

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 | 11/17/2024 | Dustin Davis | Design Document Initial Draft |

**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 is transitioning its Draw It or Lose It game from an Android-only app to a web-based platform that supports multiple devices. The key software design challenges are ensuring that each game, team, and player has a unique identifier, enabling multiple teams and players to interact in a single game, and maintaining a system where only one instance of the game exists at any time. The solution involves creating a GameService using the Singleton pattern to manage all games and players, while ensuring that each game, team, and player has a unique ID. This structure also enforced unique names for games and teams, preventing conflicts. The Entity class will serve as the base class for Game, Team, and Player, streamlining the management of unique identifiers and names.

To proceed, we recommend implementing the backend services with a focus on the Singleton and Entity patterns, followed by building a user-friendly web interface for game creation and player management. Testing across multiple platforms will be crucial to ensure the game runs smoothly on all devices. This approach will provide a scalable, maintainable solution that meets the client’s requirements, offering a seamless gaming experience across web and mobile platforms.

## Requirements

*Business Requirements:*

* *Multi-Platform Support: The game should be transitioned from an Android-only app to a web-based app, supporting multiple platforms like desktop, mobile, and browser.*
* *Unique Game and Team Names: Game names and team names must be unique across the system to avoid conflicts when users select names.*
* *Multi-Team Support: A single game can involve one or more teams, with each team competing against each other.*
* *Team Structure: Each team must be able to have multiple players assigned to it, facilitating a team-based gameplay experience.*

*Technical Requirements:*

* *Unique Identifiers: Each instance of a game, team, and player must have a unique identifier (ID) to ensure that no two identical entities exist in memory at the same time.*
* *Single Game Instance: Only one instance of any given game can exist in memory at a time to maintain consistency across all users.*
* *Efficient Management of Entities: The system should be able to efficiently create, manage, and query games, teams, and players while maintaining the uniqueness of their names and IDs.*
* *Scalable Architecture: The backend architecture must be scalable to handle a growing number of games, teams, and players as the user base increases, especially when transitioning to a web-based platform.*

## [Design Constraints](#_2et92p0)

**Network Latency and Bandwidth:**

* Constraint: Slow internet or high traffic can cause delays in gameplay.
* Implication: The game may lag or become unresponsive, so optimized network usage and efficient data transfer methods can minimize these issues.

**Scalability:**

* Constraint: The game must support many users and games running at once.
* Implication: We need a system that can grow with more players, like using multiple servers or cloud services to handle high traffic and ensure smooth performance.

**State Management and Consistency:**

* Constraint: Keeping the game state the same across different devices can be difficult.
* Implication: The game must ensure that everyone sees the same thing in real time, and handle multiple players actions without delays or users seeing different things.

**Security and Data Privacy:**

* Constraint: The game is online, so sensitive player data must be protected from unauthorized access.
* Implication: We need secure login systems, encrypted data, and protections against hacking to keep player information safe and comply with privacy laws.

**Cross Browser and Cross Platform Compatibility:**

* Constraint: The game must work across different web browsers and devices.
* Implication: The user interface must adapt to different screen sizes and input methods, like mouse clicks or finger taps, this also means it needs to be tested on each platform to ensure it works correctly everywhere.

**Game Logic and Real Time Interaction:**

* Constraint: The game requires real time actions like drawing and guessing, which must be synchronized across all players.
* Implication: A need for real time communication to update all players immediately and manage game actions with no delays.

**Data Storage and Persistence:**

* Constraint: Player data, scores, and game history need to be stored securely and retrieved quickly.
* Implication: A fast, reliable database is needed that can also backup data, be consistent and needs to be available across all the players sessions.

## [System Architecture View](#_ilbxbyevv6b6)

* Players play the game on the frontend which is the web site or app.
* The backend makes sure the game is working correctly and sends updates to the players in real time.
* The database stores player info, scores, and game data.
* Real time updates let everyone see changes instantly.
* Security keeps everything safe for users.
* Servers handle all the heavy lifting when it comes to things working smoothly.

## [Domain Model](#_8h2ehzxfam4o)

**ProgramDriver Class:**

* Purpose: This is the entry point for the application, containing the main() method that initiates the game. Its responsible for starting the game and managing the execution flow.
* Relationships: It uses the SingletonTester class to verify that the GameService class adheres to the Singleton pattern.
* OOP Principle: Dependency Injection – is demonstrated here, as the ProgramDriver class depends on the SingletonTester class to ensure that the GameService class has only one instance, This use of dependencies ensures that components are tested for correctness before the game starts.
* Software Requirements Fulfillment: By testing the Singleton pattern, the ProgramDriver ensures that only one instance of the game management system (GameService) exists, which aligns with the requirement of maintaining a single game instance in memory at all times.

