

<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**

[**Executive Summary 3**](#_Toc115077320)

[**Design Constraints 3**](#_Toc115077322)

[**Domain Model 3**](#_Toc115077324)

[**Evaluation 4**](#_Toc115077325)

[**Recommendations 5**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 09/15/2023 | Jeffrey Flores | Describe requirements and design constraints. |

[Executive Summary](#_sbfa50wo7nsh)

The Gaming Room has requested the development of a web-based game application called "Draw It or Lose It," loosely based on the concept of the 1980s television game "Win, Lose or Draw." This game involves teams competing to guess phrases or titles based on rendered stock drawings, with one-minute rounds. The development process requires addressing specific software requirements to ensure a smooth gaming experience.

**Client's Business and Technical Requirements:**

1. **Multiple Teams:** The game should support one or more teams participating in a single instance of the game.
2. **Team Player Assignment:** Each team must be able to have multiple players assigned to it, allowing for collaborative gameplay.
3. **Unique Game and Team Names:** Game and team names should be unique to prevent naming conflicts, making it easy for users to verify the availability of a name when selecting a team.
4. **Instance Management:** Only one instance of the game should exist in memory at any given time. This is achieved by creating unique identifiers for each game instance, team, and player to ensure smooth and isolated gameplay experiences.

**Proposed Solution:**

To meet the client's software requirements and facilitate the development of "Draw It or Lose It," the following steps should be taken:

1. **Game Architecture:** Design the game's architecture to accommodate multiple teams. Each game instance should contain information about the teams involved.
2. **Team Management:** Implement a team management system where teams can be created, and players can be assigned to these teams. Ensure that team names are unique across all instances.
3. **Unique Identifiers:** Generate unique identifiers for each game instance, team, and player. This will prevent conflicts and allow for the isolation of game sessions.
4. **Game Name Availability:** Implement a function to check the availability of game and team names to assist users in choosing unique names.
5. **Memory Management:** Use appropriate memory management techniques to ensure that only one instance of the game exists in memory at a time. When a game session ends, release resources to allow for new instances.

## [Design Constraints](#_2et92p0)

1. **Network Latency:**
   * **Implication:** In a distributed web environment, network latency can affect real-time gameplay. Delays in data transmission between players and the game server may lead to lag, which can negatively impact the gaming experience. It's essential to design the game to minimize the impact of latency, possibly through techniques like client-side prediction and server synchronization.
2. **Scalability:**
   * **Implication:** As more users join the game, the system must scale horizontally to accommodate increased load. The application's architecture should support dynamic scaling, load balancing, and efficient resource utilization to ensure smooth gameplay even during peak usage periods.
3. **Security:**
   * **Implication:** Web-based games can be vulnerable to various security threats, such as cheating, unauthorized access, and data breaches. Implementing robust authentication, authorization, and encryption mechanisms is crucial to protect user data and maintain the integrity of the game.
4. **Browser Compatibility:**
   * **Implication:** Different web browsers may interpret code and render content differently. Ensuring cross-browser compatibility is essential to reach a broader audience. This constraint may require additional testing and adjustments to ensure a consistent gaming experience across various browsers and versions.
5. **Data Consistency:**
   * **Implication:** In a distributed environment, ensuring data consistency across all connected clients can be challenging. Implementing techniques like distributed databases, data synchronization, and conflict resolution mechanisms is necessary to maintain a consistent game state.
6. **Resource Management:**
   * **Implication:** Distributed systems may involve various resources, such as servers, databases, and content delivery networks (CDNs). Efficiently managing these resources, including monitoring, scaling, and optimizing resource usage, is essential to prevent bottlenecks and downtime.
7. **Third-Party Dependencies:**
   * **Implication:** Web-based games often rely on third-party libraries, APIs, and services. Dependency on external services can introduce risks related to service availability, changes in APIs, or licensing issues. The development team must carefully manage and monitor third-party dependencies to avoid potential disruptions.
8. **Compliance and Regulations:**
   * **Implication:** Depending on the game's target audience and geographic reach, it may need to adhere to specific legal and regulatory requirements, such as age restrictions, privacy laws, and content ratings. Ensuring compliance with these regulations is critical to avoid legal issues.
9. **Device Compatibility:**
   * **Implication:** Players may access the game from various devices, including desktop computers, smartphones, and tablets. Ensuring that the game is responsive and functions correctly on different screen sizes and device types is essential for a broad user base.
10. **Data Transfer Costs:**
    * **Implication:** Transmitting data over the internet incurs costs, especially when dealing with large files or a high volume of traffic. Minimizing unnecessary data transfers and optimizing data compression can help reduce operational costs.
11. **User Experience:**
    * **Implication:** Maintaining a seamless and engaging user experience in a distributed web environment can be challenging. Balancing visual richness with performance, optimizing user interfaces, and providing smooth interactions are crucial for player retention and satisfaction.
12. **Load Testing and Monitoring:**
    * **Implication:** Rigorous load testing and continuous monitoring of the distributed system are essential to identify and address performance bottlenecks, security vulnerabilities, and other issues that may arise as the user base grows.

