

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 | 03/23/25 | Kileigh Adams | First Draft |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room has engaged Creative Technology Solutions to transform their Android game "Draw It or Lose It" into a web-based application accessible across multiple platforms. The primary challenge is to preserve the competitive team-based gameplay experience while meeting both business and technical requirements.

From a business perspective, the application must support games that include one or more teams, with each team consisting of multiple players who work together during play. It is also essential that game and team names are unique so that users can verify name availability when forming or joining a team. Furthermore, to maintain consistent game state and data integrity, only one instance of the game can exist in memory at any given time.

On the technical side, the application is being developed using modern Java techniques and object-oriented design principles within a distributed web-based environment. The design incorporates a base class called Entity, from which the Game, Team, and Player classes inherit common attributes such as an identifier and a name. This ensures code reuse and simplifies future maintenance. The GameService class, which manages game instances, uses the singleton design pattern to guarantee that only one active game instance exists. In addition, the iterator pattern is employed in methods that add and retrieve games, teams, and players to enforce unique naming constraints efficiently.

## [Design Constraints](#_2et92p0)

One requirement is that only one game instance exists in memory at any given time. In a situation where many users access the application at once, maintaining a single game state is challenging. This is handled by using the singleton design pattern in the **GameService** class. The developers must ensure that the game state is properly managed to avoid any conflicts or data errors.

Another constraint is the need to check that names are unique. Since the game can have multiple teams and each team can have several players, the application must verify that no two games or teams have the same name. The solution uses the iterator pattern in the **add** and **get** methods to check for duplicates and assign unique identifiers to games, teams and players.

The application must also run smoothly on a web based distributed network. This means there will be challenges regarding network latency, data synchronization and ensuring that the experience is consistent across different devices and operating systems. These issues must be considered during development to maintain a reliable and user-friendly game experience.

## 

## [Domain Model](#_8h2ehzxfam4o)

Classes and Their Relationships:

* ProgramDriver and SingletonTester:

ProgramDriver: Contains the main() method, serving as the application's entry point. It uses SingletonTester to test the singleton design pattern.

* Entity and Its Subclasses:

Entity: Base class with id and name attributes and methods like getId() and getName().

Inherited by Game, Team, and Player, promoting code reuse and consistency.

* GameService and Other Classes:
* GameService: Manages collections of Game, Team, and Player instances.
* Uses composition (0...\*) to indicate ownership of multiple instances.
* Game: Maintains a list of Team instances.
* Team: Holds a list of Player instances, reflecting a hierarchical structure.

Object-Oriented Programming Principles:

* Encapsulation:

Private attributes and public getter methods ensure controlled data access.

* Inheritance:

Entity as a parent class enables subclasses to inherit attributes, reducing redundancy.

* Polymorphism:

Subclasses like Game can override methods such as toString() for customized behavior.

* Abstraction:

GameService offers high-level methods to hide complexity, simplifying usage.

* Singleton Design Pattern:

GameService ensures a single instance with a private static attribute and getInstance() method.

* Composition:

Strong ownership relationships between GameService, Game, Team, and Player.

Fulfillment of Software Requirements:

* Supports multiple teams per game and multiple players per team through composition.
* Ensures unique game and team names via validation methods in GameService.
* Maintains a single game instance using the singleton pattern in GameService.
* Creates unique identifiers for each instance using the id attribute from Entity

