

# 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 | <03/30/25> | <Hunter McGraw> | In this revision, the original classes were refactored to better utilize software design patterns and improve maintainability. An Entity base class was introduced to encapsulate common attributes and behaviors shared by Game, Team, and Player classes, reducing code redundancy. Each of these classes now extends Entity, leveraging inheritance to simplify class structures. The GameService class was explicitly implemented using the Singleton pattern to ensure only one instance exists across the application, providing a consistent management interface. Additionally, the Iterator pattern was employed within methods such as addGame(), addTeam(), and addPlayer() to effectively traverse collections and enforce uniqueness constraints for game, team, and player names. The provided testing and driver classes (ProgramDriver and SingletonTester) were also updated to correctly reference the Singleton instance, ensuring accurate testing and proper Singleton behavior. These modifications collectively enhance readability, reduce duplication, and align the application with industry-standard design practices. |

**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 software design challenge involves creating a robust game management system that ensures unique identification of games, teams, and players while effectively managing resources in a multi-user environment. To solve this, the application employs the Singleton design pattern to ensure a single, centralized instance of the GameService manages all game-related interactions. This ensures consistency across the application and prevents redundant instances, thereby optimizing memory usage and enhancing application stability. Additionally, the Iterator pattern has been implemented to maintain uniqueness constraints for names of games, teams, and players, preventing data duplication and ensuring integrity throughout the lifecycle of the application.

## Requirements

The client requires a game management application that supports multiple concurrent games, teams, and players with unique identifiers and names. The system should prevent duplicate naming for games, teams, and players to ensure data integrity. The client specifically requested implementation of common design patterns to improve efficiency, maintainability, and scalability. Moreover, the application must adhere to industry-standard best practices, including readability, clarity of inline comments, and well-structured object-oriented design, to facilitate future modifications and support ongoing maintenance.

## [Design Constraints](#_2et92p0)

Developing the game application within a web-based distributed environment introduces constraints such as maintaining statelessness where possible, ensuring thread safety, and managing concurrent user interactions efficiently. The Singleton pattern directly addresses constraints around resource optimization by restricting multiple instances of key services, which is critical in web applications to manage resource overhead effectively. However, this necessitates careful handling to avoid thread contention or potential bottlenecks. The Iterator pattern aids in traversing and validating uniqueness efficiently, reducing the complexity of operations involving multiple simultaneous data lookups or updates. These constraints require rigorous implementation practices, ensuring methods are designed with concurrency control and resource management in mind.

## [Domain Model](#_8h2ehzxfam4o)

The UML class diagram provided depicts the structural organization and relationships of classes within the game management application. At the top level, the abstract base class Entity contains common attributes such as id and name that are inherited by subclasses Game, Team, and Player. This inheritance demonstrates object-oriented programming principles, specifically inheritance and abstraction, enabling reuse of shared properties and behaviors, which significantly reduces redundancy and simplifies code management.

The GameService class uses the Singleton design pattern, represented by the private constructor and the static method getInstance(), ensuring only one instance of the class manages all game-related entities. This efficiently centralizes control and manages resources consistently throughout the application. Relationships between GameService, Game, Team, and Player classes are represented as aggregations (depicted by lines with hollow diamonds and multiplicity indicators), indicating that a Game can have multiple Team instances and each Team can have multiple Player instances. This clearly illustrates the one-to-many relationships within the application.

Classes ProgramDriver and SingletonTester illustrate interactions with the GameService, demonstrating how external classes utilize the Singleton instance. Overall, the diagram encapsulates critical object-oriented principles such as inheritance, encapsulation (private attributes), and the Singleton design pattern, effectively meeting the software's requirements for uniqueness, resource efficiency, and maintainability.