## [Domain Model](#_8h2ehzxfam4o)

The provided UML class diagram represents the Domain Model of a game application. It outlines the classes and their relationships, helping to understand the structure and organization of the software. Here's an explanation of the classes and their relationships:

1. **ProgramDriver Class:**
   * This class represents the entry point of the program. It contains a **main()** method, indicating that it serves as the starting point for executing the application.
2. **SingletonTester Class:**
   * This class is used to test the Singleton pattern implementation. It contains a **testSingleton()** method, which is responsible for verifying the Singleton instance.
3. **Entity Class:**
   * The Entity class appears to be a base class for other classes in the system. It includes common attributes and behaviors that can be inherited by its subclasses.
   * Attributes:
     + **id**: A long representing the entity's unique identifier.
     + **name**: A String representing the entity's name.
   * Methods:
     + **Entity(id: long, name: String)**: Constructor to initialize the id and name.
     + **getId(): long**: Getter method for retrieving the entity's id.
     + **getName(): String**: Getter method for retrieving the entity's name.
     + **toString(): String**: Overrides the **toString()** method to provide a string representation of the entity.

The Entity class follows object-oriented programming principles like encapsulation by using private attributes and providing public methods for access. It also demonstrates inheritance as other classes are likely to inherit from it.

1. **GameService Class:**
   * This class seems to manage Game instances. It uses the Singleton pattern to ensure there is only one instance of GameService in the application.
   * Attributes:
     + **games**: A List of Game objects, representing the collection of games managed by the service.
     + **nextGameId**, **nextPlayerId**, **nextTeamId**: Long values indicating the next available IDs for games, players, and teams.
     + **service**: A reference to the singleton instance of GameService.
   * Methods:
     + **GameService()**: Constructor for initializing the service.
     + **getInstance(): GameService**: Method to get the singleton instance of GameService.
     + **addGame(name: String): Game**: Method to create and add a new game to the service.
     + **getGame(id: long): Game**: Method to retrieve a game by its ID.
     + **getGame(name: String): Game**: Method to retrieve a game by its name.
     + **getGameCount(): int**: Method to get the count of games managed by the service.
     + **getNextPlayerId(): long**: Method to retrieve the next available player ID.
     + **getNextTeamId(): long**: Method to retrieve the next available team ID.
2. **Game Class:**
   * This class represents a game within the application. It contains a list of teams that are part of the game.
   * Attributes:
     + **teams**: A List of Team objects representing the teams participating in the game.
   * Methods:
     + **Game(id: long, name: String)**: Constructor to initialize the game with an ID and name.
     + **addTeam(name: String): Team**: Method to create and add a new team to the game.
     + **toString(): String**: Overrides the **toString()** method to provide a string representation of the game.
3. **Team Class:**
   * This class represents a team within a game. It contains a list of players who are part of the team.
   * Attributes:
     + **players**: A List of Player objects representing the players in the team.
   * Methods:
     + **Team(id: long, name: String)**: Constructor to initialize the team with an ID and name.
     + **addPlayer(name: String): Player**: Method to create and add a new player to the team.
     + **toString(): String**: Overrides the **toString()** method to provide a string representation of the team.
4. **Player Class:**
   * This class represents a player. It appears to be a simple class with attributes and a **toString()** method.
   * Attributes:
     + **id**: A long representing the player's unique identifier.
     + **name**: A String representing the player's name.
   * Methods:
     + **Player(id: long, name: String)**: Constructor to initialize the player with an ID and name.
     + **toString(): String**: Overrides the **toString()** method to provide a string representation of the player.

