

Draw it or lose it

# **CS 230 Project Software Design**

Version 3.0

## 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 | 09/17/2023 | Pravishna Nand | Updates for initial project design |
| 2.0 | 10/1/2023 | Pravishna Nand | Update evaluation section with server side, client side and development tools information. |
| 3.0 | 10/14/2023 | Pravishna Nand | Update 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)

As we embark on the exciting transition of "Draw it or Lose It" to a web-based platform, it's crucial to be mindful of several key factors. Firstly, the game will now be accessible on diverse devices and web browsers, necessitating a strong focus on cross-platform compatibility to ensure a consistent user experience. Additionally, as user data will be involved, stringent security measures will be implemented to protect privacy and data integrity. Fair gameplay is paramount; anti-cheat mechanisms will guarantee a level playing field for all users. Responsive design will adapt the game's interface seamlessly to various screens. Lastly, ongoing maintenance and updates will be essential to keep the game fresh and responsive to evolving technology and user expectations.

This transition aligns with The Gaming Room's vision to serve all platforms, moving beyond its current Android-exclusive status. The application, known as "Draw it, or Lose It," draws inspiration from the 1980s television game show "Win, Lose, or Draw." This strategic shift to web-based gaming promises to broaden the game's reach and offer an accessible, secure, and enjoyable gaming experience to a broader audience. By keeping these considerations in mind, we are dedicated to ensuring the successful expansion of "Draw it, or Lose It" to the web, delivering a game that captures the essence of the classic while embracing modern possibilities.

## [Design Constraints](#_2et92p0)

**Cross-Platform Compatibility**:

* *Constraint*: The game application must be accessible on various platforms and web browsers.
* *Implications*: This necessitates responsive design and extensive testing to ensure a consistent user experience, which can increase development complexity and testing efforts.

**Security**:

* *Constraint*: Stringent security measures must protect user data.
* *Implications*: This requires the implementation of robust security protocols, encryption techniques, and secure data storage, adding complexity to development and ongoing maintenance.

**Fair Gameplay**:

* *Constraint*: The game must prevent cheating.
* *Implications*: Complex anti-cheat mechanisms must be developed and maintained to ensure fair gameplay.

**Real-Time Rendering**:

* *Constraint*: Drawings must be rendered steadily and complete at the 30-second mark.
* *Implications*: Efficient rendering algorithms and performance optimizations are necessary to meet this requirement.

**Time Constraints**:

* *Constraint*: The game consists of four rounds, each lasting one minute.
* *Implications*: Developers must ensure smooth transitions between rounds and adhere to specified time limits.

**Team and Player Management**:

* *Constraint*: The application should support multiple teams with multiple players.
* *Implications*: Extensive database and backend development is required for managing teams and players effectively.

**Unique Game and Team Names**:

* *Constraint*: Game and team names must be unique.
* *Implications*: A naming system must be implemented, and conflicts need to be handled during team creation.

**Single Instance in Memory**:

* *Constraint*: Only one game instance can exist in memory.
* *Implications*: The development involves creating unique identifiers for game, team, and player instances and managing memory efficiently.

## [Domain Model](#_8h2ehzxfam4o)

**ProgramDriver Class**:

* This class represents the entry point of the application and contains a single method, main(), which serves as the starting point for execution. It demonstrates the principle of encapsulation by encapsulating the main method.

**SingletonTester Class**:

* The SingletonTester class contains the testSingleton() method, responsible for testing the singleton pattern implementation. The <<uses>> relationship from ProgramDriver to SingletonTester indicates that the ProgramDriver class utilizes the SingletonTester class for testing purposes.

**Entity Class**:

* The Entity class serves as a base class with common attributes and behaviors shared by other classes in the application. It encapsulates attributes id (of type long) and name (of type String) and provides constructors and accessors for these attributes. This class demonstrates encapsulation and inheritance principles as it forms the basis for other classes.

