

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 | 07/15/2025 | Destiny Katz | First iteration |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room wants to expand their Android game, Draw It or Lose It, into a web-based application that works across multiple platforms. The main design problem is that the current version of the game is limited to one platform and does not support a distributed environment with scalable, real-time multiplayer functionality. To solve this, I will use the provided Java classes to build a flexible and efficient design that allows multiple teams and players to join games, ensures each game and team name is unique, and supports smooth gameplay in a web environment. The design will meet the client’s requirements while laying the foundation for a user-friendly and accessible cross-platform experience.

## Requirements

To ensure the success of the web-based version of Draw It or Lose It, the game must meet several key business and technical requirements. The game must support multiple teams and allow several players to be assigned to each team. Both game and team names must be unique to avoid confusion and ensure a smooth user experience. The application must support a consistent structure of four rounds per game, with a countdown timer and clear gameplay rules. From a technical perspective, the system should allow only one active instance of the game at a time, maintain performance in a web-based environment, and be scalable to handle multiple users accessing it from different platforms. The final product should be user-friendly, stable, and ready for future growth.

## [Design Constraints](#_2et92p0)

Developing the game application in a web-based distributed environment introduces several design constraints that must be considered during development. First, the application must manage concurrent access from multiple users, which requires careful handling of shared data to avoid conflicts or duplication, especially when teams or games are being created. Second, only one active instance of the game should exist in memory at a time, which limits how the system handles game sessions and requires centralized control. Third, ensuring that game and team names are unique adds a constraint on how data is stored and searched, impacting how the application handles validation and user input. Additionally, because the game is web-based, it must be designed to work reliably across different devices and network conditions, which puts constraints on performance, responsiveness, and error handling. These design considerations directly impact how the application is structured and how efficiently it can scale while maintaining a smooth user experience.

## [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 outlines the core structure of the game application and demonstrates how the classes are related using object-oriented principles. At the foundation is the Entity class, which holds shared attributes like id and name, and provides basic methods such as getId(), getName(), and toString(). The Game, Team, and Player classes all inherit from Entity, using inheritance to avoid duplication and keep the code organized. The GameService class acts as the central controller and follows the Singleton design pattern, as shown by its private constructor and getInstance() method. It manages a list of Game objects and tracks unique IDs for games, players, and teams. The diagram shows that GameService can be associated with multiple Game instances, and each Game can have multiple Team objects, while each Team can contain multiple Player objects. These relationships are indicated by solid lines labeled “0…\*,” representing one-to-many associations. Encapsulation is used throughout the diagram, with class attributes marked as private (-) and accessed through public methods (+). The ProgramDriver and SingletonTester classes are utility classes for running and testing the application, and their <> relationship indicates that ProgramDriver calls SingletonTester. Overall, the diagram demonstrates strong use of inheritance, encapsulation, and modular design, all of which support efficient development, scalability, and maintainability of the game application.

