

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.3

## 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 | 09/2025 | Vernon Dodo | Completed Executive Summary, Requirements, Design Constraints and Domain Model |
| 1.2 | 10/11/2025 | Vernon Dodo | Added the Evaluation section |
| 1.3 | 10/18/2025 | Vernon Dodo | Added the analysis of the characteristics specific to various system architectures |

## [Executive Summary](#_sbfa50wo7nsh)

The client, The Gaming Room, wants to develop a web-based game, based on their current Draw It or Lose It game, and it needs to be available on multiple software platforms – it is currently available in an Android app only. A game consists of four rounds of play of one minute per round. The game works by rendering images, selected from a large library of stock drawings. The rendering takes 30 seconds, during which a team needs to guess what is being rendered. If the playing team fails the guess, an opposing team is granted 15 seconds to guess the picture being rendered. The game can consist of multiple teams with multiple players, but teams and players must have unique identities.

## Requirements

*<* Please note: While this section is not being assessed, it will support your outline of the design constraints below. *In your summary, identify each of the client’s business and technical requirements in a clear and concise manner.>*

## [Design Constraints](#_2et92p0)

The game must be able to be deployed across multiple platforms. This implies that the game would have to be developed in a programming language that is agnostic of the software operating platform on which the game will be deployed.

A game should have the ability to have one or more teams assigned to it. Therefore, a game entity must have the ability to have more than one team assigned to it.

A team must have multiple members. This necessitates that the team entity needs to have the ability to have multiple members assigned to it.

Game and team names must be unique. To satisfy this constraint, we would need an entity to keep track of games and team names. In the case of a possible duplicate assignment, the game needs to have validation features to verify that proposed names do not already exist and, if they do, to reject their addition.

Only one instance of the game can exist in memory. This can be achieved by using a singleton design pattern for the game entity.

## [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 Entity class forms the base class of the whole application. It holds all the generic attributes of the class members that are derived from it. By encapsulating the common attributes into one class, we abstract those attributes from the end user while at the same time avoiding duplication of those attributes in the derived classes.

The Game, Team, and Player classes inherit from the Entity class while extending their own functionality through the incorporation of class-specific attributes and methods. At the same time, they function through composition relationships, specifically in that a game can be composed of multiple teams, and a team can be composed of multiple players.

The GameService class, while on its own, is in a composition relationship with the Game class to the effect that it can be composed of multiple Game class instances. It implements encapsulation in that it incorporates all its functions into a single module while hiding its implementation and limiting access to its attributes through access controls.

The ProgramDriver class serves as the main entry to the application. Within it, a singleton GameService instance is created, satisfying the constraint that only one instance of GameService should reside in memory. The ProgramDriver class, through instantiating all the other class members, is responsible for creating games, adding teams to games, adding players to teams, and creating players.

This UML class diagram implements several object-oriented principles. The Entity class and its derived classes, implemented inheritance by grouping all the common attributes into the Entity class and having the Game, Team, and Player classes inherit from it. In this particular instance, the inheritance relationship also implements data abstraction in that it abstracts all the common data while only exposing the necessary data members. Furthermore, the UML diagram displays the composition relationships between the GameService, Game, Team and Player classes.

