

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.7

## 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 4**](#_Toc115077322)

[**System Architecture View 7**](#_Toc115077323)

[**Domain Model 11**](#_Toc115077324)

[**Evaluation 15**](#_Toc115077325)

[**Recommendations 19**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 09/18/2024 | Brett Plemons | Initial draft of software design document |
| 1.1 | 09/19/2024 | Brett Plemons | Added Executive Summary and Design Constraints |
| 1.2 | 09/20/2024 | Brett Plemons | Completed Domain Model section |
| 1.3 | 09/20/2024 | Brett Plemons | Added Evaluation section for all platforms |
| 1.4 | 09/21/2024 | Brett Plemons | Finalized Recommendations section and overall review |
| 1.5 | 10/4/2024 | Brett Plemon | Client-Server Considerations & Comparisons |
| 1.6 | 10/12/2024 | Brett Plemons | Memory Management Considerations and Approaches (Separate Document) |
| 1.7 | 10/20/2024 | Brett Plemons | Complete “Recommendations” section |

**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, a client of Creative Technology Solutions (CTS), is seeking to expand their Android-based game "Draw It or Lose It" into a web-based application that can serve multiple platforms. This software design document outlines our proposed solution to meet their requirements and streamline the development process.

"Draw It or Lose It" is a team-based guessing game inspired by the classic TV show "Win, Lose or Draw". The game displays stock images as clues, which teams must use to guess a phrase, title, or object. Each game consists of four one-minute rounds, with images fully revealed at the 30-second mark. If the active team fails to guess correctly within the time limit, other teams have a 15-second window to make one guess each.

## Requirements

My proposed solution will transform this concept into a robust, web-based application capable of running on various platforms. Key features of our design include:

* **Multi-platform compatibility:** The game will be accessible via web browsers on desktop and mobile devices, expanding its reach beyond the current Android-only version.
* **Scalable team structure:** The application will support multiple teams, each containing multiple players, adhering to the client's requirements.
* **Unique identifiers:** We will implement a system to ensure unique game and team names, preventing duplicates and enhancing user experience.
* **Singleton pattern implementation:** To meet the requirement of having only one game instance in memory at a time, we will utilize the Singleton design pattern, creating unique identifiers for each game, team, and player instance.
* **Responsive design:** The user interface will be designed to provide an optimal experience across various device sizes and orientations.
* **Efficient asset management:** A system for managing and serving the extensive library of stock drawings will be implemented to ensure smooth gameplay and minimal load times.

By leveraging web technologies and following best practices in software design, we aim to create a scalable, efficient, and engaging multi-platform version of "Draw It or Lose It". This solution will not only meet The Gaming Room's current requirements but also provide a solid foundation for future enhancements and feature additions.

## [Design Constraints](#_2et92p0)

In developing the web-based version of "Draw It or Lose It" for The Gaming Room, we must consider several design constraints inherent to web-based distributed environments. These constraints, and their implications on application development, are as follows:

* **Cross-platform compatibility:**
  + **Constraint:** The application must function consistently across various operating systems and devices.
  + **Implication:** This requires using web technologies that are widely supported, such as HTML5, CSS3, and JavaScript. It may limit the use of certain cutting-edge features that are not universally supported.
* **Browser variations:**
  + **Constraint:** Different web browsers may interpret and render web technologies differently.
  + **Implication:** Extensive testing across multiple browsers (Chrome, Firefox, Safari, Edge, etc.) will be necessary. This may increase development time and complexity.
* **Network latency and reliability:**
  + **Constraint:** Users may have varying internet connection speeds and reliability.
  + **Implication:** The application must be designed to handle network interruptions gracefully and optimize data transfer to minimize the impact of latency on gameplay.
* **Stateless nature of HTTP:**
  + **Constraint:** Web applications typically use HTTP, which is stateless by nature.
  + **Implication:** We need to implement robust session management techniques to maintain game state, possibly using technologies like WebSockets for real-time updates.
* **Security considerations:**
  + **Constraint:** Web applications are exposed to various security threats.
  + **Implication:** Implementation of security best practices is crucial, including data encryption, secure authentication, and protection against common web vulnerabilities (e.g., XSS, CSRF).
* **Scalability requirements:**
  + **Constraint:** The application must handle multiple concurrent games and users.
  + **Implication:** This necessitates efficient server-side architecture and possibly the use of load balancing and database optimization techniques.
* **Mobile device limitations:**
  + **Constraint:** Mobile devices have smaller screens and potentially less processing power.
  + **Implication:** The UI must be responsive and the application optimized for performance on mobile devices, which may influence design decisions.
* **Data persistence:**
  + **Constraint:** Game data must be stored and retrieved efficiently in a distributed environment.
  + **Implication:** Careful consideration of database design and caching strategies is required to ensure quick access to game data without overloading the system.
* **Asset management:**
  + **Constraint:** The game relies on a large library of stock drawings.
  + **Implication:** An efficient system for storing, retrieving, and caching these assets must be developed to ensure quick load times and smooth gameplay.
* **Single instance requirement:**
  + **Constraint:** Only one instance of the game can exist in memory at any given time.
  + **Implication:** This requires careful implementation of the Singleton pattern and consideration of how this will work in a distributed, multi-user environment.

