

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 | <10/01//23> | <Nathalie Morales> | <Initial version of the gaming system. Implemented core classes like Game, Player, Team, and SingletonTester. Integrated the GameService as a singleton to manage game instances. > |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room, a client of Creative Technology Solutions (CTS), seeks to expand the reach of their current Android game, "Draw It or Lose It", by developing a version that is supported by the web. The identified game, reminiscent of the early 1980s of the known television game "Win, Lose or Draw", requires teams to participate in guessing a random phrase, or random title, or any other thing based on the vast libraries drawings rendered. The biggest challenge identified lies in transitioning the game in development from a mobile-only platform to a good web-based multi-platform platform for gamers all over. In this document I propose a software design solution that goes an extra mile to address the requirements provided by the client, while I ensure a seamless and efficient transition to the web platform.

## Requirements

The Gaming Room has specified the following requirements for the web-based game application:

The game should support multiple teams.

Each team ought to have a selection of multiple players.

How the game is referred to or how the team is referred to must be unique.

Only a single identified instance can be handled in a memory at any time for the game being developed.

A unique identification is included in the game under development for each game instance, team, or player.

## [Design Constraints](#_2et92p0)

Developing "Draw It or Lose It" as a game that supports the web introduces some identifiable design constraints:

* Platform Independence: The game must be accessible and playable across various devices and browsers without compromising on performance or user experience.
* Scalability: Given the potential for multiple teams and players, the backend infrastructure ought to be scalable to handle the large number of active concurrent users.
* Real-time Interaction: The game requires real-time rendering of drawings and real-time interactions among players, demanding a robust real-time communication framework.
* Data Persistence: With distinct identities for matches, teams, and players, an efficient database system is required to store and retrieve data.
* Security is crucial, especially when working with web-based applications. This includes the security of user data and gaming sessions.
* State Management: Managing the state of the game, especially when only a single instance can exist in memory, poses challenges in a distributed environment.

The implications of these constraints include the need for a robust backend infrastructure, possibly using cloud services for scalability, the integration of real-time communication tools, and the implementation of a secure and efficient database system. Additionally, the software design must prioritize efficient memory management and state preservation techniques to ensure game integrity.

## [Domain Model](#_8h2ehzxfam4o)

"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.

The above smartly designed UML Class Diagram is used to represent the entire structure being used to develop the game application, detailing the relationships and functionalities of various classes.

**Entity Class:**

This is a foundational class that holds common attributes like id (of type long) and name (of type String).

It happens to provide constructors to initialize these attributes and methods to retrieve them (getId(), getName()) and a toString() method for representation.

**GameService Class:**

This class serves as a service layer for game management. It includes a list of games (games) as well as static variables for keeping track of the next IDs for games, players, and teams.

The Singleton design pattern is used to ensure that only one instance of the GameService class is present in memory at any given moment. The private constructor (GameService()) and the getInstance() method demonstrate this.

It provides methods to add games (addGame()) and retrieve them either by ID or name (getGame()). It also offers methods to get the count of games (getGameCount()) and fetch the next IDs for players and teams.

**Game Class:**

Represents individual games and contains a list of teams.

Provides methods to add teams (addTeam()) and a toString() method for representation.

**Team Class:**

Represents individual teams within games and holds a list of the available players.

Provides good adoptable methods to add players (addPlayer()) and a toString() method for representation.

**Player Class:**

Represents individual players within teams.

Provides a toString() method for representation.

ProgramDriver and SingletonTester Classes:

These classes are the utility or driver classes for the application. The ProgramDriver class happens to use the SingletonTester class implemented, as indicated when the following symbol has been used "<<uses>>" relationship.

**Relationships:**

The GameService class as seen in the UML has a one-to-many relationship with the Game class this clearly indicates that one game service has the ability to manage multiple games.

As seen, the Game class holds a one-to-many relationship along with the Team class, indicating that a single game can have numerous teams.

The Team class has a one-to-many link with the Player class, which means that a single team can include several players.

The Game, as well as Team, combined with Player classes all inherit from the identified Entity class, as shown by the open arrow pointing towards the Entity class. This inheritance indicates that these classes share common attributes and behaviors defined in the Entity class.

OOP - Object-Oriented Programming Principles:

* Inheritance: The Game, as well as Team, combined with Player classes inherit from the Entity class, promoting code reusability and a hierarchical structure.
* Encapsulation: Attributes in classes are private, ensuring data integrity and restricting direct access. Public methods are provided to interact with these attributes.
* Polymorphism: The toString() method is overridden in multiple classes, allowing different implementations depending on the class context.
* Singleton Pattern: The GameService class employs the Singleton design pattern to ensure only one instance exists in memory, meeting the software requirement efficiently.
* The UML diagram efficiently captures the structure and relationships of the game application, ensuring that the software requirements are met using object-oriented principles.

