

**Draw It or Lose It**

# **CS 230 Project Software Design**

Version 3.0

## Table of Contents

[**CS 230 Project Software Design** 1](#_Toc115077317)

[**Table of Contents 2**](#_Toc115077318)

[**Document Revision History 2**](#_Toc115077319)

**Executive Summary 2**

[**Requirements 2**](#_Toc115077321)

[**Design Constraints 3**](#_Toc115077322)

[**System Architecture View 3**](#_Toc115077323)

[**Domain Model 3**](#_Toc115077324)

[**Evaluation 4**](#_Toc115077325)

**Recommendations 7**

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 3.0 | 08/16/25 | Adam Gomez | Updated for Project Three: Expanded OS architecture, storage, memory, distributed systems, and security details. Incorporated feedback for completeness. |

**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)

Creative Technology Solutions (CTS) has been tasked with helping The Gaming Room expand its Android-only game *Draw It or Lose It* into a scalable, web-based application capable of serving multiple platforms. The current version of the game involves teams of players guessing stock images rendered during timed rounds. The company now wants to support web clients while ensuring that a single active game instance is maintained and that game and team names remain unique to prevent conflicts.

This document introduces a proposed solution that meets these goals using a software design approach grounded in object-oriented principles. The core functionality includes dynamic team and player creation, name uniqueness enforcement, and centralized game management via a singleton class structure. A modular design will enable future extensibility, while iterator patterns will support efficient data searches. The software design laid out here provides a strong foundation for cross-platform compatibility and supports the business objectives of The Gaming Room.

## Requirements

***Business Requirements:***

* *A game will allow one or more teams to participate.*
* *Each team will consist of multiple players.*
* *Game and team names must be unique to prevent duplicates.*
* *The game should be playable through a web browser and support multiple platforms.*
* *Only one active game instance may exist in memory at a time.*

***Technical Requirements:***

* *Implement the* ***singleton pattern*** *to enforce a single game instance in memory.*
* *Ensure* ***name uniqueness*** *by using an* ***iterator-based search*** *when adding new games and teams.*
* *Follow* ***object-oriented principles*** *(encapsulation, inheritance, and polymorphism) to structure code that is reusable and maintainable.*
* *Design the application to support deployment in a* ***web-based distributed environment****, ensuring platform independence and scalability.*

## [Design Constraints](#_2et92p0)

Several important constraints impact the design of this web-based game application:

1. **Distributed Web Environment:**  
   The application must function reliably in a distributed system accessible via various platforms and devices. This limits the use of platform-specific tools and requires web-compatible technologies.
2. **Single Game Instance (Singleton Pattern):**  
   Only one instance of the game should exist in memory to prevent duplication and synchronization errors. This constraint is enforced using the singleton design pattern in the GameService class.
3. **Unique Game and Team Names:**  
   Game and team names must be unique to prevent confusion and ensure consistency. This is implemented using name validation checks via the iterator pattern in the addGame() and addTeam() methods.
4. **Scalability and Extensibility:**  
   The application must support future enhancements like additional game modes, mobile support, or real-time multiplayer. Thus, the system should be loosely coupled and easily maintainable.
5. **Object-Oriented Design:**  
   The structure must follow object-oriented principles, using encapsulation, inheritance, and polymorphism to organize code and functionality clearly.

These constraints influence every design decision, from class structure to method implementation, ensuring the solution is robust, secure, and adaptable.

## [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 shows a clear hierarchical relationship between entities in the application. At the top level, the ProgramDriver class serves as the main entry point, calling the testSingleton() method in the SingletonTester class to verify singleton behavior. A <<uses>> dependency indicates this connection.

The core abstraction is the Entity class, which contains common fields (id, name) and related methods, such as getId(), getName(), and toString(). The Game, Team, and Player classes all inherit from this base class, promoting **code reuse** through **inheritance**.

The GameService class is implemented as a **singleton**, with a private static instance and a getInstance() method to ensure only one instance is active. It manages lists of Game objects, and tracks identifiers for games, teams, and players. This fulfills the client requirement that only one game should exist in memory.

Each Game contains multiple Team objects, and each Team holds multiple Player objects. These relationships are marked with multiplicity (0...\*) in the UML and are examples of **composition**, where larger objects manage their internal components.

The design makes use of:

* **Encapsulation:** Private fields with public accessors.
* **Inheritance:** Shared logic through the Entity base class.
* **Polymorphism:** Overridden toString() methods for better debugging and output.
* **Iterator Pattern:** Used to check for existing names in addGame() and addTeam() methods, ensuring uniqueness and preventing duplicates.

This object-oriented structure supports modularity, improves readability, and satisfies the business rules effectively.

