

# **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 | 01/24/25 | Christopher Disla | The content was restructured and reworded to provide more detailed and unique explanations for each section. The recommendations now include more specific details on each point, including the advantages and weaknesses of the chosen technologies and approaches. For example, the explanation of operating platforms and memory management has been expanded to clarify how they would work in the context of the "Draw It or Lose It" game. Additionally, security measures have been elaborated with a focus on protecting user data and preventing malicious attacks. The content is now more comprehensive and tailored to the needs of The Gaming Room’s game development and expansion plans. |
| 1.1 | 2/7/25 | Christopher Disla | Updated the document to provide a structured evaluation of operating platforms, categorizing server-side, client-side, and development tools separately for Linux, Mac, Windows, and Mobile. The revisions emphasize cost, compatibility, and development considerations to align with The Gaming Room’s requirements |
| 1.2 | 2/20/25 | Christopher Disla | Updated the document to provide clearer explanations of the operating platform, architecture, and system management techniques. Enhanced details on distributed systems, networking strategies, and security measures to ensure seamless cross-platform communication and robust data protection. |

**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 proposed design for the game application, "Draw It or Lose It," provides an efficient solution to support a web-based distributed environment where players compete in a drawing-based guessing game. The application is designed to accommodate multiple teams, each consisting of multiple players, while adhering to strict rules regarding unique identifiers for games, teams, and players. A single instance of the game must exist at any given time to ensure consistent state management and logical flow.

The core of the solution is built around well-established software design patterns, including the Singleton pattern for centralized game instance control and the Iterator pattern for streamlined traversal of game entities like teams and players. These patterns enhance code maintainability, modularity, and scalability, allowing the application to support additional features or functionality in the future. This design ensures a robust foundation for implementing the time-sensitive mechanics of gameplay, including one-minute rounds, gradual clue rendering, and time-restricted guessing phases.

By leveraging modern development practices and tools, this software design provides a solid framework for a reliable, engaging, and scalable game application. It offers a clear pathway to deployment while maintaining flexibility for future expansion or enhancement based on evolving client needs.

## Requirements

The first key business requirement is to support multiple teams within each game. These teams will consist of several players, enabling a competitive and collaborative environment where players can actively participate. The game will be structured around rounds where teams take turns guessing the drawings, with only one game instance allowed to exist in memory at any given time. This ensures that the game runs in a controlled environment without conflicts. The unique identifiers for each game, team, and player are essential for the clarity of the game’s operation, as players need to select distinct names when joining or creating a team.

In addition to the structural aspects, the game rounds must be carefully managed. Each round lasts one minute, and the drawings are displayed gradually, becoming fully visible at the 30-second mark. If a team fails to guess correctly, other teams will have the opportunity to offer a guess within a 15-second time limit. This time-based mechanic is vital to maintaining the game’s pace and excitement.

From a technical standpoint, the application must be developed using the Singleton design pattern. This will ensure that only one instance of the game exists at any time, managing the game state efficiently and preventing unnecessary resource usage. The Iterator pattern will also be applied to handle collections of teams and players, allowing the system to efficiently traverse and interact with these entities.

The game will be built for a web-based environment, requiring scalability and robustness to handle potentially high traffic and real-time interactions. As players engage in gameplay, asynchronous communication will be used to ensure smooth transitions between rounds and player actions. Real-time updates, such as countdown timers and player responses, will be essential to maintaining the game's pace and ensuring players are always in sync with the action.

Finally, the system will include strict validation processes to ensure that each game, team, and player is uniquely identifiable. This helps prevent errors during the game setup and ensures a smooth user experience by guaranteeing that no two entities share the same identifier. Data consistency and integrity will be maintained throughout the application, ensuring a seamless and reliable experience for the players.

By meeting these business and technical requirements, the proposed solution offers a well-rounded and robust platform that will not only meet but exceed the expectations of the client. The system's design is adaptable and scalable, allowing for easy future updates and improvements as user’s needs evolve.

## [Design Constraints](#_2et92p0)

The development of this application involves several constraints that shape the design process and impact implementation decisions.

The application will operate in a distributed web-based environment, requiring it to handle concurrent user interactions effectively. This introduces the challenge of managing shared resources without conflicts while ensuring scalability to support a growing number of users. To address these challenges, synchronization mechanisms will be incorporated to maintain data consistency, and the system will be designed to handle high volumes of simultaneous requests efficiently.