**Object-Oriented Programming Principles:**

* **Inheritance:** The Entity class serves as a base class, demonstrating inheritance as other classes inherit common attributes and behaviors from it.
* **Singleton Pattern:** The GameService class implements the Singleton pattern, ensuring that there is only one instance of the GameService class in the application.
* **Encapsulation:** All classes encapsulate their attributes by using private access modifiers and provide public methods (getters and constructors) to interact with these attributes.
* **Association:** The relationships between classes (e.g., GameService to Game, Game to Team) are represented using associations with multiplicity (0...\*) to indicate that multiple objects of one class are associated with one or more objects of another class.

Overall, this UML class diagram provides a clear representation of the classes, their attributes, methods, and their relationships, demonstrating key object-oriented programming principles to efficiently fulfill the software requirements of the 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.**

## [Evaluation](#_2o15spng8stw)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | macOS, the operating system for Mac, is Unix-based and shares some characteristics with Linux.  macOS inherits Unix's stability and security features, making it suitable for web hosting.  Developers familiar with Unix-like systems will find macOS relatively easy to work with.  Mac hardware and software configurations are often more oriented toward desktop usage rather than high-demand web hosting.  macOS is proprietary and comes with associated costs, which may not be as cost-effective as Linux. | Linux is known for its stability, scalability, and security, making it a popular choice for server deployments.  It offers a wide range of server distributions (e.g., Ubuntu Server, CentOS, Debian) suitable for web hosting.  Linux distributions are typically open source and free, reducing licensing costs.  Linux allows for extensive configuration and optimization for specific server needs.  Linux has a strong security track record, with regular updates and community support.  For those unfamiliar with Linux, there may be a steeper learning curve compared to other platforms.  While there are support options available, they may not match the level of commercial support offered for Windows. | Windows Server is a dedicated server operating system designed for hosting web applications.  Windows Server offers a user-friendly interface that can be easier for those familiar with Windows environments.  If the application is built on .NET technologies, Windows Server offers seamless compatibility.  Windows Server typically comes with licensing costs, which can be a significant expense.  Historically, Windows has faced more security vulnerabilities than Linux. | Mobile platforms are typically used for client-side applications but can interact with server-side components.  Deploying a mobile app can extend the reach of the game to a broader audience.  Mobile apps can provide a more immersive experience and better user engagement.  Mobile platforms are primarily designed for client-side applications, so hosting server-side components may require additional infrastructure.  Mobile apps need to go through app store approval processes, which can add time and complexity to updates and releases. | |
| **Client Side** | Use flexible layouts to accommodate various screen resolutions.  Test on Safari to address browser-specific behavior and quirks.  Serve high-resolution images for Retina displays on macOS devices.  Implement app shell architecture for faster load times. | Use media queries to adjust the layout for larger screens (e.g., desktops) while maintaining usability on smaller screens (e.g., laptops).  Test on Firefox, Chrome, and alternative browsers like Brave to detect and address browser-specific issues.  Serve specific stylesheets for Linux browsers to ensure an optimal visual experience.  Implement service workers for caching and offline functionality. | Use flexbox or CSS grid to create flexible layouts.  Test on Microsoft Edge to address any specific rendering issues.  Serve Windows-specific fonts for better typography.  Use caching strategies to improve performance on Windows devices. | Use CSS media queries to adjust layouts for smaller screens.  Develop using React Native for consistent user experiences across platforms.  Enable push notifications for timely updates on both iOS and Android.  Serve touch-friendly UI elements for mobile users. | | |