**SingletonTester Class:**

* Purpose: This class is used to validate that the GameService class is following the Singleton design pattern, which ensures only one instance of the service exists during the applications lifecycle.
* Relationships: It interacts directly with the GameService class to test its Singleton implementation, ensuring that no more than one instance is created.
* OOP Principle: Singleton Pattern – is demonstrated here, confirming that only one instance of GameService can exist in the system. This pattern is critical for managing the overall game state in a centralized manner.
* Software Requirement Fulfillment: By ensuring that GameService is a Singleton, this class guarantees the software requirement that only one instance of the game management service should be active at any time, preventing conflicts or inconsistencies.

**Entity Class:**

* Purpose: The Entity class is an abstract or base class that holds common properties like id and name, which are shared across different entities such as games, teams, and players.
* Relationships: It is inherited by the Game, Team, and Player classes, allowing them to reuse the id and name properties and related methods (getId(), getName(), and toString()).
* OOP Principle: Inheritance is used here. The Game, Team, and Player classes inherit common attributes from Entity, promoting code reuse and reducing redundancy.
* Software Requirement Fulfillment: Inheritance helps maintain a uniform structure across different game objects while ensuring each has a unique identifier and name.

**GameService Class:**

* Purpose: The GameService class manages the overall game state, ensuring the creation of new games, adding teams and players, and tracking the number of games. It is responsible for keeping the game system running smoothly.
* Relationships: The GameService class aggregates multiple Game objects via the List<Game>, managing all active games. It also creates and manages teams and players within those games.
* OOP Principle: The Singleton Patter is applied to GameService to ensure that only one instance is available throughout the application, centralizing game management. Aggregation is also shown, where GameService holds multiple Game objects.
* Software Requirement Fulfillment: Managing game creation and keeping track of game instances, the GameService fulfills the requirement that only one instance of the game exists at any time. The aggregation of games ensures flexibility in managing multiple game sessions.

**Game Class:**

* Purpose: The Game class represents a single game instance. It manages the teams involved in the game and provides methods for adding teams and handling game data.
* Relationships: The Game class has multiple Team objects via List<Team>. It inherits from the Entity class, which means it shares the id and name attributes with teams and players.
* OOP Principle: Composition is demonstrated here, as a Game object contains teams. If a game is deleted, its teams are also deleted, which shows the strong relationship between the two. Inheritance is also used to share common attributes with Team and Player.
* Software Requirement Fulfillment: Composition ensures that a game always has at least one team and that the game cannot exist without teams. This relationship supports the requirement that a game must involve multiple teams.

**Team Class:**

* Purpose: The Team class represents a group of players participating in the game. It manages a list of players and provides methods for adding them to a team.\
* Relationships: The Team class has multiple Player objects via List<Player>. It inherits from the Entity class, giving it a unique id and name.
* OOP Principle: Composition is again used here, as a team cannot exist without players. The Team class "owns" the Player objects, and deleting a team also deletes its players. Inheritance allows the Team class to reuse the id and name attributes from Entity.
* Software Requirement Fulfillment: Composition ensures that each team must have multiple players, fulfilling the requirement that teams consist of multiple players. The inheritance of id and name ensures uniqueness for each team.

**Player Class:**

* Purpose: The Player class represents an individual player in the game, with an id and name. It is the basic unit of a team.
* Relationships: The Player class inherits from the Entity class, which means it shares common attributes such as id and name with the Game and Team classes.
* OOP Principle: Inheritance is used here, as Player inherits from Entity to share common attributes like id and name, promoting code reuse and consistency across the system.
* Software Requirement Fulfillment: Inheritance ensures that each player has a unique id and name, fulfilling the requirement for unique identifiers for each player. The class is flexible and can easily be extended to add more player-related functionality if needed.

