

Isreal Banda

# **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 | 11/19/2024 | Isreal Banda | Refactored the program to introduce an Entity base class, streamlining shared attributes and behaviors for Game, Team, and Player through inheritance. Additionally, we enforced the Singleton pattern in GameService and ensured unique names for games, teams, and player using the iterator pattern. |

**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’s existing game, Draw It or Lose It, is currently limited to Android devices. Expanding it into a web-based application presents an opportunity to reach a broader audience across multiple platforms. This transition requires designing a scalable, efficient, and user-friendly system that meets specific software requirements while enabling seamless gameplay for users.

Key Challenges might include:

* Supporting multiple teams and players while ensuring unique team and game names.
* Maintaining data integrity by guaranteeing only one instance of the game is in memory at any given time.
* Providing a mechanism for identifying and managing unique identifiers for games, teams, and players.
* Laying a foundation that can later be tailored to the necessary hardware requirements.

Proposed Solutions:

* Centralized Game Management: Use a Singleton design pattern to ensure only one instance of the game exists at any given time, preserving memory efficiency and preventing data conflicts.
* Unique Identifier System: Implementing a robust mechanism for generating unique ID’s for games, teams, and players to prevent naming conflicts and ensure smooth gameplay management.
* Dynamic Team and Player Support: Designing a flexible system that supports multiple teams with multiple players, including validation logic to enforce unique names for teams and games.
* User-Friendly Interface: Developing an intuitive, responsive web interface for players to create or join games, manage teams, and interact with the game.
* Validation and Security: Ensuring data integrity with server-side validation for names and IDs while implementing secure authentication methods for user accounts.

Critical Information:

* Platform independence: Our design will ensure compatibility across devices.
* Development Phases: The project will progress in phases, starting with architecture and API development, followed by interface design and integration.
* Collaboration: Continuous communication between CTS and The Gaming Room staff will be critical to ensure alignment on game rules, functionality and user expectations.
* Timeline: We will provide detailed timeline for deliverable after finalizing the design specifications.
* Futureproofing: The system will be modular, allowing for future expansion and enhancements, such as adding more game modes or platform specific optimizations.

## Requirements

Single Game Instance:

* Business Requirement – Ensure only one instance of the game exists in memory at any time to prevent duplication or conflicts during gameplay.
* Technical Requirement – Implement the Singleton design pattern in the backend to enforce a single instance across distributed environments.

Unique Identifies for Games, Teams, and Players:

* Business Requirement – Allow users to create unique names for games, teams, and players to avoid confusion or naming conflicts.
* Technical Requirement – Develop a mechanism to validate and enforce unique names using a database with unique constraints or a centralized verification system.

Cross-Platform Compatibility:

* Business Requirement – Make the application accessible to users on desktops, tablet, and mobile devices.
* Technical Requirement – use responsive web technologies to ensure seamless user experiences across platforms.

Distributed System Architecture:

* Business Requirement - Ensure the game operates smoothly in a web-based environment accessible to multiple users simultaneously.
* Technical Requirement - Design a scalable backend using RESTful APIs, implement load balancing, and ensure consistency across distributed servers.

Performance and Real-Time Updates:

* Business Requirement - Support gameplay features such as rendering images in real-time and managing team interactions within strict time limits.
* Technical Requirement - Implement efficient algorithms, low-latency communication protocols, and real-time technologies like WebSockets to support these features.

Security Requirements:

* Business Requirement - Protect user data and prevent unauthorized access to the game and its features.
* Technical Requirement - Use secure protocols (HTTPS), implement encryption, and enforce authentication mechanisms such as OAuth2 or similar standards.

Scalability and Future Expansion:

* Business Requirement - Ensure the application can grow to support additional features or larger user bases in the future.
* Technical Requirement - Use a modular architecture, distributed databases, and scalable technologies like microservices to allow for future growth.

Resource and Budget Constraints

* Business Requirement - Deliver the project within the client’s budget and resource limitations.
* Technical Requirement - Prioritize cost-effective tools, open-source frameworks, and phased development to balance quality with resource constraints.

## [Design Constraints](#_2et92p0)

Single Game Instance:

* Constraint – Only one instance of the game should exist in memory at any time.
* Implication – Requires careful state management and synchronization across users, adding complexity to the backend design.

Unique Identifies for Games, Teams and Players:

* Constraint – All game, team and player names must be unique.
* Implication – Demands real-time validation and database-level constraints to handle uniqueness efficiently, which can increase server load and design complexity.

Cross-Platform Compatibility:

* Constraint – The application must function seamlessly on desktops, tablets, and mobile devices.
* Implication – Requires responsive design, comprehensive testing and platform independent technologies, increasing development and QA efforts.

Performance and Real-Time Updates:

* Constraint – The game must support real-time rendering of images and interactions under strict time limits.
* Implication – Adds the need for optimized algorithms, asynchronous processing, and robust server infrastructure to handle latency and concurrency.

## [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)

Entity Class (Base Class):

* Purpose: Acts as a foundation for shared attributes like id and name.
* Reusability: Other classes inherit from it to avoid repetition of common properties.

GameService Class:

* Purpose: Manages all games and ensure there’s only one active game service in the system.
* Singleton Pattern: Ensures only one instance of this class exists at any time.
* Functions: Handles adding and retrieving games, and tracks IDs for new games, teams and players.

Game Class:

* Purpose: Represents a single game and contains a list of team participating.
* Inheritance: Inherits id and name from Entity.
* Relationship: A game can have multiple teams.

Team Class:

* Purpose: Represents a team, containing a list of players.
* Inheritance: Inherits from Entity.
* Relationship: A team can have multiple players.

Player Class:

* Purpose: Represents and individual player.
* Inheritance: Inherits from Entity.