**"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** | MacOS offers the following server-based deployment methods:  **Apache Web Server**  **MAMP**  **Cloud-Based macOS Hosting**  The most suitable deployment method would be cloud-based hosting; added advantage that it offers scalability**.**  Cost wise, the pricing for such a setup starts around $1.083/hour(Intel) or $0.65(M1) with a 24-hour minimum, with AWS handling Apple’s licensingvia Dedicated Hosts.  Depending on the development stack, we might need licenses for database systems, middleware or frameworks, and MDM or deployment tools. | **Characteristics**  Open-source architecture  Modular design  Supports multi-user & multitasking  Has distribution diversity  Command-Line Centric  **Advantages**  Cost efficient because its free to use and distribute  No licensing fees, even for enterprise-grade deployments.  Performance & Stability: Lightweight distributions minimize resource usage.  **Disadvantages**  Requires a steep learning curve  Fragmentation due to too many distributions  Software Compatibility in that some proprietary software might not run on it. Also, GUI-based applications are less supported. | Windows has Window Server, which is a powerful platform for hosting web-based software applications.  **Advantages**  Microsoft Ecosystem Integration  GUI-based Management  Regular security and software updates  Integrated Development and Testing environments  Typically, Windows Server do not have service fees, making them a least-cost option for hosting web-based applications. It is important to note though that it has a high prevalence of known security vulnerabilities. | Mobile devices can host web-based applications, but they come with a unique set of characteristics that both strengthen and weaken their capabilities.  **Characteristics**  Portability  Touch interface  Limited Resources such as lower CPU, RAM, and storage.  Operation is constrained by battery life.  Dedicated ecosystem inhibits app portability  Network variability – network can fluctuate in speed and reliability because of the dependence on cellular networks.  Although there would be considerable savings in costs, these characteristics make mobile devices wholly unsuitable for the gaming application. |
| **Client Side** | **Cost Considerations** Apple Hardware Licensing and Tools  Testing Infrastructure  Developer Salaries  **Time Considerations** Platform-Specific UI/UX because macOS clients often require native design patterns, which adds development time.  Cross-Platform Support: If supporting other platforms, we would need abstraction layers or separate codebases.  Testing and Debugging: macOS usually supports several versions at the same time, meaning that testing would have to take place on all of them.  **Expertise Requirements**  Languages and Frameworks specific to macOS  System-Level Knowledge of macOS  Cross-Platform Engineering | **Cost Consideration**  Since Linux is free and has no licensing fees, costs are minimal  **Time Considerations**  The time it would take for some developers to learn how to use this operating system could be better spent on application development.  **Expertise**  The use of this operating system requires a specialized set of skills that might not always be available out on the open market.  **Suitability**  Since this operating system does not strongly support GUI-based application, and since client-side applications are usually GUI-based, this platform would not be suitable for this particular project. | **Cost**  Licensing (OS, IDEs, SQL Server), infrastructure, testing tools  **Time**  Multi-client architecture, UI/UX divergence, deployment pipelines  **Expertise**  C#, .NET stack, Windows UI frameworks, Azure, DevOps, cross-platform tooling  Supporting multiple client types on Windows is feasible and powerful, especially with the .NET ecosystem, but it demands careful orchestration of resources, timelines, and technical depth. | The cost consideration of hosting client-side applications on mobile devices is negligible because mobile devices are one of the targets of client-side applications and thus are perfectly suited for this purpose, with minimal adaptations.  Key considerations when creating client-side applications for mobile devices involve responsive design, harnessing the native features of the mobile device, such as the camera, GPS, and puh notifications, and addressing connectivity limitations. |
| **Development Tools** | **Languages**  Swift – primary language for macOS  Objective-C: legacy language for macOS; interoperable with Swift  C++  Python Java JavaScript  **IDEs**  Xcode  Visual Studio Code  AppCode  IntelliJ  Eclipse  **Development Utilities**  HomeBrew  Git  Terminal + iTerm2  Simulator & Instruments  **Cross-Platform** Frameworks  Electron  Flutter  Qt  React Native  These technical requirements necessitate a vast and sometimes divergent array of skills that might need more than one development team. Taking cost into account, we might need licenses for some of these for enterprise-wide development. | **Languages**  Python  JavaScript  Java  C/C++  PHP  Ruby  Bash  **IDEs**  Visual Studio Code  JetBrains  IntelliJ IDEA  PyCharm Eclipse  Geany/Kate/Sublime Text  **Essential Development Tools**  Build & Package Managers  Testing & Debugging Tools  Deployment & DevOps  Monitoring & Logging  The technical requirements to use this platform necessitate the use of experienced software developers with a firm knowledge base for Linux. That would limit the development team on whom they could recruit for this project. | Core Programming Languages for Windows Web Development  C#  JavaScript TypeScript  VB.NET  Python  Java  PHP  Go/Rust  **IDEs**  Visual Studio  Visual Studio Code  JetBrains Rider  IntelliJ IDEA  PyCharm  Eclipse  NetBeans  **Essential Development Tools**  Build & Package Managers (MSBuild, NuGet, npm/yarn, pip/poetry)  Testing & Debugging Tools (xUnit/NUnit/MSTest, Postman/Swagger, Fiddler/Wireshark)  Deployment & DevOps(IIS, Azure DevOps/GitHub Actions, Docker for Windows, MSIX/WIX)  Monitoring & Logging(Application Insights, Serilog/NLog) | iOS Devices  Languages:  Swift  Objective-C  IDEs & Tools  Xcode  TestFlight – for beta testing and app distribution  Instruments – for performance profiling and memory analysis  **Android Devices**  **Languages**  Kotlin – official language for Android development  Java  **IDEs & Tools**  Android Studio  Android Debug Bridge(ADB)  Firebase – for backend services like authentication, analytics, and cloud messaging  **Cross-Platform Development**  Languages & Frameworks:  Dart + Flutter  JavaScript/TypeScript  C# + .NET MAUI / Xamarin – Microsoft’s approach for shared UI and business logic.  IDEs (Visual Studio, Visual Studio Code, Expo – Toolchain for React Native) |