**"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)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Mac is Unix-based, making it compatible with many server-side tools and languages like Node.js, Python, and Java. It’s stable and secure, which is helpful for development and testing. However, Macs are rarely used for production hosting, as Linux is more common for servers. They’re also expensive and not ideal for large-scale deployments. | Linux is the most popular choice for hosting web-based applications due to its stability, security, and scalability. It supports a wide range of server software like Apache, Nginx, and databases, and is highly customizable. Linux servers are cost-effective and widely supported in cloud environments. However, managing Linux servers requires some technical expertise, and initial setup can be complex for beginners. | Windows Server is widely used for hosting web applications, especially those relying on Microsoft technologies like ASP.NET and SQL Server. It offers strong integration with enterprise tools and a user-friendly interface. However, Windows servers can be more expensive to license and maintain and may require more resources compared to Linux. Security is good but depends heavily on proper configuration and updates. | Mobile devices themselves don’t typically host web-based applications as servers due to limited processing power, storage, and network constraints. Instead, they rely on remote servers (often cloud-based) to deliver content. However, mobile edge computing is emerging, allowing some localized processing. The main weakness is the unsuitability of mobile hardware for traditional server hosting. |
| **Client Side** | Macs provide a strong environment for client-side development, especially when targeting Apple devices, thanks to native tools like the free and powerful Xcode IDE and frameworks such as Swift and SwiftUI. This setup enables efficient development and seamless support across macOS and iOS clients. However, supporting multiple client types beyond Apple platforms requires additional time, expertise, and use of cross-platform tools, which can increase complexity and development effort. While Macs have a higher upfront hardware cost, the integrated development ecosystem and robust performance make them a solid choice for Apple-focused projects, though broader cross-platform support may require more resources and specialized skills. | Supporting multiple client types with Linux requires careful planning since desktop Linux has a smaller user base compared to Windows and macOS. Development can be cost-effective due to open-source tools, but expertise in different Linux distributions and cross-platform frameworks is needed. Time might increase when testing across diverse environments, especially for non-Linux clients. | Windows dominates the desktop market, making it a critical platform for client-side support. Developing for Windows clients is generally straightforward due to broad compatibility with software and hardware. Development costs are moderate, and time can be saved by targeting the large existing user base. However, expertise in Windows-specific frameworks (e.g., .NET) may be required for native apps, and testing across different Windows versions can add complexity. | Supporting multiple mobile client types (iOS, Android) requires careful consideration of cost, time, and expertise. Native development demands specialized knowledge of Swift/Objective-C for iOS and Kotlin/Java for Android, which can increase development time and cost. Cross-platform tools like Flutter or React Native help reduce effort by enabling a single codebase, but may involve trade-offs in performance and native feature access. |
| **Development Tools** | Mac development mainly uses Swift and Objective-C for native apps, with Xcode as the primary IDE offering tools for coding, testing, and deployment. Developers also use Terminal and package managers like Homebrew. For cross-platform or web apps, languages like JavaScript and Python with frameworks such as React Native or Electron are common, often using editors like Visual Studio Code. The Mac environment is well-equipped and optimized for building quality applications. | Linux supports many programming languages like Python, Java, C/C++, and JavaScript. Popular IDEs include Visual Studio Code, Eclipse, and JetBrains products, alongside powerful command-line tools and package managers like apt and yum. The open-source ecosystem offers extensive tools for efficient development and deployment, making Linux a flexible platform for building and running software. | Windows development often uses languages like C#, .NET, and Visual Basic, with Visual Studio as the flagship IDE offering rich features for coding, debugging, and deployment. Other tools include PowerShell for scripting and Microsoft SQL Server for databases. Cross-platform development frameworks such as Electron or React Native can also be used, supported by editors like Visual Studio Code. | Mobile app development uses Swift and Xcode for iOS, and Kotlin/Java with Android Studio for Android. Cross-platform frameworks like React Native, Flutter, and Xamarin allow building apps for both platforms from a shared codebase. Additional tools include emulators, testing suites, and deployment platforms like the Apple App Store and Google Play Console, essential for distributing apps to users. |

## 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**: I recommend using Linux as the primary operating platform to expand Draw It or Lose It across multiple computing environments. Linux is cost-effective, highly scalable, and widely supported by major cloud providers. It also works well with containerization technologies like Docker, which makes it easier to deploy consistent environments across platforms (web, desktop, mobile). Linux can support server-side logic, APIs, and databases while remaining platform-agnostic for client-side deployment.
2. **Operating Systems Architectures**: Linux uses a monolithic kernel architecture, meaning most core system functions like file system management, memory, and device drivers run in kernel space for performance efficiency. It also supports modularity, allowing components like drivers to load as needed. This architecture provides high performance and reliability, which are crucial for supporting real-time interactions in an online multiplayer game like Draw It or Lose It. Additionally, it offers excellent compatibility with various hardware and virtualization tools.
3. **Storage Management**: A recommended storage management system for Linux is the ext4 file system combined with Logical Volume Manager (LVM). ext4 is fast, reliable, and optimized for large volumes of data. LVM allows flexible partitioning and dynamic resizing of storage volumes, which is helpful as the game scales. For cloud-hosted environments, scalable object storage like Amazon S3 or Google Cloud Storage can also be integrated to store user data, images, and game assets securely and efficiently.
4. **Memory Management**: Linux uses a combination of paging, segmentation, and virtual memory for efficient memory management. It ensures optimal allocation of RAM to processes, freeing unused memory and using swap space when needed. For Draw It or Lose It, which may have multiple game sessions running concurrently, Linux can manage memory resources efficiently using features like process isolation and out-of-memory (OOM) handling to prevent system crashes and maintain performance even under high load.
5. **Distributed Systems and Networks**: To enable communication between various platforms (desktop, mobile, web), Draw It or Lose It should use a distributed architecture built with RESTful APIs or WebSockets for real-time updates. Hosting the backend in a cloud environment with load balancing and redundancy ensures high availability. Message queues like RabbitMQ or Kafka can manage communication between services. Dependencies like network latency and outages should be addressed using failover systems, health checks, and content delivery networks (CDNs) to minimize downtime and maintain smooth gameplay across all devices.
6. **Security**: To protect user information, the platform should implement end-to-end encryption using HTTPS/TLS for all data transmitted between clients and servers. User authentication can be strengthened with OAuth 2.0 and secure session handling. On the server side, Linux allows the use of firewalls (iptables/ufw), SELinux, and role-based access control (RBAC) to limit system access. Additionally, secure data storage with encryption-at-rest and regular vulnerability scanning will ensure that personal and game data remain protected against threats across platforms.