**GameService Class**:

* GameService is a critical class that manages the game-related functionality. It has attributes such as games, nextGameId, nextPlayerId, nextTeamId, and service. The <<uses>> relationship from ProgramDriver to GameService indicates that the ProgramDriver class uses GameService to interact with game-related operations.
* The Singleton pattern is implemented in GameService through the getInstance() method to ensure that only one instance exists in memory at any given time.
* The class demonstrates the principles of encapsulation, inheritance (inheriting from Entity), and Singleton design pattern.

**Game Class**:

* The Game class represents a game instance and is connected to Team classes through a "0...\*" relationship, indicating that a game can have multiple teams.
* It has attributes like teams and methods for adding teams and converting the game information to a string.
* This class demonstrates the composition relationship and the principle of encapsulation.

**Team Class**:

* Team represents a group of players within a game. It is connected to the Player class through a "0...\*" relationship, indicating that a team can have multiple players.
* Attributes include players, and methods allow adding players and converting team information to a string.
* This class showcases the composition relationship and encapsulation.

**Player Class**:

* The Player class represents individual players within a team. It encapsulates player attributes and provides methods for converting player information to a string.

**"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** | -Reliable hosting platform for web-based software applications.  -It benefits from Unix-based architecture, offering stability and security.  -Can efficiently handle web hosting, making them suitable for the client's needs.  -Licensing costs for Mac servers tend to be higher than Linux, which could impact the client's budget. | -Open-source nature, making it cost-effective for the client.  -It offers strong security and stability, critical for web-based applications.  -Provides various server distributions, each tailored for specific use cases.  -It may require more expertise to configure and maintain compared to Windows. | -Ease of use and compatibility with Microsoft technologies.  -They provide a user-friendly environment for hosting web-based applications, which could be advantageous for developers familiar with Windows.  -Licensing costs for Windows servers can be higher than Linux, potentially impacting the client's budget. | -Mobile devices, such as smartphones and tablets, are not suitable for directly hosting web-based software applications.  -Instead, they serve as clients that access the web application hosted on server platforms.  -Mobile devices have the advantage of widespread accessibility, but development must focus on creating responsive web interfaces for these devices. |
| **Client Side** | -Involves using technologies like HTML5, CSS, and JavaScript, which are standard web development tools.  -User-friendly environment, but it may require additional testing efforts to ensure compatibility with various browsers.  -Development costs are influenced by the need to test and optimize for Mac-specific aspects. | -The development process involves standard web technologies, and Linux users often appreciate open-source software.  -Testing for compatibility across different Linux distributions and desktop environments can increase development time and costs. | -Windows desktop clients are commonly used and offer a familiar environment for users.  -Developing for Windows involves standard web technologies but may require additional testing for compatibility with various Windows versions and configurations. -Development costs can be influenced by the need to optimize for Windows-specific aspects. | -Supporting mobile devices as clients necessitates developing responsive web interfaces using HTML5, CSS, and JavaScript.  -Mobile app development (iOS and Android) requires expertise in Swift, Kotlin, or Java.  -Development for mobile devices often involves additional costs and time due to platform-specific development requirements. |
| **Development Tools** | -Development for Mac clients typically involve using text editors like Visual Studio Code or web development frameworks like React, Angular, or Vue.js.  -These tools are cost-effective and readily available, making Mac development accessible for most development teams. | -Linux development tools include text editors like Visual Studio Code, as well as a wide range of open-source development frameworks and libraries.  -Licensing costs are generally low, and Linux-based tools provide flexibility for development teams. | -Windows development often utilizes text editors like Visual Studio Code or integrated development environments (IDEs) such as Visual Studio.  -While some Windows development tools may come with licensing costs, there are free and open-source alternatives available. | -Developing for mobile devices requires platform-specific tools like XCode for iOS development and Android Studio for Android development. -These tools may have specific licensing costs, and development teams need expertise in their respective platforms, potentially requiring separate teams for iOS and Android development. |

