

<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 | <09/18/25> | <William-Vanderlinden> | <Added Executive Summary, Requirements, Design Constraints, Domain Model, Evaluation and Recommendations> |

**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) will design and begin implementing a web-based, multi-platform version of Draw It or Lose It for The Gaming Room (TGR). The new application will support one or more teams per game, multiple players per team, and enforce uniqueness of names for both games and teams. Additionally, the solution must guarantee that only one authoritative instance of the game exists in memory at any given time.

To meet these needs, CTS will apply proven software design patterns and object-oriented principles. Specifically, the Singleton pattern will be used to ensure only one instance of the game service exists, while inheritance and composition will keep the codebase modular and maintainable. The solution will be deployed as a web-based, distributed system capable of scaling across multiple platforms, with performance, concurrency, and extensibility built in from the foundation.

## Requirements

**Business Requirements**

* Expand Draw It or Lose It beyond Android to a multi-platform, web-based application.
* Ensure teams and players can easily join and participate in games.
* Provide a seamless user experience across devices.

**Technical Requirements**

* Support multiple teams per game and multiple players per team.
* Enforce unique game and team names.
* Guarantee only one game service instance exists at a time.
* Assign unique identifiers for all entities (game, team, player).
* Support real-time updates for timed rounds and gameplay.

## [Design Constraints](#_2et92p0)

The system must ensure that only one authoritative game instance exists in memory. This is achieved through the Singleton design pattern, which centralizes all game creation and management in a single service. In a distributed environment, thread safety and synchronization will be essential to maintain stability and prevent conflicts when multiple users interact simultaneously. Another constraint is the requirement for unique names for games and teams. This will be enforced by the GameService during creation and, at scale, supported by database uniqueness constraints to prevent duplicates.

Because the game must operate as a web-based distributed application, it needs to be stateless at the API level and able to scale horizontally. Real-time performance is also a constraint, as gameplay depends on timed rounds. Efficient data structures and concurrency mechanisms will be used to support low-latency operations. Finally, the design must remain extensible. By keeping the core classes modular and favoring composition, future features like scoring and puzzle libraries can be added without disrupting existing functionality.

## [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 centers on an Entity base class that provides shared fields such as id and name. The Game, Team, and Player classes all extend Entity, reducing duplication and ensuring consistent identity across objects. The GameService class, implemented as a Singleton, manages the lifecycle of all entities, enforcing unique names and generating identifiers.

The relationships among classes demonstrate composition: a Game contains multiple Teams, and each Team contains multiple Players. These associations allow the system to scale naturally as new teams and players are added. Object-oriented principles are applied throughout: inheritance through Entity, encapsulation via private attributes and controlled access methods, and polymorphism with consistent toString methods. Together, these design choices create a clear, efficient structure that satisfies the client’s requirements for uniqueness, scalability, and maintainability.

**"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 servers are less commonly used in enterprise hosting. While stable and user-friendly, they are costly and not as widely supported for large-scale deployments. | Linux is the industry standard for hosting web apps. It is free, open-source, highly customizable, and scalable. Strong community support makes it ideal for this project. | Windows servers integrate well with Microsoft technologies but require licensing fees. They are heavier on resources compared to Linux and less flexible for scaling. | Mobile devices are not suitable for hosting servers due to limited processing power and network reliability. |
| **Client Side** | Supporting Mac clients requires ensuring browser compatibility (Safari, Chrome) and possibly native desktop support. Development costs are moderate if using web standards. | Linux clients are less common, but browser-based access ensures compatibility with minimal additional cost. Native support is rarely needed. | Windows has the largest user base. Browser support (Edge, Chrome, Firefox) ensures accessibility. Development effort is modest using web standards. | Mobile client support is critical. Requires responsive design and possibly native apps for Android/iOS. Higher development cost if building and maintaining separate native apps. |
| **Development Tools** | Languages: Java, Python, JavaScript. Tools: IntelliJ IDEA, Xcode (if integrating with Apple ecosystems), Docker. | Languages: Java, Python, JavaScript. Tools: Eclipse, IntelliJ IDEA, Visual Studio Code, Docker. | Languages: Java, C#, JavaScript. Tools: Visual Studio, IntelliJ IDEA, Docker, PowerShell. | Languages: Java (Android), Swift (iOS), JavaScript (cross-platform). Tools: Android Studio, Xcode, React Native/Flutter frameworks. |

## 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**: Linux is the best choice for hosting due to its scalability, cost-effectiveness, and wide community support.
2. **Operating Systems Architectures**: A 64-bit Linux distribution (such as Ubuntu Server or CentOS) running on cloud infrastructure is recommended. This allows scalability and containerization (Docker/Kubernetes) for distributed deployments.
3. **Storage Management**: A relational database (e.g., PostgreSQL or MySQL) is recommended to enforce uniqueness of names and IDs while supporting queries and persistence.
4. **Memory Management**: Linux employs virtual memory, paging, and efficient resource allocation. Combined with the JVM’s garbage collection, it ensures stable runtime performance for the game application.
5. **Distributed Systems and Networks**: Communication across platforms can be managed with RESTful APIs and WebSockets for real-time play. Hosting in a cloud environment with load balancing ensures redundancy and minimizes outages.
6. **Security**: User authentication and authorization should use secure protocols (HTTPS, OAuth 2.0/JWT). Sensitive data must be encrypted both at rest and in transit. Firewalls, intrusion detection, and routine patching on the Linux platform will further protect user information.