Another critical constraint is the use of the Singleton pattern, which ensures that only one instance of the game exists in memory at any given time. This prevents duplication of game state and promotes centralized management. Implementing this pattern within the GameService class ensures controlled access to the game instance, contributing to a more stable and predictable application behavior.

The requirement for unique names and identifiers for games, teams, and players is essential for maintaining clarity and avoiding conflicts during gameplay. This necessitates implementing efficient validation and lookup mechanisms to verify uniqueness at the time of creation. Using optimized data structures such as hash maps or indexed lists will enable quick and accurate checks.

Time-sensitive gameplay mechanics also present a design constraint. The game consists of one-minute rounds with progressively revealed clues and strict time limits for guesses. To support this functionality, the system will include a scheduling component or timer to track round durations and enforce time limits in real-time, ensuring a fair and dynamic gaming experience.

As a web-based application, the system must handle network latency and varying connection speeds among users. Backend architecture must support asynchronous communication and reliable state management to provide a smooth and responsive experience for all participants.

Finally, the application must be modular to accommodate future features such as leaderboards, additional game modes, or user interaction capabilities like chat. By following object-oriented principles and minimizing dependencies between components, the system will remain flexible and easy to extend, ensuring long-term viability and adaptability to client needs.

## [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 ProgramDriver class serves as the entry point of the application. It contains the main() method that initializes the game data, creates game instances, and tests the singleton behavior through the SingletonTester class. It uses the SingletonTester class, as indicated by the <<uses>> relationship, to verify the proper implementation of the singleton pattern in the application. This ensures that the GameService class functions as a single instance throughout the system.

The SingletonTester class is responsible for testing the singleton behavior of the GameService class. It verifies that only one instance of the GameService is used during the game's lifecycle. The SingletonTester class is connected to the GameService class via a <<uses>> relationship, emphasizing its role in testing the singleton implementation.

The Entity class is a base class containing common properties, such as id and name, shared by the Game, Team, and Player classes. These three classes inherit from the Entity class, as indicated by the open arrows pointing to Entity. This demonstrates the inheritance principle of OOP, where child classes reuse and extend the functionality of their parent class. By inheriting from Entity, the Game, Team, and Player classes avoid redundant code and maintain a consistent structure.

The GameService class is responsible for managing the overall game environment. It stores and manages the list of active games, generates unique identifiers, and provides methods for adding and retrieving games, teams, and players. The GameService class is connected to the Game class with a 0...\* relationship, signifying that one GameService instance can manage multiple Game objects. This illustrates composition, where GameService directly manages the lifecycle of Game objects.

The Game class represents a specific game instance, which holds a list of teams. The game is identified by a unique id and name, and it can add teams via the addTeam() method. The Game class is connected to the Team class through a 0...\* relationship, indicating that a single game can contain multiple teams. This relationship further exemplifies aggregation, meaning a Game is composed of Team objects, but these teams can exist independently.

The Team class represents a team within a game, which consists of a list of players. Each team has a unique id and name, and players are added through the addPlayer() method. Similar to the Game class, the Team class is connected to the Player class by a 0...\* relationship, meaning a team can have multiple players. This aggregation relationship allows flexibility in team composition, while still keeping the Team class independent from its players.

Finally, the Player class represents an individual player with an associated id and name. The Player class does not manage other entities but contributes to the team structure through the Team class.

The inheritance relationship between Game, Team, Player, and Entity allows these classes to share common properties like id and name while avoiding redundancy. The singleton pattern is employed in the GameService class to ensure that only one instance of the class exists throughout the application. The getInstance() method ensures that the same instance of the GameService is used whenever it is needed, which is crucial for managing shared resources like game data. This design pattern helps in maintaining a centralized game service without multiple instances competing for access. The relationships between Game, Team, and Player illustrate aggregation and composition principles.

Aggregation signifies that while Game objects can contain multiple Team objects, each team and player can exist independently. The composition relationship between GameService and Game implies a stronger dependency, where the service manages the entire lifecycle of the game objects. Each class in the diagram encapsulates its internal data. For instance, the Entity class stores id and name as private fields and provides public getter methods for accessing these values. This ensures that the internal state of each object is protected from direct modification, allowing controlled access and modification only through well-defined methods.

The system is designed in a modular way, with distinct classes handling specific aspects of the game. This separation of concerns improves maintainability, as each class is responsible for a clearly defined task. For example, the Game class manages the teams, the Team class manages the players, and the GameService class oversees the game lifecycle. This modular design allows easy updates or extensions to individual components without affecting the entire system.