These design constraints significantly impact the development process, requiring careful planning and architecture design. They influence technology choices, development practices, and testing strategies. By addressing these constraints proactively, we can create a robust, scalable, and user-friendly web application that meets The Gaming Room's requirements while providing a solid foundation for future growth and enhancement.

## [System Architecture View](#_ilbxbyevv6b6)

The “Draw It or Lose It” web application employes a multi-tiered, distributed architecture to ensure scalability, performance, and maintainability. This section outlines the system and subsystem architecture, including physical components, tiers, and the logical topology of communication and storage aspects.

**Physical Components and Tiers**

1. **Client Tier**
   1. Web browsers (Desktop and Mobile).
   2. Progressive Web App (PWA) Containers.
2. **Presentation Tier**
   1. Web Servers (Apache Tomcat).
   2. Static Content Delivery Network (CDN).
3. **Application Tier**
   1. Java EE Application Server (e.g., WildFly or GlassFish).
   2. Spring Boot Microservices.
   3. WebSocket implementation (using Java WebSocket API).
4. **Data Tier**
   1. Relational Database (PostgreSQL with Hibernate ORM).
   2. In-Memory Cache (Ehcache or Hazelcast).
   3. Object Storage (MinIO with JavaSDK).
5. **Supporting Services**
   1. Load Balancer (HAProxy or Nginx).
   2. Message Queue (Apache Kafka).
   3. Monitoring and Logging (ELK Stack with Log4j).

**Logical Topology**

1. **Client-Server Communication**
   1. RESTful APIs using JAX-RS (Jersey implementation).
   2. WebSocket for real-time, bi-directional data exchange using Java WebSocket API.
2. **Load Distribution**
   1. Java-based load balancing using Spring Cloud Netflix (Ribbon).
3. **Data Flow**
   1. JPA for database operations.
   2. Java Message Service (JMS) for asynchronous messaging.
   3. Event-driven architecture using Spring Events.
4. **Caching Strategy**
   1. JCache (JSR-107) implementation for standardized caching.
   2. Distributed caching with Hazelcast.
5. **Storage Architecture**
   1. JPA with Hibernate for ORM.
   2. Connection pooling with HikariCP.
   3. Java NIO for efficient I/O operations.
6. **Microservices Communication**
   1. Spring Cloud Gateway for API Gateway functionality.
   2. Service discovery using Spring Cloud Netflix (Eureka).
   3. Circuit breaking with Resilience4j.

**Subsystem Architecture**

1. **Authentication Subsystem**
   1. Spring Security for OAuth 2.0 and JWT implementation.
2. **Game Logic Subsystem**
   1. Core game logic implemented as Spring Boot microservices.
   2. State management using Java enums and the State pattern.
3. **Real-time Communication Subsystem**
   1. Java WebSocket API for live game updates.
   2. STOMP over WebSocket for structured message passing.
4. **Asset Management Subsystem**
   1. Java NIO for efficient file operations.
   2. Integration with CDN using custom Java HTTP clients.
5. **Analytics and Reporting Subsystem**
   1. Apache Spark (Java API) for big data processing.
   2. Java-based scheduling with Quartz for periodic reports.

