

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  1.1 | 05/18/2025  06/01/2025 | Ray Lyndes | Week 3 Project 1  Weel 5 Revisions to Evaluation Table |

**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 Gaming Room, a client of Creative Technology Solutions (CTS), has requested a web-based version of their Android-only game, *Draw It or Lose It*, to expand accessibility across multiple platforms. The game concept is loosely based on the television game show *Win, Lose or Draw*, in which teams guess what is being drawn based on clues. However, the digital version uses stock illustrations instead of live drawings and follows a four-round format with time constraints.

To meet the client's needs, we propose a scalable, object-oriented Java application prototype that serves as the foundation for future cross-platform web deployment. This solution is designed using industry-standard software engineering principles, such as inheritance, encapsulation, and design patterns like the singleton and iterator patterns. These will ensure a flexible, maintainable codebase.

The software design addresses all current client requirements: supporting multiple teams, assigning players to teams, ensuring uniqueness of team and game names, and limiting the game instance to a single one in memory. This solution lays the groundwork for future integration with web technologies and supports a smooth transition into a distributed web-based environment.

## Requirements

To support the transition of *Draw It or Lose It* from an Android-only platform to a web-based environment, The Gaming Room has defined several key requirements that shape the structure and behavior of the application.

From a business standpoint, the game must allow for multiple teams, each composed of several players, while ensuring that every team and game has a unique name. This helps prevent confusion and supports the multiplayer nature of the game. Players are to be grouped into teams within a single game instance, following a four-round timed format consistent with the original game. Accessibility is also a priority—the game must be playable from any modern web browser to reach a wider audience.

On the technical side, the system must be designed to support only one active game instance at a time. This will be achieved through the use of a singleton design pattern. Unique identifiers must be enforced across games, teams, and players, and object-oriented principles such as encapsulation and inheritance are to be applied to ensure a maintainable and scalable codebase. Additionally, the solution should demonstrate the use of design patterns such as iterator and singleton, providing controlled access to game data and reinforcing the single-instance requirement.

Although this prototype will be implemented in Java, the structure should anticipate integration into a broader web-based system. Scalability is essential, and the design should be flexible enough to support multiple concurrent games in future iterations.

## [Design Constraints](#_2et92p0)

Several key constraints must be considered when developing *Draw It or Lose It* as a web-based distributed application:

1. **Single Instance Constraint**: Only one instance of the game should exist in memory at any time. This limits the creation of multiple games concurrently, which impacts scalability in a distributed system. The singleton pattern is used to enforce this constraint.
2. **Uniqueness of Identifiers and Names**: Each game, team, and player must have a unique name and identifier. This requires a centralized way to track and compare existing instances, which can be challenging in a distributed environment without a synchronized data store or service.
3. **Web Compatibility**: The original version was built for Android. Porting to a web-based format means rethinking certain mobile-specific behaviors and considering cross-browser compatibility, session management, and persistent data storage.
4. **Performance and Responsiveness**: As a real-time game with timed rounds and visual rendering, performance is critical. This means selecting frameworks and technologies that ensure low latency and fast rendering times in a browser environment.
5. **Future Scalability**: While this prototype supports one game instance, a future production version will need to support multiple concurrent games and users. The current architecture must be designed with extensibility in mind.

By addressing these constraints early in the design phase, we ensure a more adaptable and stable foundation for future development and deployment.

## [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 diagram represents a clear object-oriented structure that models the core components of the *Draw It or Lose It* game. At the top of the hierarchy is the Entity class, which acts as a base class for all game-related entities. This class includes two common attributes: id and name. The following subclasses inherit from Entity:

* Game: Represents a single game instance and contains a list of Team objects.
* Team: Represents a team and contains a list of Player objects.
* Player: Represents an individual participant in a team.

The GameService class acts as a controller or manager class. It enforces the singleton pattern to ensure only one game instance exists in memory. It also contains methods for adding games, teams, and players.

The iterator pattern is used in the getGame(), getTeam(), and getPlayer() methods to check existing names for uniqueness before adding a new one. This approach encapsulates the traversal of collections while maintaining clean separation of concerns.