**"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** | Mac OS is known for its stability and user-friendly interface. When considering hosting a web-based application, Mac offers a secure and efficient environment. Its strong developer ecosystem and integration with macOS development tools are beneficial, but the cost of Mac hardware can be prohibitive for server-side use. Additionally, Mac is less commonly used for hosting web applications compared to Linux, which could result in fewer resources and a steeper learning curve for troubleshooting in a production environment. MacOS Server exists but is rarely used for large-scale hosting. Primarily suited for local development or Apple ecosystem applications. Requires macOS hardware, increasing initial investment. Lacks strong enterprise-level hosting solutions compared to Linux or Windows. | Linux is a powerful, open-source operating system known for its robustness and cost-effectiveness, making it an excellent choice for hosting web-based applications. It offers high performance, flexibility, and scalability, as well as a wide range of server-side applications and tools. A significant advantage is its low resource consumption and minimal maintenance costs. However, Linux can be challenging to configure for users unfamiliar with the platform, and some proprietary software may not be fully compatible with Linux. Linux is the preferred choice for web hosting due to its stability, security, and scalability. Open-source solutions like Apache and Nginx provide efficient web server functionality. Most distributions are free, though enterprise versions (Red Hat, Ubuntu Pro) may have support fees. Developers need Linux administration skills for setup, security, and maintenance. | Windows Server offers deep integration with the Microsoft ecosystem, making it ideal for companies relying on Microsoft technologies like SQL Server or .NET. While it provides excellent support for enterprise-level applications, it is more resource-intensive than Linux, which increases hosting costs. Additionally, Windows hosting can sometimes present compatibility issues with open-source solutions, which may limit flexibility. However, Windows is widely supported and frequently used in corporate environments. Windows Server supports IIS (Internet Information Services) and integrates well with Microsoft environments. Enterprise editions range from $500–$6,000. Consumes more system resources compared to Linux, requiring higher-end infrastructure for scalability. | While mobile devices themselves do not host server-side applications, they rely on web servers for data and functionality. Therefore, the backend system must be optimized for mobile usage, ensuring low-latency responses and scalability. Cloud services, such as AWS or Google Cloud, are often utilized to provide infrastructure for mobile applications. The server must be prepared to handle different types of traffic and devices, ensuring a seamless experience for mobile users on both Wi-Fi and cellular networks. Mobile applications don’t require direct server hosting but need reliable API communication with a cloud-based back end. Efficient data transfer and caching mechanisms are necessary for smooth mobile gameplay. |
| **Client Side** | Developing for Mac requires an understanding of its unique ecosystem, such as different macOS versions and hardware configurations. Although web-based applications can be accessed through browsers, native Mac applications often use languages like Swift or Objective-C, adding complexity to the development process. The time and cost considerations for developing software for Mac can be higher than other platforms due to the premium price of Mac devices and the associated development tools. However, Mac provides an optimized environment for creating visually appealing and stable applications. The app must be tested for Safari, as it handles some web standards differently. Requires macOS devices for testing and optimization. If expanding to a macOS app, strict App Store guidelines must be considered. | Linux, as a client-side platform, offers flexibility, but its popularity is lower than that of Windows or macOS in the consumer space. Software development targeting Linux clients requires ensuring compatibility across various distributions and desktop environments, which can increase development time. The complexity of Linux systems means that developers need to address different package managers, system configurations, and user environments, adding extra overhead in testing and debugging. Fully supports modern browsers like Chrome and Firefox for web-based gameplay. Linux distributions vary in desktop environments, which may cause inconsistencies in user experience. | Windows dominates the desktop market, making it essential for client-side development. Creating applications for Windows is relatively simple due to its extensive support for various programming languages and frameworks. Developers must ensure compatibility with a wide range of Windows versions, including legacy systems, and optimize applications for various screen resolutions. While time and expertise are important considerations, Windows provides a large target audience, making it a priority for web-based applications. Works well with Chrome, Edge, and Firefox. Some users may still run Internet Explorer, requiring fallback solutions. Proprietary tools like Microsoft .NET could add licensing expenses if used for additional features. | For mobile clients, the primary challenge lies in ensuring the web-based application performs well on different screen sizes and devices. Development for mobile typically focuses on responsive design or building progressive web apps that adjust to the constraints of smaller devices. While mobile-friendly websites are a priority, cross-platform development tools like React Native or Flutter can also be used to create applications for both Android and iOS. The resources and expertise needed for mobile development are higher, as testing across various devices is essential for optimal performance. The web app must be designed for mobile interactions, ensuring smooth navigation and performance. Apple’s WebKit requirement for browsers on iOS may impact performance optimization. A Progressive Web App (PWA) approach could reduce the need for separate app store listings and development overhead. |
| **Development Tools** | Programming Languages: JavaScript, HTML5, CSS for frontend; Node.js, Python, Ruby, or Java for backend.  IDEs: Xcode for native development, Visual Studio Code, or Sublime Text for general development. Xcode is required for building and testing iOS apps. Apple hardware is mandatory for macOS and iOS development. Fewer professional-grade development tools are available compared to Windows. | Programming Languages: Python, Ruby, PHP, JavaScript for backend development; HTML5, CSS, and JavaScript for frontend.  IDEs: Visual Studio Code, IntelliJ IDEA, Eclipse, or Sublime Text. Open-source IDEs like VS Code, Eclipse, and JetBrains products make Linux a low-cost development platform. Developers must be comfortable with terminal-based operations for server management and debugging. | Programming Languages: C#, JavaScript, Python, Java for backend services; HTML5, CSS3, and JavaScript for frontend.  IDEs: Visual Studio, Visual Studio Code, or Eclipse. Supports a mix of open-source and proprietary tools, including Visual Studio, Unity, and .NET frameworks. Paid versions of Windows development tools can increase costs. Ensuring cross-browser support for Edge and older Windows versions may require additional effort. | Programming Languages: HTML5, CSS3, JavaScript for mobile web apps; Java for Android; Swift or Objective-C for iOS.  IDEs: Xcode for iOS, Android Studio for Android, Visual Studio Code for cross-platform development. React Native, Flutter, and Unity help streamline development across mobile platforms. Android Studio and Xcode must be used separately for platform-specific debugging. Apple Developer Program ($99/year) and Google Play Developer ($25 one-time) add to long-term costs. |

