compatibility issues.
     + Consider Progressive Web App (PWA) features to enhance accessibility and offline support.

## [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 class diagram represents the structure of the **Draw It or Lose It** game application. At the core is the **Entity** superclass, which provides shared attributes like id and name to all its subclasses: **Game**, **Team**, and **Player**. A **Game** consists of multiple **Teams**, and each **Team** includes multiple **Players**.

The **GameService** class, following a singleton pattern, manages the creation and lifecycle of **Game** instances, maintaining references to them. It has a composition relationship with the **Game** class, which similarly connects to the **Team** class, and then to the **Player** class. The **ProgramDriver** class acts as the application’s entry point, using the singleton **GameService** instance to manage games, teams, and players. It also relies on the **SingletonTester** class to confirm the singleton behavior.

The diagram demonstrates key object-oriented principles. Inheritance is shown through the **Entity** superclass, which reduces redundancy and ensures consistency among its subclasses. Encapsulation is evident in how the **GameService** class restricts access to its attributes, promoting controlled interactions. Abstraction simplifies interactions between classes, hiding unnecessary details while emphasizing essential functionality.

**"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** | macOS provides a stable Unix-based environment, supporting Apache, Nginx, and other web servers. However, it has limited enterprise-level hosting support and licensing costs may be high. | Linux is the most preferred server OS due to its open-source nature, stability, and extensive hosting support. It offers various distributions tailored for web hosting, such as Ubuntu Server and CentOS. | Windows Server supports IIS for web hosting and integrates well with Microsoft technologies. However, licensing costs can be high and it is generally less efficient than Linux for web-based applications. | Mobile devices are not ideal for hosting server applications due to hardware and software limitations. They are better suited as clients connecting to the web-based application. |
| **Client Side** | Developing for macOS requires expertise in Swift and macOS frameworks. Cross-platform web applications can be built using Electron or standard web technologies. macOS users primarily use Safari, requiring additional testing for browser compatibility. | Linux users generally rely on web applications accessed via browsers like Firefox and Chromium. Developing a native Linux client requires expertise in GTK, Qt, or Java. | Windows is widely used, making it essential for client support. Applications can be developed using .NET, Electron, or traditional web technologies. Testing must ensure compatibility with Edge, Chrome, and Firefox. | Mobile app development requires expertise in Swift (iOS) and Kotlin/Java (Android). Cross-platform frameworks like Flutter or React Native can help streamline development. Responsive web design is crucial for mobile browser accessibility. |
| **Development Tools** | Xcode is the primary IDE for macOS development. Web-based applications can use VS Code, WebStorm, or Sublime Text. Licensing costs for Apple development tools are minimal, but macOS hardware is required. | Linux supports a wide range of open-source development tools such as VS Code, Eclipse, and JetBrains IDEs. It provides flexibility and cost-efficiency, making it ideal for development. | Windows offers Visual Studio, VS Code, and JetBrains IDEs for software development. While some tools have licensing costs, Windows remains a widely used platform for development. | Android Studio (for Android) and Xcode (for iOS) are necessary for mobile app development. Web technologies using React Native or Flutter can reduce the need for multiple development teams. |

## 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**: For The Gaming Room to expand "Draw It or Lose It" to other computing environments, a Linux-based server platform is recommended. Linux offers superior stability, scalability, and cost efficiency compared to other operating systems. It supports a wide range of web technologies, making it ideal for hosting web-based applications. Ubuntu Server or CentOS would be excellent choices for managing the game’s backend operations due to their security, performance, and extensive community support.
2. **Operating Systems Architectures**: The recommended operating platform architecture follows a client-server model, where the backend runs on a Linux server and the frontend is accessed via web browsers on various devices. Linux’s modularity allows for a lightweight, efficient system tailored to the game’s requirements. The architecture leverages microservices and containerization (e.g., Docker, Kubernetes) to ensure easy scalability and maintainability. The game logic, authentication, and real-time communication components can be deployed as separate services for optimal performance.
3. **Storage Management**: A relational database management system (RDBMS) like PostgreSQL or MySQL is recommended for structured data storage, ensuring fast queries and data integrity. Additionally, a NoSQL database like MongoDB can be used for storing real-time game states and player interactions. Cloud-based storage solutions such as Amazon S3 or Google Cloud Storage can be utilized for game assets, ensuring efficient content delivery and redundancy.
4. **Memory Management**: The Linux operating system handles memory management effectively using virtual memory, paging, and swapping. To optimize performance for "Draw It or Lose It," the backend services should utilize caching mechanisms such as Redis or Memcached to store frequently accessed data. Garbage collection and memory allocation optimization techniques should be implemented within the application to prevent memory leaks and enhance performance.
5. **Distributed Systems and Networks**: To ensure cross-platform compatibility and seamless communication, the game should adopt a cloud-based distributed architecture. Load balancers can be used to distribute incoming traffic across multiple server instances, preventing bottlenecks. WebSockets or gRPC can facilitate real-time multiplayer interactions across different platforms. Content Delivery Networks (CDNs) will help deliver assets efficiently, reducing latency for players across different geographic locations. Network redundancy and failover strategies should be in place to handle potential outages and ensure high availability.
6. **Security**: Security is paramount for protecting user data and maintaining trust. Secure communication protocols such as HTTPS and TLS should be enforced to encrypt data transmission. Authentication mechanisms like OAuth 2.0 or JWT (JSON Web Tokens) should be implemented for secure user login. Role-based access control (RBAC) can help restrict unauthorized access to sensitive features. Firewalls, intrusion detection systems (IDS), and regular security audits should be incorporated to prevent cyber threats. Data backups and disaster recovery plans must be established to ensure quick recovery in case of failures.