## 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**

When looking at an operating platform that would best fit the requirements for the Draw It or Lose it Game, I looked at operating system market share [[1]](#footnote-1), and Windows came out slightly ahead of iOS. Globally, Android is in effect the leading operating system[[2]](#footnote-2), due to its proliferation on mobile devices, so we must consider that too when selecting an operating system. What we ideally want is a server-side operating system on which we can develop the game logic, but that can also be integrated with client-side applications on other platforms. The Windows operating system is well-suited for this because it can be integrated with multi-platform development frameworks such as Flutter, React Native, Xamarin, Ionic, and Kotlin.

**Operating Systems Architectures**

The Windows operating system architecture consists of a layered design that includes the kernel, the user mode, a hardware abstraction layer which abstracts the working implementation of the hardware away from the user, subsystems that provide different functionalities, and a memory management module. Additionally, it provides a robust graphical interface and system APIs that can facilitate development across various other platforms.

**Storage Management**

Azure, since it is owned by Microsoft and comes built-in in Windows Server, would be the most obvious solution for storage management. The free edition comes with limited functionality though so we might as well look at other cloud-based storage solutions, such as Amazon S3, if we are going to pay for storage anyway. Cloud storage, in general, would ensure efficient data storage, backup, and accessibility across all platforms.

**Memory Management[[3]](#footnote-3)**

The Windows operating system has a complex memory management system that handles the allocation, tracking, and protection of memory for applications. Key components of the memory management system include:

**Virtual Memory**: a system that allows each process to operate its own address space. This is important for a game such as Draw It or Lose it, where multiple instances of the game can run at the same time.

**Memory Allocation**: where the memory manager allocates memory to processes based on their needs. In the gaming application, games may start while others may be concluded; in such a case, it would be important to free up the memory by discarded completed games and to allocate memory only to current games.

**Memory Protection**: Memory protection prevents one process from accessing the memory space of another. This is an important feature for the gaming application since it will run multi-threaded processes.

**Core Services**: The memory manager provides essential services such as memory-mapped files, copy-on-write memory, and support for large configurations, all which enhance the efficiency of memory usage and improve application performance.