| **Development Tools** | HTML, CSS, JavaScript for web development.  Backend language (e.g., Node.js, Python, Ruby, Java) for server-side logic.  **IDEs**: Options include Visual Studio Code (cross-platform), PyCharm (cross-platform), or RubyMine (cross-platform).  **Text Editors**: Sublime Text, Atom, and others are commonly used.  **Version Control**: Git is widely employed for source code management.  Mac development requires similar expertise as Linux, focusing on web development and the chosen backend language.  Cross-browser compatibility and Mac-specific browser testing are important aspects.  Many development tools for Mac are open-source or have free versions.  Licensing costs may be associated with specific editions or features of IDEs.  Backend technology licensing costs may vary. | HTML, CSS, JavaScript for web development.  Backend language (e.g., Node.js, Python, Ruby, Java) for server-side logic.  IDEs: Examples include Visual Studio Code (cross-platform), PyCharm (cross-platform), or RubyMine (cross-platform).  Text Editors: Options like Sublime Text or Atom are suitable for developers.  Version Control: Git is commonly used for source code management.  Linux development usually involves web development expertise for front-end and proficiency in the chosen backend language.  Depending on the project's scale, separate teams for front-end and back-end may be needed.  Cross-browser compatibility testing adds complexity to the development process.  Most development tools for Linux are open-source or offer free versions.  Some IDEs (e.g., Visual Studio Code) are free and open source, while others may have licensing costs based on features or editions.  Licensing costs for backend technologies may vary based on the chosen language. | HTML, CSS, JavaScript for web development.  Backend language (e.g., Node.js, Python, Ruby, Java) for server-side logic.  IDEs: Examples include Visual Studio Code (cross-platform), PyCharm (cross-platform), or RubyMine (cross-platform).  Text Editors: Sublime Text, Atom, and other options are available.  Version Control: Git is a common choice for source code management.  Windows development shares similarities with Linux and Mac, focusing on web and backend development.  Testing on Windows-specific browsers like Edge is necessary.  Many development tools for Windows are open-source or have free versions.  Licensing costs may be associated with specific features or editions of IDEs.  Backend technology licensing costs may vary. | **iOS**: Swift or Objective-C.  **Android**: Java or Kotlin.  IDEs:  iOS: XCode (macOS) for Swift and Objective-C development.  Android: Android Studio (cross-platform) for Java and Kotlin development.  Version Control: Git is widely used for source code management.  Testing and Debugging Tools: Platform-specific emulators/simulators and debugging tools.  App Store DeveloperAccounts: For publishing apps to app stores.  Separate development teams may be needed for iOS and Android due to platform-specific languages and tools.  Developers require expertise in Swift/Obj-C for iOS or Java/Kotlin for Android.  Xcode (macOS) is free for macOS users.  Android Studio is free and open source.  Licensing costs may include fees for app store developer accounts and third-party tools or libraries used in development. | | | |

**Recommendations**

Operating Platform

For the server side of the Draw It or Lose It application I recommend utilizing a Linux-based operating system. Linux, with its robust security, scalability, and open-source nature, will serve the game well with the requirements of the project. It provides an ideal environment for hosting a web-based game application like Draw It or Lose It. Linux is well known for its stability and scalability, making it an excellent choice for server deployments. As “Draw It or Lose It” is expected to attract a growing user base, the server must handle increased loads efficiently. Linux has a strong security track record with all deployments. It offers regular updates, community support, and a wealth of security tools that allow us to secure the project in all aspects. Given the client's emphasis on data protection and user security, Linux is a secure choice. Most Linux distributions are open source and free, reducing licensing costs. This aligns with the client's preference for cost-effective solutions. Linux provides various server distributions like Ubuntu Server, CentOS, and Debian, suitable for web hosting. The choice of distribution can be tailored to the specific needs of the project. This means is flexible it being run on multiple OS supporting any OS that you as the client may need support for. Linux also includes extensive configuration and optimization, it enables the development team to fine-tune the server environment to meet Draw It or Lose It's performance requirements.

