

<Name-of-Software-Application>

# **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 | 03/23/2025 | Mariana Zornes | Initial draft of the design document. |

**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 has requested the development of a web-based version of their game, *Draw It or Lose It*, which is currently available only as an Android app. The new version must support multiple platforms, ensure unique game and team names, and allow only one instance of the game in memory at any time. To meet these requirements, we propose using the Singleton design pattern to manage game instances and the Iterator pattern to ensure unique names. Additionally, we will use object-oriented programming principles such as inheritance and encapsulation to create a scalable and maintainable application. This solution will allow The Gaming Room to expand their game to a broader audience while maintaining consistency and efficiency.

## Requirements

***Business Requirements:***

1. *Develop a web-based version of Draw It or Lose It that supports multiple platforms (Windows, Mac, Linux, and mobile devices).*
2. *Ensure that game and team names are unique to avoid conflicts.*
3. *Allow only one instance of the game to exist in memory at any time.*

***Technical Requirements:***

1. *Use the Singleton design pattern to manage game instances.*
2. *Use the Iterator pattern to validate unique names for games, teams, and players.*
3. *Implement a base Entity class to hold common attributes (id and name).*
4. *Ensure the application is scalable and maintainable using object-oriented programming principles.*

## [Design Constraints](#_2et92p0)

The development of the game application is constrained by the need to operate in a web-based distributed environment. This requires compatibility across multiple platforms, including Windows, Mac, Linux, and mobile devices. Additionally, the application must ensure unique names for games, teams, and players, which necessitates efficient search and validation mechanisms. The Singleton pattern will be used to enforce a single instance of the game in memory, while the Iterator pattern will facilitate the validation of unique names. These constraints impact the design by requiring robust memory management, efficient data structures, and cross-platform compatibility.

## [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 UML class diagram represents the domain model for the game application. It illustrates the relationships between classes and their attributes and methods. Below is a detailed explanation of the diagram:

**1. Entity Class**

* **Attributes**:
  + id: long: A unique identifier for the entity.
  + name: String: The name of the entity.
* **Methods**:
  + Entity(id: long, name: String): Constructor to initialize the entity with an ID and name.
  + getId(): long: Returns the entity's ID.
  + getName(): String: Returns the entity's name.
  + toString(): String: Returns a string representation of the entity.
* **Role**: The Entity class serves as the base class for Game, Team, and Player. It encapsulates common attributes and behaviors, promoting code reuse through inheritance.

**2. GameService Class**

* **Attributes**:
  + games: List<Game>: A list of active games.
  + nextGameId: long: The next unique identifier for a game.
  + nextTeamId: long: The next unique identifier for a team.
  + nextPlayerId: long: The next unique identifier for a player.
  + service: GameService: A static instance of the GameService class (used for the Singleton pattern).
* **Methods**:
  + addGame(name: String): Game: Adds a new game to the list.
  + getGame(index: int): Game: Retrieves a game by index.
  + getGame(name: String): Game: Retrieves a game by name.
  + getGameCount(): int: Returns the number of active games.
* **Role**: The GameService class manages game instances using the **Singleton pattern** to ensure only one instance exists in memory. It also uses the **Iterator pattern** to validate unique names for games, teams, and players.

**3. Game Class**

* **Attributes**:
  + teams: List<Team>: A list of teams in the game.
* **Methods**:
  + Game(id: long, name: String): Constructor to initialize the game with an ID and name.
  + addTeam(name: String): Team: Adds a new team to the game.
  + toString(): String: Returns a string representation of the game.
* **Role**: The Game class represents a game instance and manages teams. It inherits from the Entity class.

**4. Team Class**

* **Attributes**:
  + players: List<Player>: A list of players in the team.
* **Methods**:
  + Team(id: long, name: String): Constructor to initialize the team with an ID and name.
  + addPlayer(name: String): Player: Adds a new player to the team.
  + toString(): String: Returns a string representation of the team.
* **Role**: The Team class represents a team instance and manages players. It inherits from the Entity class.