## Recommendations

**Operating Platform**:

* For ensuring the successful expansion of Draw It or Lose It to a broader range of computing environments, I recommend adopting a web-based platform as the operating platform. A web-based platform offers unparalleled accessibility, allowing users to play the game on various devices, including desktops, laptops, tablets, and smartphones, irrespective of their operating systems. By leveraging web technologies such as HTML5, CSS, and JavaScript, the game can be made compatible with major web browsers like Chrome, Firefox, Safari, and Edge, effectively breaking down the barriers of operating system dependencies. This approach enables players on Mac, Linux, Windows, and even mobile devices to access and enjoy the game seamlessly. Furthermore, it aligns with the industry trend towards web-based gaming, making it easier to reach a broader audience while minimizing the need for platform-specific development efforts. Additionally, a web-based platform allows for easier updates and maintenance, ensuring that the game remains responsive to evolving technology and user expectations across diverse computing environments.

**Operating Systems Architectures**:

* The operating platform architecture for Draw It or Lose It is based on a client-server model, which is a fundamental structure for web-based applications. In this architecture, the client-side, representing the player's device, interacts with the server-side, where the game logic and data management reside. On the client-side, the architecture follows a thin-client approach, where most of the application's functionality is executed on the server, and the client mainly handles the user interface and rendering. This approach ensures that even devices with lower processing power, such as mobile phones, can efficiently access and play the game. The client-side relies heavily on web technologies like HTML5, CSS, and JavaScript to create a responsive and interactive user interface. The server-side architecture is designed to be robust and scalable. It involves multiple layers, including a web server responsible for handling client requests, a game server that manages game logic, a database server for storing player and game data, and various security layers to protect user information. The server-side architecture is typically based on a microservices or service-oriented architecture (SOA) to promote modularity and scalability. It ensures that the game can handle a large number of concurrent players and efficiently manage data across different platforms.

**Storage Management**:

* To effectively handle the storage requirements of Draw It or Lose It on a web-based platform, a modern and scalable database management system will be needed. Specifically, a NoSQL database system like MongoDB or Firebase Firestore could be an excellent choice. These databases are well-suited for web applications due to their flexibility, horizontal scalability, and real-time synchronization capabilities. They can efficiently store various types of data, including game records, user profiles, and game assets. Additionally, they offer cloud-based solutions, which align with the web-based nature of the platform, ensuring accessibility and data integrity across different computing environments. These NoSQL databases also provide robust security features and support for user authentication, essential for protecting user data in a multi-platform environment.

**Memory Management**:

* The recommended operating platform utilizes memory management techniques primarily through the browser's memory management capabilities. The web platform employs several memory management techniques:
  1. **Garbage Collection**: Web browsers include automatic garbage collection mechanisms that identify and reclaim memory occupied by objects no longer in use. This prevents memory leaks and inefficient memory usage within the application.
  2. **Memory Allocation**: When the Draw It or Lose It software runs in a browser, memory is allocated dynamically as needed. As users interact with the game, the browser allocates memory for variables, objects, and assets, and releases it when no longer required.
  3. **Resource Loading and Unloading**: Modern web applications use memory efficiently by loading resources (such as images, sounds, and scripts) on-demand and unloading them when they are no longer visible or needed. This technique minimizes memory consumption during gameplay.
  4. **Caching**: Browsers employ caching mechanisms to store frequently used assets in memory, reducing the need to reload them from the server. This improves response times and minimizes memory usage.
  5. **Tab Management**: In a multi-tab or multi-window environment, the browser allocates memory resources to each tab or window. It may prioritize active tabs over inactive ones and suspend or limit the memory usage of background tabs to ensure a responsive experience.
  6. **Web Workers**: For computationally intensive tasks, web workers can be utilized to offload processing to separate threads, helping manage memory more effectively and prevent the main UI thread from becoming unresponsive.
  7. **Memory Profiling**: Developers can leverage browser developer tools to profile memory usage, identify memory leaks, and optimize code to minimize memory consumption.