**Scalability and Fault Tolerance**

1. **Horizontal Scaling**
   1. Containerization using Docker with Java Applications.
   2. Kubernetes for orchestration of Java Microservices.
2. **Fault Tolerance**
   1. Circuit breaker pattern implementation with Resilience4j.
   2. Distributed tracing with Spring Cloud Sleuth and Zipkin.
3. **Disaster Recovery**
   1. Java-based backup solutions (e.g., Apache Commons Net for FTP backups)
   2. Fallover clustering using Java clustering frameworks (e.g., JGroups).

The above architecture addresses the requirements for multi-platform support, real-time gameplay, and efficient data management while ensuring high availability and performance within a Java ecosystem.

## [Domain Model](#_8h2ehzxfam4o)

The UML class diagram provided represents the domain model for "Draw It or Lose It" game application.

**"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.**

Let's analyze the classes and their relationships:

**Class Descriptions and Relationships**

1. **Entity**: This is an abstract base class that holds common attributes (id and name) and behaviors (getId(), getName(), toString()) for other classes. It demonstrates the principle of abstraction by providing a common structure for derived classes.
2. **Game**, **Team**, and **Player**: These classes inherit from the Entity class, showcasing the object-oriented principle of inheritance. This allows them to reuse the common attributes and methods defined in Entity while adding their specific properties and behaviors.
3. **Game**:
   * Contains a list of Team objects, representing a one-to-many relationship between Game and Team.
   * Has methods to add teams and convert to string representation.
4. **Team**:
   * Contains a list of Player objects, representing a one-to-many relationship between Team and Player.
   * Has methods to add players and convert to string representation.
5. **Player**:
   * Represents individual players in the game.
   * Only adds a toString() method to the inherited Entity properties.
6. **GameService**:
   * Acts as the central service for managing game instances.
   * Implements the Singleton pattern, ensuring only one instance of GameService exists.
   * Manages lists of games and provides methods for adding games, retrieving games, and generating unique IDs for games, teams, and players.
7. **ProgramDriver** and **SingletonTester**:
   * These classes are likely used for testing and running the application.
   * ProgramDriver uses SingletonTester, as indicated by the "uses" relationship.

**Object-Oriented Principles Demonstrated**

1. **Inheritance**: The Game, Team, and Player classes inherit from the Entity class, promoting code reuse and establishing a clear hierarchy.
2. **Encapsulation**: The use of private attributes (indicated by "-" in the UML) and public methods (indicated by "+") demonstrates encapsulation, hiding internal details and providing controlled access to object data.
3. **Abstraction**: The Entity class serves as an abstraction, defining a common interface for its subclasses.
4. **Polymorphism**: Although not explicitly shown, the use of inheritance suggests that polymorphism could be employed, allowing Entity references to hold objects of its subclasses.
5. **Composition**: The relationships between Game and Team, and between Team and Player, demonstrate composition, where one class contains collections of other class objects.
6. **Singleton Pattern**: The GameService class implements the Singleton pattern, ensuring a single instance of the service throughout the application's lifecycle.

**Fulfillment of Software Requirements**

1. **Multiple teams in a game**: The Game class contains a list of Team objects, fulfilling the requirement for one or more teams per game.
2. **Multiple players per team**: The Team class contains a list of Player objects, allowing multiple players to be assigned to each team.
3. **Unique game and team names**: The GameService class provides methods (getGame()) that can be used to check for existing names, ensuring uniqueness.
4. **Single game instance**: The Singleton pattern implemented in GameService ensures that only one instance of the game service exists, which can manage all game instances centrally.
5. **Unique identifiers**: The GameService class includes methods for generating next IDs for games, teams, and players, ensuring unique identification for each entity.

This domain model effectively captures the core entities and relationships required for the "Draw It or Lose It" game, while employing object-oriented principles to create a flexible, maintainable, and extensible design. The use of inheritance and composition allows for easy addition of new features or entities in the future, while the Singleton pattern ensures centralized control over game instances and ID generation.