**"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** | macOS offers a refined Unix environment, supporting web servers like Apache and Nginx. It provides a secure and efficient platform well-suited for development and testing. However, with Apple's focus shifting away from server innovations and the high cost of hardware, macOS might not be the best choice for large-scale production environments. | Linux is recognized as a reliable platform for running web-based game applications. Known for its stability and efficiency, Linux distributions like Ubuntu Server, CentOS, or Red Hat provide a strong foundation for managing multiple game sessions at a low cost. Its flexibility and robust support for container-based tools facilitate scalable deployment, but it requires administrative expertise due to its reliance on command-line management. | Windows Server offers a familiar environment, particularly for organizations aligned with the Microsoft ecosystem. It features numerous graphical administration tools that simplify daily tasks and integrate well with enterprise-grade frameworks. However, it involves licensing fees and higher resource consumption, which could impact cost-effectiveness when scaling to handle heavy workloads. | Mobile devices simply aren't built for hosting server components. They lack the processing power, memory, and network stability needed for reliable server operations. They work great as clients with their touch interfaces and portability, but the actual game server needs to run on proper infrastructure like Linux or Windows servers. |
| **Client Side** | macOS offers a refined user experience with consistent interfaces and seamless integration of design and functionality. Tools like Safari Developer Tools facilitate testing and debugging web applications. Although development costs may be higher due to premium hardware requirements, the platform provides a polished experience appealing to users who prioritize aesthetics and performance. | Linux provides versatility for designing client applications that require seamless interaction across desktops and devices. While the diverse range of Linux distributions may necessitate additional compatibility testing, the platform's command-line tools and open-source nature ensure flexible and efficient development. | As the most widely used desktop platform, Windows offers extensive reach for client applications. Its familiar interface minimizes the learning curve for users, and broad hardware compatibility ensures consistent performance across different devices. While corporate configurations may require additional testing, Windows' reliability and enterprise integration make it a suitable choice for delivering rich, interactive experiences. | Mobile devices demand responsive, adaptive design to meet user expectations for fast and intuitive interfaces. Whether through Progressive Web Apps or dedicated mobile applications, native capabilities and cross-platform frameworks help ensure consistent performance. The wide range of device models and screen sizes necessitates thorough testing, but a well-optimized approach can lead to a highly engaging experience. |
| **Development Tools** | For macOS development, developers often use a mix of native tools and cross-platform aids. Xcode is the main tool for building native interfaces with Swift or Objective-C. Java development is also supported by the same IDEs used on Linux. The macOS Terminal, with package managers like Homebrew, makes managing dependencies straightforward in its stable Unix environment. | Linux is a great platform for backend services, with Java being a popular choice supported by IDEs like Eclipse and IntelliJ IDEA. Text editors such as Visual Studio Code enhance productivity, while command-line tools and Git for version control keep workflows efficient and flexible. Docker makes it easy to manage deployments across different Linux distributions. | For Windows, Visual Studio offers a robust integrated environment that handles both frontend and backend development, supporting languages like C# and Java when needed. The ecosystem is rich with debugging, testing, and deployment tools, which makes managing complex projects easier, although licensing and resource requirements can sometimes be challenging. | Mobile development needs a platform-focused approach. Android Studio (for Java or Kotlin) and Xcode (for Swift) are ideal for their respective platforms. Alternatively, cross-platform frameworks like React Native or Flutter let you use a single code base for both iOS and Android. Tools like emulators and BrowserStack help ensure your app works well on different devices and screen sizes. |

## 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:** Opt for Linux as the core server-side platform. Its ability to deliver cost-effective and stable solutions makes it an excellent choice. By implementing a web-based client interface, the game becomes accessible on Windows, macOS, and mobile devices without the need for additional platform-specific adaptations.
2. **Operating Systems Architectures**: Linux is set up in a way that's both modular and backed by a solid monolithic kernel, which makes it perfect for running game servers. It's great at keeping processes separate, so session conflicts are a non-issue, and it also does a good job of protecting memory. Plus, it works well with containerization tools like Docker, making it easy to deploy and scale across various environments.
3. **Storage Management**: Use a layered storage approach with PostgreSQL to handle structured data, Redis for lightning-fast in-memory tasks, and object storage for managing static assets. This combination strikes a good balance between speed and reliability, ensuring quality performance and data integrity.
4. **Memory Management**: Linux’s memory management, including virtual memory, demand paging, and memory mapping, allows for efficient use of resources. Coupled with shared libraries and automatic garbage collection, this ensures the game runs smoothly, adapting to varying loads dynamically.
5. **Distributed Systems and Networks**: To handle the main operations, go with RESTful APIs, and for real-time communication, use WebSocket connections. Microservices architecture is a smart choice; it allows each component to be managed independently, making the system more robust and scalable. To ensure gamers enjoy a smooth experience worldwide, introduce a CDN to minimize latency.
6. **Security**: Prioritize strong security by using TLS to encrypt data during transmission and OAuth 2.0 for authentication. Make sure to hash passwords with reliable algorithms and run regular security audits. You can also tap into Linux features like SELinux and firewalls to add an extra layer of protection for user data.