

**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 | 6/07/2025 | Alondra Paulino Santos | Initial version of the software design draft |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room is expanding its game Draw It or Lose It from a mobile-only Android app to a web-based platform accessible on multiple devices. As a consultant, I designed a Java-based software solution that manages games, teams, and players efficiently while supporting future scalability and cross-platform functionality.

The system follows object-oriented programming principles and incorporates two key design patterns. The Singleton pattern ensures that only one instance of the game management system (GameService) exists, maintaining consistency in all game data throughout the application. The Iterator pattern is used to enforce uniqueness by checking for duplicate game, team, or player names before adding new entries.

These design choices not only simplify maintenance and reduce errors but also make the system easier to extend. Whether adding new features like score tracking or transitioning to a cloud-hosted environment, the current design lays a solid foundation for long-term growth and performance in a distributed system.

## Requirements

The Gaming Room's primary goal is to expand its game, Draw It or Lose It, beyond Android to reach users on multiple platforms, including desktop and web. From a business perspective, the system must support centralized game management, allow for future scalability, and maintain consistent performance across platforms. Technically, the software needs to enforce unique names for games, teams, and players, ensure all game data is shared through a single service instance, and eventually support persistent storage for real users. The application must be designed using object-oriented programming with patterns that help maintain consistency and avoid duplication. As the game grows, it should also be easy to deploy in a distributed environment and integrate secure storage, memory, and communication systems.

## [Design Constraints](#_2et92p0)

Several design constraints were identified to meet the project goals:

Singleton Pattern: Only one instance of the GameService class can exist. This simplifies memory usage and ensures all parts of the program share the same data. However, if the application expands to support multiple users or threads, synchronization methods may be needed to prevent race conditions or inconsistent behavior. It also means developers must access the service in a controlled way.

Unique Identifiers: Each game, team, and player must have a unique ID. This makes it easier to track and reference objects throughout the system. These IDs are automatically generated, which helps avoid manual errors. However, if the system connects to a distributed database later, ID generation will need to stay consistent across all servers.

Iterator Pattern for Name Uniqueness: Before adding a new game or team, the system uses an iterator to check if the name already exists. This ensures uniqueness and avoids confusion during gameplay. The method works well for small datasets, but as the number of objects increases, the system may need a faster solution like hash-based structures or a database query.

In-Memory Data Management: The prototype stores all information in memory, which limits data to one session. Once the app supports real users, it will need persistent storage like a database. Otherwise, any game data would be lost when the program ends. This also means user access, saving, and syncing between devices cannot be supported without a backend.

Scalability and Portability: The system is being designed with future growth in mind. It should eventually support desktop, mobile, and web platforms. The use of object-oriented programming and modular classes helps make the system easier to expand or update. This design approach also helps when integrating new features like scoreboards, chat functions, or remote syncing.

## [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 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.**

The UML diagram illustrates the relationships between the seven classes in the game system:Entity is an abstract superclass with two shared fields: id and name. The classes Game, Team, and Player all inherit from Entity, demonstrating the principles of inheritance and abstraction. This reduces duplicate code and allows for centralized control of shared properties.

GameService is implemented using the Singleton pattern, meaning only one instance of this class can exist during the application’s lifecycle. It manages all games, teams, and players, ensuring data consistency across the system. This design is especially useful for centralized game management and avoids conflicts between multiple instances.

Game contains a list of Team objects. When a new team is added, the addTeam() method uses an iterator to scan the existing list and make sure no duplicate team names are used. This keeps team names unique within each game.

Team holds a list of Player objects. It works similarly to Game, using the addPlayer() method and an iterator to ensure that each player in the team has a unique name. This avoids confusion during multiplayer sessions and keeps team rosters organized.

Player is a simple class that stores the player’s name and ID. It inherits from Entity but doesn’t manage or contain any other objects.

ProgramDriver is the entry point of the application. It uses GameService.getInstance() to access the Singleton and initializes the game system with sample data.

SingletonTester checks that the same GameService instance is used consistently, confirming that the Singleton pattern works as intended. It prints the current game data to verify memory is shared.