**"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 has Unix-based architecture and can support web application servers such as Apache, Nginx, and Tomcat. It provides a stable environment for development and testing, especially for iOS-related backend services. However, it is less common for large-scale production server deployments due to higher hardware costs and limited enterprise support compared to Linux. | Linux is the most widely used platform for hosting web-based applications. It is open-source, highly customizable, secure, and optimized for performance. It supports a vast array of server tools and frameworks, and its package managers make deployment easy. Its main disadvantage may be a steeper learning curve for developers unfamiliar with command-line operations. | Windows Server offers powerful integration with .NET applications and enterprise software. It has strong support for Microsoft tools and services such as IIS, SQL Server, and Active Directory. However, it tends to have higher licensing costs and is more resource-intensive compared to Linux. It is less favored for open-source stacks like LAMP or MEAN. | Mobile devices are not typically used as web servers due to hardware limitations, battery constraints, and restricted operating system capabilities. While some mobile devices can technically serve small-scale applications (e.g., local web development tools or P2P services), they are not suitable or scalable for production-level server-side hosting. |
| **Client Side** | Developing client-side software for macOS typically requires familiarity with Apple’s frameworks (like Cocoa), Swift or Objective-C, and Xcode. Supporting macOS clients can increase costs due to the need for Apple hardware and a macOS-specific development environment. Time and expertise are also required for learning Apple’s Human Interface Guidelines and managing frequent OS updates. However, the user base is often willing to pay for quality applications, making it worth the investment. | Supporting Linux clients involves targeting various desktop environments and package formats (e.g., .deb or .rpm), which may increase complexity. Applications are often built using C++, Python, or Java and may rely on cross-platform frameworks like Qt or GTK. Cost is generally low due to the open-source nature of tools, but expertise in navigating the ecosystem and testing across distributions is necessary. | Windows has the largest desktop user base, making client support essential. Tools like Visual Studio and languages such as C#, C++, and .NET make development efficient. Costs may include software licenses for development tools, though Microsoft offers free versions like Visual Studio Community. Expertise in Windows APIs and UI design is needed, but the development time can be lower thanks to strong IDEs and documentation. | Mobile client development requires supporting both iOS and Android, often requiring separate codebases unless using a cross-platform framework like Flutter or React Native. Cost is high due to the need for different tools and testing devices. Time and expertise vary depending on whether native (Swift/Kotlin) or cross-platform development is chosen. The rapid pace of mobile OS updates also adds to maintenance demands. |
| **Development Tools** | macOS development is centered around Xcode, Apple’s official IDE, which supports Swift, Objective-C, and integration with Apple’s SDKs. Developers can also use JetBrains AppCode or cross-platform tools like Flutter. Terminal tools like Homebrew aid development. For web apps, tools like VS Code and Node.js are also common. | Linux development is versatile, supporting a wide range of tools like GCC, Python, Java, and Node.js. IDEs include Eclipse, NetBeans, VS Code, and Qt Creator. Linux’s package managers and shell scripting capabilities make it ideal for custom builds and automation. Docker is also widely used for containerized development and testing. | Windows developers typically use Visual Studio for .NET applications or VS Code for lighter projects and web apps. Languages include C#, C++, JavaScript, and Python. PowerShell and WSL (Windows Subsystem for Linux) provide extended capabilities. Game development often uses Unity or Unreal Engine on Windows due to broad support. | Mobile development tools include Android Studio (Java/Kotlin) for Android and Xcode (Swift/Obj-C) for iOS. Cross-platform frameworks like Flutter, React Native, and Xamarin allow simultaneous development. Emulators and physical device testing are key. Firebase and third-party SDKs are commonly integrated for analytics, ads, and cloud features. |