## [Evaluation](#_2o15spng8stw)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | macOS offers a Unix-based environment with robust security features and excellent stability. It supports popular web servers like Apache and Nginx, and has good compatibility with various development stacks. However, Mac servers are generally more expensive than Linux alternatives and have a smaller market share in server environments, which may result in less community support and fewer specific server management tools. | Linux is the most popular choice for web servers due to its stability, security, and cost-effectiveness. It offers excellent performance and scalability, crucial for a multiplayer game server. Linux supports a wide range of web technologies and has extensive community support. However, it may require more specialized knowledge to set up and maintain compared to Windows servers. | Windows Server provides a user-friendly interface and integrates well with other Microsoft technologies. It offers good performance and scalability, though it may be less cost-effective than Linux for web hosting. Windows servers are well-suited for .NET-based applications and have strong enterprise support. However, they may require more resources and have higher licensing costs compared to Linux alternatives. | Mobile devices typically don't host servers themselves, but optimizing server-side code for mobile clients is crucial. This includes minimizing data transfer, optimizing API responses for lower-bandwidth scenarios, and implementing efficient caching strategies. Server-side rendering might be considered to reduce the processing load on mobile devices. |
| **Client Side** | Developing for Mac clients requires consideration of Safari browser compatibility, which may have some unique rendering behaviors. Mac users typically expect high-quality, polished user interfaces. The development process may require access to Mac hardware, increasing costs. However, Mac users often have up-to-date systems, which can reduce the need for extensive backwards compatibility testing. | Linux has a smaller desktop market share, so specific client-side considerations are less critical. However, ensuring compatibility with various Linux distributions and browsers (particularly Firefox) is important. Development for Linux clients often translates well to other platforms due to its adherence to web standards. The diverse ecosystem of Linux distributions can make thorough testing more challenging. | Windows has the largest desktop market share, making it crucial to ensure compatibility with various versions of Internet Explorer and Edge browsers. Development for Windows clients often covers a wide range of user scenarios due to the diverse Windows ecosystem. Testing across different Windows versions may be necessary to ensure consistent performance and appearance. | Mobile development requires careful consideration of various screen sizes, resolutions, and device capabilities. Responsive design is crucial for ensuring a good user experience across devices. Performance optimization is particularly important due to potentially limited processing power and battery life. Testing on various devices and operating systems (iOS, Android) is necessary. Progressive Web App (PWA) technologies can be leveraged to provide a native-like experience. |
| **Development Tools** | Mac provides a rich set of development tools, including Xcode for native app development and command-line tools for web development. Popular IDEs like Visual Studio Code, IntelliJ IDEA, and WebStorm are available. The Unix-based terminal allows for easy use of web development tools and frameworks. Mac also offers excellent support for design tools like Sketch and Adobe Creative Suite, which can be beneficial for creating game assets. | Linux offers a wide array of development tools and IDEs, many of which are open-source and free. Popular choices include Visual Studio Code, Eclipse, and JetBrains IDEs. The command-line interface in Linux is particularly powerful for web development tasks. While some design tools may have limited Linux support, alternatives like GIMP and Inkscape are available for asset creation and modification. | Windows offers a comprehensive set of development tools, particularly for .NET environments. Visual Studio provides an excellent IDE for web development, and Microsoft's Visual Studio Code is a popular cross-platform option. Windows supports most major web development frameworks and tools. Adobe Creative Suite and other design software are well-supported, facilitating game asset creation. | Mobile-specific development tools include Android Studio for Android and Xcode for iOS. However, for a web-based game, cross-platform frameworks like React Native or Flutter might be considered for a more native feel. Tools for responsive design testing, such as browser developer tools with device emulation, are essential. Mobile-specific testing frameworks and emulators are crucial for ensuring compatibility across a range of devices. |

## Impact on Development Team:

## Multi-platform expertise: The team will need developers familiar with web technologies (HTML5, CSS3, JavaScript) and potentially mobile-specific frameworks.

## Testing resources: Comprehensive testing across multiple platforms and devices will require significant time and resources.

## Design considerations: UI/UX designers will need to create responsive designs that work well across various screen sizes and resolutions.

## Performance optimization: Developers will need to focus on optimizing the application for performance across different devices, especially mobile platforms.

## Continuous integration and deployment: Setting up a CI/CD pipeline that can build and test for multiple platforms will be crucial.