Object-Oriented Principles Demonstrated:

Encapsulation: All class fields are private and accessed through public methods, protecting internal data.

Inheritance: Shared fields (id, name) are inherited from the Entity superclass.

Abstraction: The abstract Entity class provides a clean, reusable template for subclasses.

Design Patterns:

Singleton (GameService) ensures centralized management.

Iterator is used in addTeam() and addPlayer() to prevent name duplication.

These design choices result in a clean, maintainable structure that meets the client’s requirements and allows for future growth.

## [Evaluation](#_2o15spng8stw)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | macOS can technically host a web application using tools like Apache or Nginx, but it’s rarely chosen for production environments. Apple discontinued macOS Server, so it lacks long-term support and scalability features needed for serious deployment. While it’s great for local development and UI testing, it’s not cost-effective for server-side hosting due to expensive hardware and limited enterprise support. | Linux is the industry standard for web-based application hosting. It’s open source, lightweight, secure, and extremely scalable, ideal for supporting thousands of concurrent users. Web servers like Apache, Nginx, and Tomcat are natively supported. There are no licensing fees, which makes it budget friendly. Linux is also highly customizable and works well with modern deployment tools like Docker and cloud services, making it a strong and future-proof option for scalable web hosting. | Windows Server can handle high-volume web applications through IIS (Internet Information Services) and integrates smoothly with .NET and Microsoft’s enterprise ecosystem. It’s powerful and user-friendly for teams familiar with Windows environments. However, it does require licensing costs, and system resources can be heavier than Linux. It’s a solid option for enterprise-style deployments but not as flexible or affordable. | Mobile devices aren’t viable for hosting web-based applications. While it’s technically possible to run a local server on Android, mobile devices lack the uptime, processing power, and infrastructure needed for real-world hosting. They should strictly be used as client devices in a distributed system. For server-side operations, mobile platforms don’t meet reliability or scalability requirements. |
| **Client Side** | macOS users primarily access applications through Safari and Chrome, so developers need to ensure full compatibility with these browsers. Development time may increase slightly due to occasional Mac-specific rendering issues or system differences, but Apple’s strong developer support and compliance with web standards make this manageable. The cost is higher for testing due to Apple hardware, but macOS provides a consistent and stable client platform. | Supporting Linux clients means ensuring browser compatibility with Firefox and Chromium-based browsers. Since Linux comes in many distributions and desktop environments, UI testing needs to account for more variation. However, Linux users often have technical knowledge and are more tolerant of minor inconsistencies. Development effort is moderate, and Linux adds value by supporting open standards and being easy to integrate with testing pipelines. | Windows is the most widely used desktop OS, so optimizing the application for Edge, Chrome, and Firefox is essential. Development teams must also consider multiple versions of Windows and common screen resolution variations. Fortunately, Windows provides strong documentation, wide testing tools, and a consistent user base. Supporting Windows ensures the broadest reach for desktop users, with moderate cost and development time. | Mobile users will access the app through Safari (iOS) and Chrome (Android), so responsive web design is essential. Developers must optimize for small screens, touch input, and slower network conditions. Mobile testing adds to the workload, especially for ensuring performance and layout stability across devices. Frameworks like Bootstrap or React can reduce effort, but cross-platform testing remains a key challenge. Supporting mobile clients is critical for user engagement, but it requires extra planning and testing. |
| **Development Tools** | Mac developers typically use tools like Xcode (mainly for iOS), VS Code, IntelliJ IDEA, and Eclipse. While the tools are free or low-cost, Apple hardware is expensive, which could limit team size or testing environments. On the plus side, macOS supports Unix-like tools and makes it easier to develop for iOS and cross-platform solutions. Teams using Mac can efficiently target mobile and web from a single environment. | Linux offers unmatched flexibility for developers. Tools like Eclipse, VS Code, and JetBrains IDEs are widely supported. It also allows command-line workflows, scripting, and direct control over packages, which speeds up backend and web development. Since there are no licensing fees, Linux reduces cost while giving teams powerful control. However, some training may be needed for developers unfamiliar with Linux commands or environment setup | Windows supports a wide range of IDEs, including Visual Studio, IntelliJ, Eclipse, and NetBeans. It’s ideal for .NET or enterprise developers, and Microsoft’s ecosystem makes development smooth and well-documented. However, some tools like Visual Studio Professional may have licensing costs. The Windows GUI environment is beginner-friendly and familiar, making onboarding and training faster for most teams. | Mobile app development typically requires Android Studio (for Android) and Xcode (for iOS). For web-based mobile support, frameworks like Flutter, React Native, or PWA tools are useful. Supporting both platforms natively may require separate teams unless a cross-platform approach is taken. Tooling is mostly free but managing device testing and deployment increases time and complexity. Cross-platform frameworks help streamline this but can introduce performance trade-offs. |