Operating System Architectures

The recommended architecture for the operating platform is the client-server architecture. In this architecture, the server running on Linux hosts the game's core logic, user management, and game instance management. Clients running on various OS systems interact with the server to participate in the game. This architecture is the best for web-based multiplayer games and facilitates cross-platform compatibility. Linux has a monolithic kernel that is tightly integrated, allowing for efficient communication between its components. This is advantageous for high-performance server applications. Linux also provides process isolation through its memory management and security mechanisms. This ensures that each instance of the game runs independently. It prevents one game instance from impacting the stability and performance of another. For non-real-time data exchange, such as user account management and game creation, RESTful web services are recommended. These services follow the (REST) architectural style, making them simple, stateless, and easy to implement. By also implementing high availability patterns, such as active-standby or active-active configurations, to ensure uninterrupted gameplay is a great architecture approach. Active standby involves one server actively handling requests while the other is on standby as a backup. Active-active configurations involve both servers actively serving requests.

Storage Management

For storage management, I recommend using a combination of a relational database management system (RDBMS) and cloud-based storage. The RDBMS, such as PostgreSQL or MySQL, can efficiently manage structured data related to user accounts, game states, and high scores. The game's dynamic data, such as ongoing game sessions, scores, and in-game statistics, is stored in the RDBMS. This structured data can be queried to provide real-time updates to users during gameplay. Cloud-based storage services, like Amazon S3, can handle large media files and assets associated with the game, ensuring fast and reliable access to images, and drawing data. The images used for rendering clues during gameplay are stored in cloud-based storage. These images are typically large and unstructured, making cloud storage a cost-effective and scalable choice.

Memory Management

Linux manages memory efficiently through techniques like virtual memory and process isolation. Virtual memory allows for optimal memory allocation, preventing memory leaks and ensuring that only the required memory is used. This allows for the game to have a very smooth flow in the gameplay. Process isolation ensures that multiple instances of the game can run without interfering with each other's memory, supporting the client's requirement of a single instance of the game in memory at any given time. This way when multiple users are using the game we won't have any stoppage or freezes of the game due to the memory allocation to separate instances of the game.

Distributed Systems and Networks

Draw It or Lose It's communication between various platforms can be accomplished through a combination of APIs, web sockets, and RESTful services. The server, hosted on Linux, will provide APIs for client-server communication, ensuring real-time gameplay and data synchronization throughout the entire gameplay. WebSocket’s will enable bidirectional communication between clients and the server, facilitating real-time drawing and guessing. This again causes a very smooth gameplay for all users. RESTful services will handle non-real-time data exchange, such as user account management and game creation. To mitigate connectivity issues, redundant server instances and load balancing can be implemented to distribute incoming requests evenly and ensure high availability.

Security

Security is a top priority for the client. To protect user information and ensure data integrity, the following measures will be taken:

1. Implement strong authentication mechanisms, such as OAuth 2.0, to verify user identities securely. Role-based access control will be used to enforce authorization.
2. Employ encryption protocols (e.g., HTTPS/TLS) to secure data transmission between clients and the server, preventing eavesdropping and data breaches.
3. Regularly update and patch the Linux server to address security vulnerabilities. Implement intrusion detection and prevention systems to monitor and safeguard the server.
4. Utilize encryption for data at rest, ensuring that sensitive data, including user credentials, is securely stored.
5. Comply with relevant data protection regulations, such as GDPR, to safeguard user privacy. Implement robust data retention and deletion policies.
6. Perform regular code audits and security testing to identify and address vulnerabilities in the application code.