**"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** | Mac OS is a Unix-based system known for its user-friendly interface, security, and reliability. It works well for smaller web applications and development with tools like Xcode. However, it’s less common for web hosting compared to Linux, with fewer server resources and higher costs, and may not perform as well for larger applications. | Linux is an open-source, Unix-based operating system widely used for web hosting due to its stability, flexibility, and cost-effectiveness. It supports a wide range of web technologies and server tools, making it ideal for large-scale applications. Its advantages include being highly customizable, secure, and free. However, it can be more complex to set up and manage, especially for beginners, and may require more technical expertise compared to user-friendly systems. | Windows OS is a popular operating system known for its ease of use and compatibility with many software applications. It offers strong support for web development tools like .NET and IIS (Internet Information Services), making it a good choice for hosting web applications built on Microsoft technologies. Its advantages include a user-friendly interface, extensive support, and integration with other Microsoft products. However, Windows can be more expensive than Linux, less flexible, and may require more resources, making it less efficient for large-scale or resource-intensive applications. | Mobile devices, running iOS or Android, are optimized for apps with great user experiences and portability. They offer strong app ecosystems and integration with device features like cameras and sensors. However, they are not ideal for hosting web applications due to limited resources and processing power. Additionally, developing for mobile often requires optimization for each platform, adding complexity for cross-platform support. |
| **Client Side** | Developing for Mac requires considering cost (Mac-specific tools and hardware), time (extra testing for compatibility), and expertise (knowledge of Mac development tools like Xcode and languages like Swift). You need to understand macOS features and optimize the app for smooth performance across different versions. | Developing for Linux involves considering cost (Linux is free, but you may need specific hardware or tools), time (more time may be needed for setup and testing across different Linux distributions), and expertise (knowledge of Linux environments, command line tools, and server setup). You’ll need to understand how Linux works, how to manage dependencies, and ensure your app runs well on various Linux versions and configurations. | Developing for Windows involves considering cost (Windows licenses and development tools like Visual Studio), time (more testing may be needed for different Windows versions and configurations), and expertise (knowledge of Windows-specific tools like .NET, IIS, and PowerShell). You’ll need to be familiar with the Windows environment and how to integrate your app with Microsoft technologies, which can require specialized skills for proper setup and optimization. | Developing for mobile devices requires considering cost (developer accounts for app stores), time (extra testing for both iOS and Android), and expertise (knowledge of tools like Xcode and Android Studio, plus languages like Swift and Kotlin). You also need to optimize your app for different devices and screen sizes. |
| **Development Tools** | To develop software for Mac, you’ll mainly use Swift or Objective-C as the programming languages. The main tool for building Mac apps is Xcode, which is an integrated development environment (IDE) that provides everything you need, including a code editor, simulator, and debugging tools. For certain tasks, you might also use Terminal (command-line tools) and Cocoa (a framework for building Mac applications). Xcode is free to download and works well for developing apps specifically for macOS. | To develop software for Linux, you can use programming languages like C, C++, Python, or Java. Common tools include GCC (GNU Compiler Collection) for compiling code, Visual Studio Code or Eclipse as IDEs for writing and editing code, and Terminal for using command-line tools. For web development, you might also use Apache or NGINX for web servers, and MySQL or PostgreSQL for databases. Linux is flexible and supports many development environments, making it a good choice for server-side and open-source projects. | To develop for Windows, you’ll use languages like C#, C++, or Python. The main tool is Visual Studio, an IDE that helps with coding, debugging, and compiling. For web apps, you might use ASP.NET and SQL Server for databases. Other tools include IIS for web hosting and PowerShell for automation. Visual Studio is free for smaller projects and is widely used for Windows development. | To develop for mobile devices, you’ll use Swift for iOS (Apple) and Kotlin or Java for Android. The main tools are Xcode for iOS development and Android Studio for Android development. Both are integrated development environments (IDEs) that help with writing code, testing, and debugging. For cross-platform development, you can use tools like Flutter or React Native. These tools help you build apps that work on both iOS and Android with a single codebase. |

## 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**: For hosting the web-based game, Linux is the best choice due to its stability, performance, and cost-effectiveness. It’s ideal for large-scale web applications. If mobile versions are needed, use Xcode for iOS and Android Studio for Android to build the game apps.
2. **Operating Systems Architectures**: Linux has a monolithic kernel that manages system resources like CPU and memory. It uses a hierarchical file system, starting from the root directory. The OS includes a shell for user commands, system utilities, and libraries to run applications. Linux supports multitasking, multi-user access, and device drivers for hardware management, making it flexible and customizable.
3. **Storage Management**: Using a relational database like MySQL or PostgreSQL for structured data, especially if the game needs to store player information, game stats, or other relational data. For file storage (like images or game assets), consider using cloud storage services like Amazon S3 or Google Cloud Storage for easy access and scalability. This combination ensures efficient, organized storage while keeping costs low and performance high.
4. **Memory Management**: For memory management, use dynamic memory allocation to request and release memory as needed. Memory pools can improve performance by reusing memory blocks, while garbage collection helps prevent memory leaks. Caching frequently used data can speed up access and reduce delays. These techniques ensure the game runs smoothly and efficiently.
5. **Distributed Systems and Networks**: For distributed systems and networks, use a client-server model where the game server handles requests from multiple clients. Implement load balancing to distribute traffic evenly across servers. Use caching and content delivery networks (CDNs) to reduce latency and speed up content delivery. Secure the network with encryption and authentication protocols to protect player data. This setup will ensure scalability, reliability, and security for the game.
6. **Security**: For security, use encryption to protect data, multi-factor authentication for user verification, and keep software updated to fix vulnerabilities. Implement firewalls and intrusion detection to monitor for threats, and apply least privilege access to limit user access to necessary data. These steps will help protect the game and user information.