

# **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 | 9/17/2023 | Anthony Fillmore | Built out Design Document |

## [Executive Summary](#_sbfa50wo7nsh)

The software design problem we need to tackle is to develop a web-based gaming application called "Draw It or Lose It". Critical software requirements include having multiple teams with distinct names, having only one game instance exist at any one time, and enabling real-time communication. My proposed solution encompasses data checks for unique names, enforcing each object type to have a unique identifier and full unit and integration testing to ensure code functionality. We can ensure we meet our technical and business requirements by following these suggestions.

.

## Requirements

**Business Requirements:**

* Ability to have one or more teams within a game.
* Each team can consist of multiple players.
* Unique game and team names to prevent duplicates.

**Technical Requirements:**

* Implement a mechanism for ensuring only one game instance exists in memory at any given time, using unique identifiers.
* User account management for player assignments.
* Data validation of names to ensure unique values.
* Database to assist with real time communication and storage
* Infrastructure to host webserver that is secure and scalable.

## [Design Constraints](#_2et92p0)

We have several design constraints to consider for our web game. First, enforcing the uniqueness of game and team names requires data validation checks within code. We also have a constraint of having only one instance of the game in memory, which requires unique identifiers for games, teams, and players. Lastly, we need to have a database that is dynamic and resilient to ensure real time communication of our web application, which will ensure multiple users can play the same game without issues. The constraints we have are critical to ensuring our game functions without issues and will require strategic build plans to ensure they are integrated seamlessly into our application.

## [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 GameService class is structured as a singleton class, ensuring that only one instance of it exists throughout the application. It has a 0 to many relationships with the Game class, a single GameService instance can manage multiple Game objects. This design aligns with the technical requirement of having a single GameService instance and efficiently manages game-related operations.

The Entity class serves as the base class for Game, Team, and Player. This inheritance relationship showcases the object-oriented programming principle of inheritance. It allows for code reuse and streamlining by inheriting common attributes and behaviors from the Entity class. This ensures that each object type shares attributes while allowing for their unique elements.

The Game class, derived from the Entity class, maintains a 0 to many relationships with the Team class. This signifies that a single Game object can be associated with multiple teams. This is crucial for accommodating multiple teams in a game session, adhering to the software requirement that a game can have one or more teams.

The Team class derived from the Entity class, has a 0 to many relationships with the Player class. This denotes that a single Team object can consist of multiple players. This relationship aligns with the requirement that each team can have multiple players.

The ProgramDriver class is reliant on the SingletonTester class.

Data Encapsulation is shown throughout the application in the form of privately declared class variables, ensuring that data is only exposed to intended mechanisms.

**"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** | Stable environment, user-friendly OS, platform security is high.  Licenses and hardware costs are high. Enterprise support may be available. | Open source solutions can help lower costs and give engineers the ability to support their own products. Great environment for hosting with plenty of options and tools to support it. Enterprise support lacking especially around free/open source solutions. | Requires window licenses for servers, which can increase cost. Platform is less secure and performance may not be optimal. Enterprise Support may be available if issues arise. | We should avoid hosting our web app on a mobile device as they are not suited for this task. Lacks the tools and scalability required for the application to be successful. |
| **Client Side** | Requires macOS skills, hardware cost is expensive. Secure platform. Has access to standardized web services like HTTP/HTTPS which makes this a good client option. | Open-source development, lower development costs, requires Linux admin skills. More control afforded to developers than other Operating systems. Linux based systems are not traditional end-user systems. | Requires Windows skills, potential for higher development costs. Security concerns more prevalent.  Has access to standardized web services like HTTP/HTTPS which makes this a good client option. | Requires mobile development skills, varying development costs for different platforms. Using tools like PhoneGap can make client code mobile device agnostic, making it a good choice for developing client side mobile applications. |
| **Development Tools** | **Languages:** Swift, Objective-C  **IDEs:** Xcode, Visual Studio Code, Atom, IntelliJ  **Other Tools:** Virtual Box, Docker | **Languages:** C, C++, Python, Ruby, Java **IDEs:** Visual Studio Code, Eclipse, JetBrains IDEs  **Other Tools:** Make, Virtual Box, Docker | **Languages**: C#, .NET, C++  **IDEs:** Atom, Visual Studio Code, IntellliJ  **Other Tools**: .NET Framework | **Languages:** Java (Android), Swift, Objective-C (iOS), PhoneGap  **IDEs:** Xcode, Android Studio  **Other Tools:** IOS/Android SDK |

## 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**:

To ensure our game can expand to other compute platforms, I’m recommending that ‘The Gaming Room’ adopts a web-based application model. This will allow their game to be usable on different platforms like Windows, macOS, and mobile devices, minus any cross-platform constraints. We can maintain a single code base this way, reducing technical debt and overall development costs.

1. **Operating Systems Architectures:**

The server-side of our client-server web model can be hosted on a stable Linux distribution like Ubuntu. We can ensure stability while lowering overall development costs by using a stable Ubuntu distribution, benefitting from a large open-source community and frequent OS updates. By leveraging a Linux kernel, we can provide robust security features and scalability. On the client-side, we can rely on web technologies and browser compatibility using HTTPS, ensuring a consistent user experience across various platforms. This design will enable cross-platform accessibility, provide scalability, and maintain security and stability on the server-side.

1. **Storage Management:**

For efficient storage management, we will recommend a relational database architecture like PostgreSQL. PostgreSQL will give us robust data management capabilities while making it easy to integrate with our web application. There is a large Postgres community we can interact with for additional support which makes this an attractive option.

1. **Memory Management**:

A web-based application model allows us to utilize the client’s device memory to load web content, while we use our server-side application to manage the stateful aspects of the game (players, game state) and the game logic. The workload ends up being shared between the client and the server, helping to optimize overall resource consumption by sharing the application workload.

1. **Distributed Systems and Networks**:

I would recommend a RESTful API architecture so that the game can communicate over HTTP, with the added benefit of being cross-platform compatible with most major platforms. We can use the API as a “middle person”, between the client and our server, ensuring data is exchanged smoothly no matter what the source of the communication is. We can improve the applications resilience against network outages (and other outages) by having server-side redundancy and failover options within different network segments. By having available application servers within different network segments, we can ensure application stability if a certain region or data center is affected.

1. **Security:**

We should use encryption protocols (HTTPS), user authentication, and authorization mechanisms. Utilizing SSL Certificates is recommended to ensure data integrity and to validate endpoint identity. Firewalls should be used to allow only required traffic, following a whitelist methodology of only allowing targeted traffic. Our Database should be protected against SQL Injection techniques by making sure we sanitize any user provided data. We should use parameterized queries when accessing the Database based on user provided data to further protect against SQL injection. If we collect and store any sensitive user data, we should be using an approved hashing algorithm to encrypt the data. Security reviews should be part of lifecycle management, prioritizing patching both our Kernel and dependent packages.