**"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 is Unix-based, providing stability, security, and ease of use, making it suitable for hosting web-based applications. However, high hardware costs and limited server-focused hardware options may pose challenges. | Linux is highly reliable, secure, scalable, and cost-effective, making it an excellent choice for web-based applications. However, it requires greater technical expertise for configuration and maintenance. | Windows Server provides user-friendly interfaces and robust enterprise support, ideal for integrated Microsoft environments. Licensing costs and less flexibility in customization compared to Linux can be limitations. | Mobile devices typically aren't suited for server-side hosting due to limited hardware resources, processing capabilities, and unstable network connections. They are best reserved as clients rather than servers. |
| **Client Side** | Developing for Mac requires considering higher costs and a niche market segment, with specialized skills needed in Swift or Objective-C. However, standard web applications using cross-platform frameworks mitigate many of these concerns. | Supporting Linux clients involves minimal cost due to open-source resources but requires additional testing across various Linux distributions. Developer expertise in ensuring compatibility with popular Linux browsers and environments is important. | Windows clients have broad user adoption, simplifying the development process and reducing cost through extensive tools and resources. Developers must consider compatibility with multiple Windows versions and browsers. | Windows clients have broad user adoption, simplifying the development process and reducing cost through extensive tools and resources. Developers must consider compatibility with multiple Windows versions and browsers. |
| **Development Tools** | macOS development commonly utilizes Xcode IDE, supporting languages like Swift, Objective-C, Java, and JavaScript. Tools like Docker and IntelliJ IDEA are also popular for cross-platform web development. | Linux development typically involves using open-source tools such as Eclipse, IntelliJ IDEA, Visual Studio Code, Docker, and programming languages like Java, Python, PHP, Ruby, and JavaScript. These resources offer flexibility and powerful customization. | Windows development commonly employs Visual Studio, .NET frameworks, SQL Server, and IIS. Additionally, cross-platform tools like Eclipse or IntelliJ IDEA are frequently used for Java or web development. | Mobile device applications often utilize Android Studio for Android (Java/Kotlin) and Xcode for iOS (Swift/Objective-C). Cross-platform frameworks like React Native, Xamarin, or Flutter provide tools that reduce complexity and development time. |

## 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 expanding Draw It or Lose It to other computing environments, a cross-platform web-based architecture using Linux servers and web technologies is the most suitable. This allows the game to be accessible on desktops, laptops, and mobile devices through a web browser, reducing dependency on device-specific applications. Additionally, leveraging container orchestration platforms like Kubernetes on Linux enables automated scaling and rolling updates, ensuring minimal downtime during deployments. You can also integrate CDN services to cache static assets at the edge, improving load times for geographically distributed users.
2. **Operating Systems Architectures:** Linux operates on a modular and monolithic kernel architecture, allowing efficient communication between system components while maintaining stability. This flexibility enables better performance tuning, efficient resource allocation, and customization, which is especially useful when scaling the application across distributed environments or using containerized deployments like Docker. Furthermore, Linux namespaces and cgroups provide built‑in isolation and resource governance, ensuring each game service instance has its own secure environment and predictable resource usage. Kernel live patching can be employed to apply critical security fixes without rebooting, maintaining high availability.
3. **Storage Management**: A cloud-based storage solution such as Amazon S3 or a network-attached storage (NAS) system integrated with Linux would be appropriate. These systems provide scalable, redundant, and highly available storage options that support version control, backup strategies, and real-time access across different platforms and services. Implementing lifecycle policies and object versioning in S3 can automate data retention and cleanup, optimizing costs over time. For relational data needs—like player leaderboards—consider pairing S3 with an RDS instance or a managed NoSQL store to balance performance and consistency.
4. **Memory Management** Linux offers dynamic memory allocation using paging, segmentation, and demand paging techniques to efficiently handle memory usage. For applications like Draw It or Lose It, Linux can isolate and prioritize memory allocation to the game service, reducing latency and improving performance even during high traffic, while managing background tasks effectively. Employing memory control groups (cgroups) ensures that runaway processes cannot exhaust system memory, and tuning swapiness or using hugepages can further reduce paging overhead for latency‑sensitive operations. Monitoring tools like vmstat and perf help you identify memory hotspots and optimize allocation strategies in production.
5. **Distributed Systems and Networks**: To ensure seamless communication between different platforms, Draw It or Lose It should use a distributed architecture built on RESTful APIs or WebSockets for real-time data exchange. Utilizing a robust network infrastructure with load balancers and fault-tolerant services (e.g., through AWS or Azure) ensures uptime and performance. Redundant services and monitoring can mitigate issues like connectivity loss or server outages. Incorporating service discovery (e.g., via Consul or Kubernetes DNS), message brokers (e.g., RabbitMQ or Kafka), and health‑check endpoints enables automated failover and self-healing, while edge gateways and API rate‑limiting protect against traffic spikes and DDoS attacks.
6. **Security**: Security can be enforced through HTTPS protocols, secure authentication methods (OAuth2, JWT), and encryption of sensitive user data at rest and in transit. Linux supports a variety of built-in security modules (e.g., SELinux, firewalls, and user permissions) to safeguard against threats. Additionally, using centralized logging and intrusion detection systems enhance security monitoring across all connected platforms. Regular patch management—automated via CI/CD pipelines—along with periodic vulnerability scanning (e.g., using OpenSCAP or Nessus) ensures that both the OS and application stacks remain hardened. Employing hardware‑backed key management services (KMS) and enforcing a zero‑trust network model further strengthens data confidentiality and integrity.