## Security considerations: The team will need to implement robust security measures to protect user data across all platforms.

## Scalability planning: Architects and backend developers will need to design a system that can scale to handle thousands of concurrent players.

## While a single, cross-functional team could potentially handle the development, it might be more efficient to have specialized sub-teams focusing on server-side, web client, and mobile optimization respectively. This would allow for parallel development and specialized expertise in each area.

## Licensing Considerations:

## Server-side: Linux-based solutions offer the most cost-effective licensing options, while Windows Server licenses can be more expensive, especially at scale.

## Development tools: Many high-quality development tools are available for free or with community licenses across all platforms. However, some specialized tools or enterprise-grade solutions may require paid licenses.

## Mobile development: Both Android Studio and Xcode are free, but publishing apps on the respective app stores requires a developer account, which has an associated cost (annual fee for Apple, one-time fee for Google).

## Design tools: Professional design tools like Adobe Creative Suite require subscription licenses, which should be factored into the budget.

## Frameworks and libraries: Most web and mobile development frameworks are open-source and free to use, but some specialized libraries or services may require licensing fees.

## By carefully considering these factors and choosing the right mix of technologies and platforms, The Gaming Room can create a robust, cross-platform version of "Draw It or Lose It" that meets their requirements for scalability and user experience across desktop and mobile devices.

## Recommendations

Considering the architectural considerations and specific requirements for The Gaming Room's 'Draw It or Lose It' game, I recommend using a Linux-based operating system, specifically Ubuntu Server, as the primary platform for hosting. This choice offers several benefits, including cost-effectiveness and an open-source nature that aligns with scaling needs, excellent stability, and performance for web server applications, broad community support with extensive documentation, and robust security features with regular updates. These benefits should instill confidence in the suitability of Ubuntu Server for hosting the game.

Ubuntu Server's monolithic kernel architecture provides critical features that make it ideal for this application. It offers efficient process and memory management, a modular design allowing for easy addition or removal of functionality, strong support for containerization enabling microservices architecture, and excellent compatibility with web technologies and frameworks.

For storage management, I recommend a combination of solutions. PostgreSQL should be used for structured data such as user accounts, game states, and scores. Redis can handle caching and real-time data like active game sessions and leaderboards. Object storage, such as MinIO, is ideal for game assets, including images and audio files. This multi-tiered approach provides scalability to handle increasing user loads, fast read/write operations for real-time gameplay, and efficient management of large media files.

Ubuntu server employs several beneficial memory management techniques for 'Draw It or Lose It.' These include a virtual memory system for efficient allocation, Kernel Same-page Merging to reduce usage, Transparent Huge Pages for improved performance with large memory applications, and CGroup-based memory limits for containerized applications. These features ensure optimal resource utilization, a crucial factor for a real-time, multi-user game application, and should provide reassurance about the system's capabilities.

To enable cross-platform communication, I recommend implementing a microservices architecture using Docker containers, with Kubernetes for orchestration to manage scaling and load balancing. A message queue system like RabbitMQ can handle asynchronous communication between services. At the same time, WebSocket technology can provide real-time, bi-directional communication between clients and servers. To address potential network issues, robust error handling, and reconnection strategies should be implemented, along with using a Content Delivery Network to reduce latency for static assets. The application should also be designed to degrade functionality gracefully during partial outages.

Security is paramount for 'Draw It or Lose It.' My recommended measures are robust and comprehensive, including implementing HTTPS using TLS 1.3 for all client-server communications, using OAuth 2.0 with OpenID Connect for secure user authentication, and employing JSON Web Tokens for maintaining secure sessions. Additional measures such as rate limiting to prevent DDoS attacks, regular security audits and penetration testing, using prepared statements and input validation to prevent SQL injection, implementing proper access controls and the principle of least privilege, and encrypting sensitive data at rest using AES-256 encryption, all contribute to a secure environment. Ubuntu Server provides several security features that support these measures, including AppArmor for application isolation, UFW for network security, automatic security updates, and built-in support for SELinux for granular access control.

By implementing these recommendations, The Gaming Room can develop a secure, scalable, and efficient web-based version of "Draw It or Lose It" that can quickly adapt to various computing environments while ensuring high performance and user satisfaction.