ProgramDriver Class:

* Purpose: The main entry point to run the program.

SingletonTester Class:

* Purpose: Tests that the Singleton pattern in GameService works correctly.

Features of Diagram:

* Encapsulation: All important data is kept private, with public methods to access them securely.
* Inheritance: The Entity class reduces redundancy by providing shared attributes to Game, Team and Player.
* Singleton Pattern: Ensures only one instance of the GameService class exists, meeting the clients requirement for a single game service.
* Composition: Games have teams, and teams have players, showing how components are nested hierarchically.

**"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 can host web-based application using server tools like Apache, Nginx, or macOS Server. While masOS is stable and user-friendly, its licensing costs are higher than Linux. Additionally, macOS hardware is expensive which could increase costs for large-scale deployments. | Linux is an ideal platform for hosting web applications because of its flexibility, open-source nature, and lack of licensing costs. It supports various web servers like Apache and Nginx and is scalable. However, Linux may require specialized expertise to manage efficiently. | Windows Server offer a reliable platform for hosting web-based applications, with great support for enterprise tool like IIS. Licensing costs are a major downside, but it provides seamless integration with other Microsoft services, making it ideal for enterprise use. | Mobile platforms rely on backend servers to host web-based applications. The server-side deployment must be cross-platform, using tools like RESTful APIs to ensure compatibility with iOS and Android devices. Scalability and performance must also be optimized to handle mobile traffic. |
| **Client Side** | Development for Mac desktop clients requires ensuing compatibility with Safari and other web browsers. Costs include access to macOS systems for testing and development. Expertise in responsive design and cross-platform tools like React is essential. | Linux clients require testing for compatibility with web browsers like Firefox and Chromium. Linux users typically prefer open-source solutions, so developers should ensure accessibility. The cost o development for Linux is relatively low, but expertise in testing across various distributions is required. | Development for Windows clients involves testing compatibility with browsers like Edge, Chrome, and Firefox. While Windows is widely used, developers must account for potential variations in browser behavior. Costs include Windows licensing for development and testing environments. | Mobile platforms require a responsive HTML interface that works seamlessly on both iOS and Android. Development teams must use frameworks like Flutter or React Native for efficient cross-platform support. Testing on multiple device types is necessary to ensure performance and usability. |
| **Development Tools** | Development for Mac clients typically uses Xcode and tools like Visual Studio Code. Licensing costs for macOS systems are significant, and developers need expertise in macOS-specific debugging and optimization tools. | Development on Linux uses free and open-source tools like Eclipse, IntelliJ IDEA, or VS Code. No licensing costs make it cost-effective, but the team needs strong expertise in configuring Linux environments and resolving compatibility issues across distributions. | Windows development involves tools like Visual Studio, which may incur licensing costs. Windows offers extensive support and debugging tools, but developers need expertise in Windows-specific frameworks and testing. | Mobile development for iOS and Android typically requires IDEs like Android Studio and Xcode, as well as cross-platform tools like Flutter or React Native. Licensing costs for Apple systems are higher, and developers must be proficient in mobile app optimization and testing. |

## 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 supports multiple server-side tools and frameworks, such as Apache, Nginx, and Docker, which provide the scalability and performance needed for a game that will handle thousand of concurrent users. It’s open-source nature eliminates licensing costs, reducing long-term expenses for The Gaming Room, while also providing wide support for various programming languages and development tools.
2. **Operating Systems Architectures**: The architecture of Linux is highly modular, allowing developers to configure the operating system specifically for the needs of *Draw It or Lost It*. It’s architecture support multi-threaded operations and efficient process scheduling, which is essential for a web-based game that needs to handle numerous simultaneous requests from users across platforms. Additionally, Linux’s kernel-level optimizations ensure stable and fast performance, making it a popular choice for hosting large-scale web applications.
3. **Storage Management**: For storage management, I would recommend using a cloud-based solutions, such as Amazon S3, combined with a relational database like MySQL or PostgreSQL. Cloud storage provides reliable, scalable, and secure options for storing user data, game assets, and other critical information. PostgreSQL integrates seamlessly with Linux and supports advanced features like data indexing and concurrent access, which are vital for a responsive and dynamic game application. It also offers features such as automatic backups and high availability, which minimize the risk of data loss or downtime.
4. **Memory Management**: Linux’s memory management system uses techniques like paging, segmentation, and swapping to allocate resources efficiently. These techniques ensure that the application runs smoothly without excessive memory consumption. For Draw It or Lose It, this means the server can handle multiple players interacting with the game simultaneously, even during peak usage times. Additionally, Linux’s virtual memory system ensures that even under high loads, the application can allocate and reclaim memory dynamically to avoid performance bottlenecks.
5. **Distributed Systems and Networks**: In order for Draw It or Lose It, to communicate between various platforms, distributed systems and networks must be optimized for reliability and efficiency. A RESTful API architecture ensures compatibility across platforms by allowing clients to make HTTP requests to access game data. Using frameworks like Kubernetes or Docker Swarm for containerization, the application can be deployed as microservices, ensuring fault tolerance and scalability. To manage dependencies in distributed systems, techniques such as load balancing, redundancy, and failover strategies should be implemented to minimize downtime and maintain connectivity during outages. Network protocols like WebSocket can be used to enable real-time interactions between the server and clients, ensuring a seamless gaming experience across all platforms.
6. **Security**: Security is a critical requirement for protecting user information on and between platforms. The Linux platform is known for its robust security features, including configurable firewalls, encryption tools, and built-in user authentication systems. Sensitive user information, such as passwords and game data, should be encrypted using HTTPS for data transmission and AES for data at rest. Additionally, role-based access control can be implemented to ensure that only authorized users can access sensitive data or perform admin tasks. Regular vulnerability assessments and patching will further enhance the platform's security posture.