## [Evaluation](#_2o15spng8stw)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| **Server Side** | Mac servers, while not as popular as Linux for web hosting, offer a stable and secure environment. macOS is built on a UNIX-based foundation, providing robustness. However, Mac hardware tends to be more expensive, and there's less community support compared to Linux. | Linux is a popular choice for server-side hosting due to its stability, security, and open-source nature. It has a vast community, making it easier to find solutions to potential issues. Many web hosting services prefer Linux because of its flexibility and cost-effectiveness. | Windows Server is known for its user-friendly interface and integration with other Microsoft products. However, it might be more susceptible to malware compared to Linux. Licensing costs can also be a factor to consider. | Hosting a web-based software application on mobile devices isn't typical. Mobile devices usually act as clients. However, they can host local servers for development or testing purposes, but it's not suitable for production. |
| **Client Side** | Developing for Mac clients might require expertise in Swift or Objective-C for native applications. The cost can be higher due to the need for specific hardware (Mac machines) for development. Time considerations depend on the complexity of the application. | Developing for Linux clients can be cost-effective since many tools and libraries are open-source. However, considering the variety of Linux distributions, ensuring compatibility might take extra time. Expertise in languages like Python, C, or C++ might be needed. | Windows has a large user base, so developing for it can reach a broad audience. Expertise in C# and .NET might be required. Development time can vary, but tools like Visual Studio can streamline the process. | Developing for mobile requires expertise in Swift (iOS), Kotlin or Java (Android), and possibly cross-platform tools like Flutter or React Native. Costs can vary, but multiple device testing is crucial, which can add to the time required. |
| **Development Tools** | For Mac, Xcode is the primary IDE, supporting Swift and Objective-C. Other tools like Visual Studio Code or JetBrains AppCode can also be used. | Linux supports a wide range of languages. Common IDEs include Eclipse, IntelliJ IDEA, and Visual Studio Code. Languages like Python, C, C++, and Java are frequently used. | Windows developers often use Visual Studio, which supports a range of languages, primarily C#. Other tools like JetBrains Rider or Visual Studio Code are also popular. | For iOS, Xcode is essential. Android developers use Android Studio, which supports Kotlin and Java. Cross-platform tools include Flutter (Dart language) and React Native (JavaScript). |

## Recommendations

1. **Operating Platform**: My recommendation is to use "Draw It or Lose It" mostly on Linux because it is well known for being scalable, secure, and stable on its releases. Because it's open-source, it may be customized and adjusted to suit a variety of computer settings with ease. Furthermore, Linux-based solutions are readily available from various cloud service providers, which may be advantageous in the event that The Gaming Room wishes to grow or release the game to a larger player base in the road.
2. **Operating Systems Architectures**: Because the Linux operating system is based on a monolithic kernel, the majority of OS functions are performed in kernel space. Because everything is combined into a single, sizable executable process, this kind of design enables faster process execution. The fact that it supports both the 64-bit and 32-bit architectures contributes to its broad hardware compatibility.
3. **Storage Management**: My recommendation for any storage activity is to integrate PostgreSQL or simply MySQL as the relational database management system. Both are dependable, and compatible with Linux. Their ability to manage big databases effectively guarantees the safe and speedy retrieval of gaming data, user profiles, scores, and other pertinent information.
4. **Memory Management**: Linux uses a combination of paging and segmentation for memory management. It employs a virtual memory system, allowing processes to utilize more memory than physically available. The Swap Space in Linux acts as a virtual RAM, ensuring that when the physical RAM gets filled, the least used pages in memory can be 'swapped out' to the swap space, ensuring smooth operation of the game.
5. **Distributed Systems and Networks**: To allow "Draw It or Lose It" system to communicate between different platforms, a RESTful API can be implemented. This API will act as an intermediary, ensuring seamless communication between the game server and clients, regardless of the platform. Using JSON or XML as data interchange formats will ensure compatibility. For handling connectivity outages, a retry mechanism can be implemented, ensuring that if a request fails due to network issues, it's retried a few times before failing completely. Load balancers can be used to distribute incoming game traffic across multiple servers, ensuring high availability and reliability.
6. **Security**:

Security is paramount, especially when user data is involved. Here are the steps to ensure security on and between various platforms:

* Data Encryption: Use encryption protocols like TLS to encrypt data in transit. For data at rest, tools like LUKS can be used on Linux.
* Authentication and Authorization: Implement OAuth or JWT for secure user authentication and authorization.
* Regular Updates: Ensure that the Linux OS and all software components are regularly updated to patch any known vulnerabilities.
* Firewalls: to block any unauthorized access to the game servers I would suggest to Use firewalls.
* Intrusion Detection Systems (IDS): Implement IDS to monitor and detect any malicious activities.
* Backup: Regularly backup game data to recover from potential data loss scenarios.