**Table 2.0**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Platform** | **Server-Side Deployment** | **Licensing Costs** | **Client-Side Compatibility** | **Development Considerations** | **Tools and Languages** | **Impact on Development Team** |
| **Linux** | Widely used for scalable server-side deployment; supports Apache, Nginx, Docker, Node.js, and other web frameworks. Highly stable and performance-optimized. | Free and open-source. No licensing fees. | Fully supports modern browsers (Firefox, Chrome) on desktop. | Requires command-line and Linux server knowledge. Excellent community support. | JavaScript, Node.js, Python, React; IDEs like VS Code, Eclipse. | Minimal cost. Streamlined with full-stack JS. A single team can handle front-end and back-end efficiently. |
| **Mac** | Can be used for local dev and lightweight server hosting (Node.js, Apache). Not typical for enterprise production environments. | Expensive hardware and macOS licensing. Not ideal for server scale. | Safari, Chrome, Firefox supported; good desktop browser compatibility. | Mostly used for iOS app dev and testing. Web support is good but server hosting is limited. | Swift (for iOS), Node.js, JavaScript, VS Code, Xcode. | Needed for iOS development. Not cost-effective for hosting. Adds testing requirements for macOS. |
| **Windows** | Supports hosting via IIS, .NET Core, or Node.js. Good enterprise features. Less common for large-scale open-source web deployments. | Licensing required (Windows Server); higher operational costs. | Compatible with all modern browsers (Edge, Chrome, Firefox). | Easy-to-use GUI tools; integrates well with .NET stack. Heavier resource usage. | C#, .NET Core, Node.js, Visual Studio, VS Code. | Great for enterprise devs. May require separate team if .NET is used extensively. |
| **Android/iOS (Mobile)** | Not used for server-side hosting. Mobile devices serve only as client-side endpoints. | iOS: $99/year for Apple Dev License. Android: Free tools. | Varies by platform. Android supports full web apps. iOS has stricter Safari/PWA limitations. | Responsive design and/or PWAs required. Testing across multiple devices necessary. | HTML/CSS/JS for PWA. Flutter, React Native, Swift, Kotlin for optional native wrappers. | Mobile UX team may be required. Frameworks like React Native or Flutter reduce code duplication. |

## Recommendations

### Operating Platform

For The Gaming Room’s expansion of Draw It or Lose It, I recommend **Linux as the primary operating platform**, specifically using a Linux-based server distribution such as **Ubuntu Server LTS**. Linux is open-source, highly customizable, cost-effective, and widely adopted in enterprise and gaming back-end environments. It provides excellent scalability, stability, and performance for distributed web applications. Its extensive developer support and compatibility with containerization technologies (e.g., Docker, Kubernetes) make it the most suitable choice for hosting and expanding the game across multiple platforms, including Windows, macOS, Android, and iOS clients.

### Operating Systems Architectures

Linux follows a **monolithic kernel architecture**, where the kernel manages core system functions such as memory, processes, inter-process communication, and device drivers. This architecture provides:

* **Performance**: Direct access between applications and hardware components.
* **Modularity**: Support for loadable kernel modules, enabling customization for gaming workloads.
* **Security**: Strong permission models and mandatory access controls (e.g., SELinux or AppArmor).  
  This architecture allows Draw It or Lose It to efficiently handle concurrent users and resource-intensive operations like session tracking and real-time updates.

### Storage Management

For storage management, I recommend a **relational database management system (RDBMS)** such as **PostgreSQL**, paired with Linux’s native file system (EXT4 or XFS). This combination provides:

* **Reliability**: Journaling file systems prevent corruption after crashes.
* **Scalability**: PostgreSQL supports large datasets and concurrent transactions.
* **Cross-Platform Access**: Ensures game state, user accounts, and assets remain consistent across devices.  
  Cloud storage integration (AWS S3, Azure Blob, or Google Cloud Storage) can further enhance distributed data availability and fault tolerance.

### Memory Management

Linux provides robust memory management techniques suitable for Draw It or Lose It:

* **Paging and Virtual Memory**: Efficient use of available RAM by mapping processes to virtual memory spaces.
* **Demand Paging**: Loads only required data into memory, reducing overhead.
* **Swapping**: Extends usable memory by using disk space as swap, useful under high load.
* **Shared Memory and IPC (Inter-Process Communication)**: Facilitates communication between distributed components such as game servers, authentication services, and database processes.  
  This ensures smooth gameplay, even as multiple users connect simultaneously from different platforms.

### Distributed Systems and Networks

To enable communication across multiple platforms, Draw It or Lose It can be deployed as a **distributed system** using a client-server model:

* **Server-Side**: Game logic, matchmaking, and data persistence hosted on Linux servers.
* **Client-Side**: Platform-specific clients (Windows, Android, iOS, macOS) communicate through secure APIs.
* **Networking**: RESTful APIs or WebSockets enable real-time data exchange. A **Content Delivery Network (CDN)** ensures fast asset delivery worldwide.  
  Dependencies include stable network connectivity, fault-tolerant load balancers, and failover clusters to prevent downtime during outages.

### Security

Security is critical for protecting user information across platforms. The recommended Linux environment provides:

* **Encryption**: TLS/SSL for all client-server communication.
* **Authentication & Authorization**: Role-based access controls and OAuth 2.0 for secure user login.
* **Data Protection**: Hashing for passwords and encryption-at-rest for sensitive data.
* **Network Security**: Firewalls and intrusion detection systems to defend against attacks.
* **Cross-Platform Security**: Secure coding practices in APIs prevent vulnerabilities like SQL injection, XSS, or CSRF.  
  This layered approach ensures that user identities, session data, and financial information remain protected.