**Distributed Systems and Networks**

To enable **Draw It or Lose It** to communicate seamlessly across various platforms (e.g., Windows, macOS, Android, iOS, web), we would need a well-architected **distributed system** backed by a robust **network infrastructure**. This can be accomplished as follows:

* Through the client-server model, the server will handle all the game logic while the client will implement platform specific user interfaces.
* Cross-platform communication will be established using RESTful APIs or WebSockets for real-time communication between servers and clients. Platform-agnostic protocols, such as JSON, can be used to ensure cross-platform compatibility.
* Microservices architecture
  1. Break backend into independent services (e.g., user management, drawing engine, game state, chat).
  2. Each service can scale independently and be deployed across multiple nodes.
* Network Infrastructure
  1. Use the internet as a communications network. All devices are to connect via TCP/IP. Use content distribution networks and edge servers to reduce latency for global users.
  2. Real-time Synchronization: Use WebSockets or gRPC for bi-directional low-latency communication. Implement message queues for asynchronous events.
  3. Load Balancing & Failover: Use load balancing to distribute traffic evenly across all servers. Implement redundancy and failover measures to ensure high availability.

**Security**:

The immediate security consideration would be to use the inherent security capabilities of the operating system that the gaming application would be developed on. The only problem with this is that one of the requirements of the application is that it should be made available across all platforms, so this prevents us from having such a narrow focus.

Instead, we could still use the inherent security features of Windows, while invoking platform-specific security protocols on other platforms. Since we have previously decided on cloud-based storage, that decision impacts security considerations because usernames and passwords/pass keys must be stored. To this effect, Microsoft Azure has a built-in security service that includes unparalleled security intelligence for the early identification of rapidly evolving threats. Some of the features[[4]](#footnote-4) pertinent to Draw It or Lose it, are:

* Key Vault – to safeguard cryptographic keys and other secrets used by cloud applications.
* Azure DDoS protection – to protect application servers against denial-of-service attacks
* Azure Information Protection - to protect sensitive information

On the client side, with respect to Android, it has its own built-in security features that can significantly reduce application security issues. The specific features that we are interested in though for Draw It or Lose it, would be:

* Its application framework with its robust implementation of common security functionality such as cryptography, permissions, and secure interprocess communication.
* Its user-granted permissions to restrict access to system features and data
* Its application-defined permissions to control application data on a per app basis

With regards to iOS, Apple has strict security requirements for launching applications on their app store. The iOS security architecture is designed to provide a layered defense system to protect apps and user data from unauthorized access and malicious attacks. iOS has a range of security practices[[5]](#footnote-5) that developers can use to safeguard their applications. These practices include:

* Secure Coding Practices through leveraging native iOS security APIs, such as KeyChain, for secure data storage, and Secure Enclave for cryptographic operations.
* Data Protection and Encryption using native iOS frameworks like CommonCrypto and CryptoKit for strong encryption.
* Using secure network protocols such as HTTPS with TLS for data in transit, to prevent interception or tampering.

In general, if we implement security protocols on both the server and client side, that could go a long way to satisfy the security requirements of the application.

1. *Operating System Market Share United States of America | StatCounter Global Stats*. (n.d.). StatCounter Global Stats. https://gs.statcounter.com/os-market-share/all/united-states-of-america [↑](#footnote-ref-1)
2. *Operating system market share worldwide | StatCounter Global Stats*. (n.d.). StatCounter Global Stats. https://gs.statcounter.com/os-market-share [↑](#footnote-ref-2)
3. GeeksforGeeks. (2025, July 23). *Windows Memory Management*. GeeksforGeeks. https://www.geeksforgeeks.org/operating-systems/windows-memory-managment/ [↑](#footnote-ref-3)
4. *Cloud Security | Microsoft Azure*. (n.d.). https://azure.microsoft.com/en-us/products/category/security/?msockid=30bc7b38e7ec66f139566d64e6ad6780 [↑](#footnote-ref-4)
5. *Guide to iOS app security Best Practices | Blog | Digital.ai*. (n.d.). Digital.ai. https://digital.ai/catalyst-blog/ios-app-security/ [↑](#footnote-ref-5)