## Recommendations

1. **Operating Platform**: For the server-side platform, Linux is the best choice to support Draw It or Lose Its expansion. It’s widely used in the industry for hosting scalable web applications and offers excellent support for web servers like Apache, Nginx, and Tomcat. Linux is also open-source and free to use, which keeps costs low for The Gaming Room as they grow. Because it’s lightweight and highly customizable, it can easily integrate with tools like Docker or Kubernetes for containerization and cloud deployment. This flexibility makes Linux the most reliable and scalable option for hosting the backend services and databases needed to support multiple clients across different platforms.
2. **Operating Systems Architectures**: The Linux platform follows a modular, layered architecture. At its core is the Linux kernel, which handles memory management, device drivers, and system calls. On top of that is the system library layer, which provides shared utilities, such as Glibc, for running software. Above that, the shell or GUI layer allows for user interaction, but for web servers like Draw It or Lose It, most of the system functions will run in the background through services. Linux also supports multi-user and multitasking operations, making it ideal for handling multiple client sessions at once. This layered approach enhances performance, facilitates updates, and enhances system security by isolating different layers of functionality.
3. **Storage Management**: For storage, Linux offers robust support for file systems like ext4, XFS, and Btrfs. Ext4 is a stable and widely used file system that supports large volumes, journaling (for crash recovery), and high performance. It’s a good choice for Draw It or Lose It since it can handle many high-resolution image files, player data, and logs without slowing down. If the game expands to use cloud-based storage, Linux also integrates well with Amazon S3, Google Cloud Storage, or Azure. Additionally, database solutions like MySQL or PostgreSQL can be paired with the Linux OS to manage structured game and user data efficiently.
4. **Memory Management**: Linux utilizes a combination of virtual memory, paging, and demand paging to effectively manage RAM. For example, it assigns each process its own virtual memory space and only loads parts of a program into RAM as needed. This is especially useful for Draw It or Lose It, which requires handling large image assets during short game rounds. Instead of loading all 200 images at once, the system can preload only the current round’s image into memory. Linux also includes cache management to store recently accessed files, reducing wait times between rounds. These techniques help keep memory usage efficient, avoid crashes, and maintain smooth performance across devices.
5. **Distributed Systems and Networks**: To enable Draw It or Lose It to run across various platforms (such as desktop, mobile, and web), it must operate as a distributed system. The best approach is to utilize a RESTful API hosted on a Linux server. Each client device, whether it’s a Windows PC, an iPhone, or an Android tablet—can send requests to the server to fetch or submit game data. This central server model ensures data consistency and security. For network communication, the system should use HTTPS to encrypt traffic and handle interruptions gracefully with retry logic or offline caching. Distributed monitoring tools, such as Prometheus, and failover systems can also help keep the game available even during outages or traffic spikes.
6. **Security**: Security is crucial, especially when handling user data and multiplayer sessions. Linux is known for its strong security features, including file permission controls, user authentication, and built-in firewalls like iptables. To protect data in transit, the system should use SSL/TLS encryption (HTTPS), and for data at rest, sensitive information like passwords should be hashed using secure algorithms like bcrypt. The API should implement role-based access control (RBAC) to restrict user actions based on their assigned permissions. Additional layers, such as two-factor authentication or session timeouts, can also enhance security. Linux’s support for frequent updates and patches helps ensure vulnerabilities are quickly addressed, keeping user data safe across all platforms.