**5. Player Class**

* **Methods**:
  + Player(id: long, name: String): Constructor to initialize the player with an ID and name.
  + toString(): String: Returns a string representation of the player.
* **Role**: The Player class represents a player instance. It inherits from the Entity class.

**6. ProgramDriver Class**

* **Methods**:
  + main(): The entry point of the application.
  + testSingleton(): Tests the Singleton behavior of the GameService class.
* **Role**: The ProgramDriver class is responsible for running the application and testing the Singleton pattern.

**7. SingletonTester Class**

* **Methods**:
  + testSingleton(): Tests the Singleton behavior of the GameService class.
* **Role**: The SingletonTester class is used to verify that only one instance of GameService exists in memory.

**Relationships Between Classes**

1. **Inheritance**:
   * Game, Team, and Player inherit from the Entity class, reusing its attributes (id and name) and methods.
2. **Composition**:
   * GameService contains a list of Game instances.
   * Each Game contains a list of Team instances.
   * Each Team contains a list of Player instances.
3. **Singleton Pattern**:
   * The GameService class uses the Singleton pattern to ensure only one instance exists in memory.
4. **Iterator Pattern**:
   * The GameService class uses the Iterator pattern to validate unique names for games, teams, and players.

**Object-Oriented Principles**

1. **Inheritance**:
   * The Entity class serves as the base class for Game, Team, and Player, promoting code reuse and reducing redundancy.
2. **Encapsulation**:
   * Attributes like id and name are private, and access is provided through public methods (getId(), getName()).
3. **Polymorphism**:
   * The toString() method is overridden in Game, Team, and Player to provide specific string representations.
4. **Singleton Pattern**:
   * Ensures only one instance of GameService exists, providing a global point of access.
5. **Iterator Pattern**:
   * Facilitates efficient traversal of lists (e.g., games, teams, players) to validate unique names.

**How the UML Diagram Fulfills Software Requirements**

1. **Unique Identifiers**:
   * The Entity class provides a unique id for each instance, ensuring uniqueness across games, teams, and players.
2. **Unique Names**:
   * The GameService class uses the Iterator pattern to validate unique names for games, teams, and players.
3. **Single Instance**:
   * The Singleton pattern ensures only one instance of GameService exists in memory.
4. **Scalability**:
   * The use of inheritance and composition allows the application to scale efficiently as more games, teams, and players are added.

**"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** | - macOS Server supports web hosting (Apache, Nginx). - Limited enterprise adoption compared to Linux. - Licensing costs apply for macOS Server. | - Preferred for web hosting (Apache, Nginx, cloud deployments). - Open-source, low licensing costs. - High scalability and stability. | - Supports IIS (Internet Information Services) for web hosting. - Higher licensing costs (Windows Server). - Strong enterprise integration. | - Mobile platforms (iOS/Android) are client-side only. - Cloud-based backend (Firebase, AWS) can support mobile clients. |
| **Client Side** | - Supports modern browsers (Safari, Chrome, Firefox). - Requires responsive web design for cross-platform compatibility. | - Supports all major browsers. - Requires testing for different Linux distributions. | - Supports Edge, Chrome, Firefox. - Legacy browser testing may be needed. | - Mobile browsers (Safari, Chrome) require responsive design. - Progressive Web App (PWA) or native wrappers (React Native, Flutter) may be needed. |
| **Development Tools** | - Xcode (for native macOS/iOS development). - VS Code, JetBrains IDEs. - Licensing costs for proprietary tools. | - Open-source tools (VS Code, Eclipse). - Docker, Kubernetes for deployment. - No licensing costs. | - Visual Studio (for .NET development). - Licensing costs for proprietary tools. | - Android Studio (Android), Xcode (iOS). - Cross-platform tools (React Native, Flutter). - Licensing costs for Apple Developer Program. |

## Recommendations

Analyze the characteristics of and techniques specific to various systems architectures and make a recommendation to The Gaming Room. Specifically, address the following:

**Operating Platform Recommendation**

For *Draw It or Lose It*'s cross-platform expansion, a Linux-based server environment (e.g., Ubuntu Server) stands as the optimal choice. Unlike proprietary alternatives like Windows Server or macOS Server, Linux eliminates licensing costs while offering superior scalability to handle fluctuating player loads. Its open-source nature allows deep customization of the gaming environment, and its proven stability in enterprise deployments minimizes downtime risks. The platform's native support for containerization through Docker ensures consistent performance across development, staging, and production environments, while compatibility with major cloud providers like AWS and Azure facilitates seamless future scaling.

