

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 | 07/21/24 | Mason Williams | Make web-based app of the game application |

**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 requires a web-based version of their game application, "Draw It or Lose It," where teams compete to guess what is being drawn. This application will render images from a stock library as clues. The game will consist of four rounds, each lasting one minute, with drawings completed at the 30-second mark. If the initial team does not guess the puzzle within the allotted time, the remaining teams will have 15 seconds to offer one guess each.

## Requirements

To develop this application, the following software requirements must be met:

1. Support for multiple teams per game.
2. Multiple players assigned to each team.
3. Unique game and team names to avoid duplication.
4. Ensure only one instance of the game exists in memory at any time by creating unique identifiers for each game, team, and player.

## [Design Constraints](#_2et92p0)

<Identify the design constraints for developing the game application in a web-based distributed environment and explain the implications of the design constraints on application development.>

## [System Architecture View](#_ilbxbyevv6b6)

Developing "Draw It or Lose It" in a web-based distributed environment presents several design constraints:

1. **Scalability**: The application must handle multiple teams and players efficiently, which requires robust backend support and efficient database management.
2. **Concurrency**: Only one instance of the game should be active in memory at any time, necessitating careful management of game states and resources.
3. **Uniqueness**: Game and team names must be unique, requiring real-time checks and validations during name selection.
4. **Security**: User data, including game states and player information, must be protected from unauthorized access.
5. **Performance**: The game should render images and manage user inputs smoothly, ensuring a seamless gaming experience.

These constraints imply the need for a well-designed architecture, efficient database queries, and secure coding practices. They also necessitate a real-time data validation mechanism and a robust session management system to maintain game states.

## [Domain Model](#_8h2ehzxfam4o)

The UML class diagram for the game application represents the domain model and includes the following classes:

1. **ProgramDriver**: This class is responsible for initiating the program. It contains the main() method, serving as the entry point of the application.
2. **SingletonTester**: This class is used to test the Singleton pattern implementation in the GameService class. It contains the testSingleton() method.
3. **Entity**: A base class that holds common attributes and behaviors shared by other entities in the application. It includes:
   * id: long
   * name: String
   * Entity(id: long, name: String)
   * getId(): long
   * getName(): String
   * toString(): String
4. **GameService**: Manages the game's overall functionality. It includes:
   * games: List<Game>
   * nextGameId: long
   * nextPlayerId: long
   * nextTeamId: long
   * service: GameService
   * GameService()
   * getInstance(): GameService
   * addGame(name: String): Game
   * getGame(id: long): Game
   * getGame(name: String): Game
   * getGameCount(): int
   * getNextPlayerId(): long
   * getNextTeamId(): long
5. **Game**: Represents a game instance with attributes and methods to manage the game state. It includes:
   * teams: List<Team>
   * Game(id: long, name: String)
   * addTeam(name: String): Team
   * toString(): String
6. **Team**: Represents a team within a game. It includes:
   * players: List<Player>
   * Team(id: long, name: String)
   * addPlayer(name: String): Player
   * toString(): String
7. **Player**: Represents a player within a team. It includes:
   * Player(id: long, name: String)
   * toString(): String

**Class Relationships:**

* ProgramDriver uses SingletonTester for testing purposes.
* GameService has a one-to-many relationship with Game (represented by the "0...\*" notation).
* Game has a one-to-many relationship with Team (represented by the "0...\*" notation).
* Team has a one-to-many relationship with Player (represented by the "0...\*" notation).
* Game, Team, and Player inherit from the Entity class, indicating shared attributes and behaviors.

**Object-Oriented Programming Principles Demonstrated:**

1. **Encapsulation**: Each class encapsulates its data and provides methods to interact with that data, ensuring a clear separation of concerns.
2. **Inheritance**: The Entity class serves as a base class for other entities (Game, Team, Player), allowing shared attributes and behaviors to be inherited, promoting code reusability.
3. **Polymorphism**: Methods in the base class Entity can be overridden in derived classes to provide specific implementations, enhancing flexibility and maintainability.
4. **Abstraction**: The diagram abstracts the complexities of each entity by focusing on their interactions and relationships, simplifying the design process.

**"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.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Costly and less common for servers. Stable and secure, but not ideal for high-load web hosting compared to Linux. | Robust, secure, and cost-effective. Preferred choice for web hosting due to scalability and community support. | Widely used in enterprises but can be costly. Strong integration with Microsoft products. | Not suitable for server-side applications due to limited resources. |
| **Client Side** | Costly and requires expertise in macOS tools. Best for Apple-specific applications. | Cost-effective and flexible, but requires more technical expertise. | Widely used and supports numerous tools and languages. Higher costs but extensive support. | Platform-specific tools required. Development is time-consuming and costly. |
| **Development Tools** | Uses Xcode for macOS/iOS development. Ideal for Apple ecosystem apps. | Supports diverse languages and tools like Eclipse and VS Code. | Widely used and supports many tools and languages. Higher costs but extensive support. | Uses SDKs like Android Studio and Xcode. Involves platform-specific tools and testing. |

## 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 server-side platform, I recommend using Linux. Linux is a robust, secure, and cost-effective choice for hosting the server components of "Draw It or Lose It." It offers excellent scalability, supports a wide range of development tools, and has a strong community backing for troubleshooting and optimizations.
2. **Operating Systems Architectures**: The Linux platform architecture will leverage a microservices approach, where different components of the game (e.g., game logic, user management, and real-time communication) are separated into distinct services. Each service can be scaled independently, ensuring the system remains responsive under high loads. The architecture will use Docker containers for portability and Kubernetes for orchestration.
3. **Storage Management**: I recommend using PostgreSQL as the primary storage management system. PostgreSQL is a powerful, open-source relational database that offers advanced features like JSON support and full-text search. It’s well-suited for managing the complex relationships between games, teams, and players, and can scale horizontally to accommodate growth.
4. **Memory Management**: The recommended Linux platform will employ efficient memory management techniques such as shared memory, memory-mapped files, and caching to optimize performance. The use of in-memory data stores like Redis can be integrated to cache frequently accessed data, reducing latency and improving the user experience.
5. **Distributed Systems and Networks**: To support distributed systems and ensure seamless communication between platforms, I recommend using WebSockets for real-time, bidirectional communication between the client and server. The system will be designed to handle network dependencies by implementing redundancy and fallback mechanisms, ensuring the game remains operational even during connectivity issues or partial outages.
6. **Security**: Security will be prioritized by implementing SSL/TLS encryption for all data transmitted between clients and servers. User authentication and authorization will be handled using OAuth 2.0 and JWT tokens to ensure secure and scalable access control. Regular security audits, input validation, and secure coding practices will be integral to protecting user information and preventing vulnerabilities such as SQL injection and cross-site scripting (XSS).