This structure uses the following object-oriented principles:

* **Inheritance**: All game entities share common properties and behaviors via the Entity superclass.
* **Encapsulation**: Each class manages its own state through private fields and public accessors/mutators.
* **Polymorphism**: While not fully used yet, the design allows for method overrides and dynamic behavior if needed in future extensions.
* **Abstraction**: The system separates the responsibilities of entities (data) and services (logic), promoting a modular architecture.

Together, this domain model supports the current requirements and provides a strong foundation for future enhancements, including web integration and distributed processing.

**"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 and stable, making it capable of hosting development servers, but it’s not widely used in production. Its closed-source nature limits customization. Hosting web apps on macOS is expensive and not ideal for scaling. | Linux is the industry standard for server-side hosting. It offers high stability, performance, and scalability. Its open-source nature allows customization and automation via shell scripts. However, it may require more technical expertise to set up. | Windows servers support .NET-based applications and have strong integration with Microsoft tools. However, they come with licensing costs and can be less efficient than Linux for high-performance or scalable web apps. | Mobile devices are not designed for hosting server-side applications. Limitations in performance, battery life, and networking make them unsuitable for this use case. Servers should communicate with mobile clients instead. |
| **Client Side** | Supporting macOS clients requires ensuring compatibility with Safari and native macOS applications. Development can be costly due to Apple’s hardware requirements and proprietary tools, but the user base is stable and often high-value. | Linux desktop clients are less common, so supporting them may not be a priority. However, it’s useful for developers and testers. Time and cost are minimal if apps are web-based. Expertise in handling various Linux distros is needed. | Windows is the most widely used desktop OS. Ensuring compatibility is essential. Development is cost-effective with tools like Visual Studio, and expertise is readily available. However, supporting legacy systems can increase complexity. | Supporting mobile clients requires building native or cross-platform apps (iOS, Android). Development can be costly and time-consuming but is necessary due to widespread mobile use. Testing across devices is a major factor. |
| **Development Tools** | macOS supports tools like Xcode, Swift, Objective-C, and also cross-platform IDEs like VS Code. It is required for iOS development. Homebrew is helpful for installing dev packages. | Linux offers powerful tools like GCC, Python, Node.js, Docker, and Eclipse. Most programming languages and dev environments are well-supported. Terminal-based tools are efficient but have a steep learning curve. | Windows supports .NET, C#, Visual Studio, and Windows Subsystem for Linux (WSL) for Unix-based tools. It is developer-friendly and integrates well with Azure. Some open-source tools can be harder to install. | Mobile development uses Android Studio (Java/Kotlin), Xcode (Swift), and cross-platform tools like Flutter, React Native, and Unity. Each platform has its own SDK and emulator. Build times can be slow and tools resource-intensive. |

## 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**: Use **Linux** for server-side deployment and a combination of **Windows, macOS, and mobile platforms (iOS/Android)** for client-side support. Linux offers scalability and cost-efficiency, while client support ensures maximum user reach.
2. **Operating Systems Architectures**: Linux server architecture is modular and based on the monolithic kernel with support for containers and microservices. Client systems should follow standard OS architectures: Windows with Win32/64 APIs, macOS with Unix-based architecture, and mobile with Android (Linux kernel) or iOS (Darwin/XNU kernel).
3. **Storage Management**: Use a cloud-based storage solution such as **Amazon S3, Azure Blob Storage,** or **Google Cloud Storage** for scalable object storage. Linux systems can use **ext4** or **XFS** file systems for high-performance disk I/O. Mobile devices should store minimal user data locally using secure APIs.
4. **Memory Management**: Linux uses virtual memory management with demand paging and kernel-level memory optimization. Draw It or Lose It can benefit from efficient caching and memory pooling in Linux. Clients manage memory via OS-level APIs, with iOS and Android offering lifecycle-aware memory handling.
5. **Distributed Systems and Networks**: The game should be built using a **RESTful API or WebSocket** architecture, allowing different clients to communicate with the server asynchronously. A CDN (Content Delivery Network) can ensure fast global access. Dependency issues such as device outages or slow connections can be handled via retries, load balancing, and failover servers.
6. **Security**: Use **HTTPS** and **OAuth2.0** for encrypted communication and secure authentication. Sensitive data should be encrypted at rest and in transit. Linux provides robust user-level and file-level permissions. Mobile platforms offer secure keychains and encrypted local storage (KeyStore on Android, Keychain on iOS). Regular updates and secure coding practices should be enforced across all platforms.