**Operating System Architecture**

The Linux architecture delivers distinct advantages for gaming applications. Its modular kernel enables selective loading of only necessary components, reducing memory overhead. The kernel's process scheduler dynamically prioritizes game server threads to maintain consistent frame rates during peak traffic. Linux's ext4 filesystem provides journaling capabilities to prevent data corruption during unexpected outages , critical for preserving game state integrity. The operating system's native cgroups and namespaces enable efficient resource isolation between game instances, ensuring fair allocation of CPU and memory resources across concurrent player sessions. This architecture supports everything from small-scale deployments to massive multiplayer environments through horizontal scaling.

**Storage Management Solution**

A PostgreSQL relational database serves as the cornerstone for persistent game data storage. Its ACID compliance guarantees data integrity for critical operations like in-game purchases or score updates. The database's JSONB support allows flexible storage of semi-structured game data while maintaining query performance through indexing. For player assets like avatars or custom drawings, we'll implement a hybrid storage approach: metadata in PostgreSQL with actual files stored in an S3-compatible object storage system. Automated snapshot backups will run every six hours with transaction log archiving, enabling point-in-time recovery to within minutes of any failure. Storage volumes will use LUKS encryption-at-rest to protect sensitive player information.

**Memory Management Implementation**

Linux's sophisticated virtual memory system employs demand paging and transparent huge pages to optimize *Draw It or Lose It*'s memory usage. The kernel's OOM killer will automatically intervene if memory pressure becomes critical, prioritizing game server processes over background tasks. For Java-based components, we'll configure the G1 garbage collector to minimize pause times during memory reclamation. The game will implement object pooling for frequently spawned entities (like drawing tools or player avatars) to reduce allocation overhead. We'll configure swappiness parameters to balance RAM and swap usage appropriately for the workload, and use mlock() to keep critical game state data pinned in physical memory, preventing unwanted paging during intense gameplay sessions.

**Distributed Systems Architecture**

The multiplayer infrastructure will employ a microservices architecture with clear separation between matchmaking, game state, and social features. A RESTful API gateway will handle all incoming requests, routing them to appropriate backend services while enforcing rate limits and authentication. For real-time drawing synchronization, WebSocket connections will maintain persistent links between players' clients and dedicated game servers. We'll deploy Redis as a distributed cache to share session data across servers, with a pub/sub system handling live events like new player joins. The network topology will use multi-region deployment in AWS or Google Cloud, with anycast DNS directing players to the nearest available cluster. Automated health checks will trigger failover procedures within seconds of detecting server issues, while client-side prediction algorithms will mask any temporary network latency.

**Comprehensive Security Framework**

Security measures will protect the system at every layer:

* Network: TLS 1.3 with PFS for all communications, DDoS protection via Cloudflare
* Authentication: OAuth 2.0 with JWT tokens refreshed hourly, optional 2FA
* Data: AES-256 encryption for databases, client-side encryption for sensitive player data
* Servers: Hardened Linux kernel configs, daily vulnerability scans, immutable infrastructure
* Monitoring: SIEM integration for anomaly detection, mandatory audit logging
* Compliance: Regular third-party penetration testing, GDPR/CCPA readiness

The system will implement zero-trust principles, verifying each request regardless of origin. Player accounts will feature compromised credential checks against breach databases during login. For payment processing, we'll use PCI-compliant tokenization to avoid storing financial data. Regular tabletop exercises will ensure the team can effectively respond to security incidents.