## 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's "Draw It or Lose It" game, Linux is the optimal server platform. Its open-source nature provides scalability and cost-effectiveness, making it ideal for expanding to various computing environments, including desktops, mobile devices, and cloud-based infrastructure. Linux's stability, flexibility, and community support ensure long-term viability, making it well-suited for a growing multiplayer gaming application.
2. **Operating Systems Architectures**: Linux operates on a monolithic kernel architecture, meaning that core functionalities such as process management, memory management, and device handling are integrated into a single, efficient framework. This design provides high performance, reducing overhead and improving execution speed, crucial for handling concurrent user interactions in real-time gaming. Additionally, Linux's modular capabilities allow for seamless updates and integration of new features as the game evolves.
3. **Storage Management**: For managing game data, user profiles, and game states, a relational database management system (RDBMS) like PostgreSQL or MySQL is recommended. These databases are optimized for structured data and ensure efficient storage, retrieval, and integrity of crucial information, such as player rankings, game statistics, and session history. Both options are ACID-compliant, ensuring data consistency, durability, and reliability, which is essential for multiplayer gaming environments.
4. **Memory Management**: Linux employs advanced memory management techniques, including virtual memory, paging, and memory protection, to enhance system performance. Virtual memory enables efficient handling of large applications by providing processes with an illusion of unlimited memory, while actual allocation remains optimized. Demand paging ensures that only necessary data is loaded into physical memory, reducing wastage and enhancing speed. These techniques help maintain fast response times, even as the number of concurrent players increases..
5. **Distributed Systems and Networks**: To ensure seamless communication between platforms, "Draw It or Lose It" should leverage a distributed system approach. RESTful APIs will handle structured requests such as retrieving player scores and updating game progress, while WebSockets will facilitate real-time communication, allowing immediate interactions like guesses and drawing updates. Load balancing techniques should be employed to distribute server traffic evenly, ensuring optimal performance under high user loads. Cloud-based solutions and microservices architectures will enhance scalability, while caching strategies will reduce latency by storing frequently accessed data in memory. These elements together ensure a resilient and responsive gaming experience, minimizing the impact of connectivity issues or outages.
6. **Security**: Protecting user information is paramount. All communications between clients and servers should be secured with SSL/TLS encryption, preventing unauthorized access to sensitive data such as usernames, passwords, and game statistics. Authentication and authorization should be managed using OAuth2 or JSON Web Tokens (JWT), ensuring secure and verified user sessions. Secure coding practices, including input sanitization, will mitigate risks from SQL injection, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF). Regular penetration testing, security audits, and the implementation of firewalls, Intrusion Detection Systems (IDS), and Intrusion Prevention Systems (IPS) will further strengthen security. Additionally, data redundancy and backup strategies will be crucial for protecting user data against loss or breaches.