**Distributed Systems and Networks**:

* **Client-Server Model**: Implementing a client-server architecture is essential. The game server acts as the central hub, managing game sessions, player interactions, and data synchronization. Clients (web browsers on different platforms and mobile devices) connect to the server to participate in the game.
* **APIs and Protocols**: Standardized APIs (Application Programming Interfaces) and communication protocols like HTTP/HTTPS and WebSocket are used to facilitate communication between clients and the server. These protocols ensure data exchange and real-time updates.
* **Load Balancing**: To handle a potentially large number of concurrent players, load balancing mechanisms can distribute incoming connections across multiple servers. This ensures scalability and fault tolerance, preventing server overloads and downtime.
* **Data Storage and Replication**: Player data, game state, and other critical information are stored in a distributed database system. Data replication across multiple servers or data centers enhances data availability and resilience against failures.
* **Caching**: Caching mechanisms reduce server load by storing frequently accessed data in memory. This can include game assets, player profiles, and frequently queried database records. Caches need to be synchronized across distributed servers to maintain data consistency.
* **Content Delivery Networks (CDNs)**: CDNs are employed to deliver static assets like images and game files efficiently. By caching content on edge servers distributed worldwide, CDNs reduce latency and ensure quick content delivery to players regardless of their geographical location.
* **Redundancy and Failover**: Redundancy is crucial to minimize the impact of server or network outages. Multiple server instances and network paths are maintained to ensure uninterrupted gameplay. Automated failover mechanisms can swiftly switch to backup servers in case of failure.
* **Monitoring and Logging**: Continuous monitoring of network traffic, server performance, and player interactions is essential. Detailed logs and analytics provide insights into system health, player behavior, and potential issues.
* **Scalability Planning**: The system is designed to scale horizontally and vertically to accommodate increasing player numbers and data volume. Scalability tests and capacity planning help ensure smooth performance during peak usage.
* **Global Connectivity**: A global network infrastructure with redundancy and diverse routing paths ensures connectivity between players across different regions. This minimizes the impact of network outages or disruptions in specific areas.

**Security**:

* **Data Encryption**: All sensitive data, including user credentials, gameplay data, and personal information, should be encrypted using industry-standard encryption protocols such as TLS/SSL. This ensures that data transmitted between clients and servers is secure and cannot be intercepted by malicious actors during transmission.
* **Authentication and Authorization**: Implement robust authentication mechanisms to verify the identity of users before granting access. Utilize multi-factor authentication (MFA) where possible to add an extra layer of security. Authorization controls should restrict user access based on their roles and permissions.
* **Secure User Storage**: User data should be securely stored on the server side using secure database management systems. Hash and salt user passwords before storing them, making it difficult for attackers to compromise user accounts even if the database is breached.
* **Secure APIs**: Secure the APIs used for communication between clients and the server. Use API keys, tokens, or OAuth for authentication, and validate API requests on the server to prevent unauthorized access.
* **Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) Protection**: Implement security measures to mitigate XSS and CSRF attacks. Input validation, output encoding, and the use of anti-CSRF tokens can help prevent these common web vulnerabilities.
* **Regular Security Audits and Penetration Testing**: Conduct regular security audits and penetration testing of the application to identify vulnerabilities and weaknesses. Address any findings promptly to maintain a high level of security.
* **Security Updates**: Keep all software components, libraries, and frameworks up to date with security patches. This includes the operating system, web server, application server, and any third-party libraries.
* **Firewalls and Intrusion Detection Systems (IDS)**: Deploy firewalls to filter incoming and outgoing traffic and use IDS to detect and respond to potential threats in real-time. These security measures help protect against network-based attacks.
* **User Privacy Controls**: Allow users to control their privacy settings and the sharing of personal information. Implement clear and transparent privacy policies and ensure compliance with data protection regulations such as GDPR or CCPA.
* **Incident Response Plan**: Develop and maintain an incident response plan outlining steps to be taken in the event of a security breach. This plan should include communication strategies, data breach notifications, and legal compliance.
* **User Education**: Educate users about best security practices, such as choosing strong passwords, enabling MFA, and recognizing phishing attempts. Provide resources and guidance on how to stay safe while using the platform.
* **Monitoring and Logging**: Implement comprehensive monitoring and logging solutions to track user activity and detect suspicious behavior. Log security-related events and regularly review logs for anomalies.