

The Gaming Room

# **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 | 07/20/2024 | Darrell Walker | Initial creation of the software design document. |

**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 seeks to expand its game, Draw It or Lose It, from an Android-only app to a web-based game that can be accessed from multiple platforms.

## Requirements

The key software requirement is to allow multiple teams, each with multiple players, and ensure unique game and team names to avoid conflicts. Additionally, only one instance of the game should exist in memory at any time. This document outlines the software design and implementation plan, addressing these requirements and providing a robust, scalable, and user-friendly solution.

## [Design Constraints](#_2et92p0)

Developing the game application in a web-based distributed environment presents several roadblocks:

1. **Scalability**: The application must handle multiple concurrent users efficiently.
2. **Consistency**: Unique identifiers must ensure no duplicate game or team names.
3. **Performance**: The system must provide fast response times for user interactions.
4. **Security**: User data must be protected across all platforms.
5. **Platform Compatibility**: The application must run smoothly on various operating systems (Windows, Mac, Linux) and mobile devices.

These constraints impact the development process by necessitating careful planning of the system architecture, choice of development tools, and strategies for data management and security.

## [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)

**UML Class Diagram Description**

The UML class diagram represents the structure of the game application. It shows the relationships between various classes and their attributes and methods. Here is a detailed description of each class and how they relate to each other:

1. **Entity Class**
   * **Attributes**: id (long), name (String)
   * **Methods**:
     + Entity(): Constructor
     + Entity(id: long, name: String): Constructor with parameters
     + getId(): long: Returns the ID of the entity
     + getName(): String: Returns the name of the entity
     + toString(): String: Returns a string representation of the entity
   * **Description**: The Entity class is a base class that contains common attributes (id and name) and methods for all entities in the game. It provides a common structure that can be inherited by other classes.
2. **GameService Class**
   * **Attributes**:
     + games (List<Game>)
     + nextGameId (long)
     + nextPlayerId (long)
     + nextTeamId (long)
     + service (GameService)
   * **Methods**:
     + GameService(): Private constructor to implement singleton pattern
     + getInstance(): GameService: Returns the single instance of GameService
     + addGame(name: String): Game: Adds a new game
     + getGame(id: long): Game: Retrieves a game by its ID
     + getGame(name: String): Game: Retrieves a game by its name
     + getGameCount(): int: Returns the number of games
     + getNextPlayerId(): long: Returns the next player ID
     + getNextTeamId(): long: Returns the next team ID
   * **Description**: The GameService class is a singleton that manages all game instances. It ensures only one instance of the service exists and provides methods to add and retrieve games.
3. **Game Class**
   * **Attributes**:
     + teams (List<Team>)
   * **Methods**:
     + Game(id: long, name: String): Constructor with parameters
     + addTeam(name: String): Team: Adds a new team
     + toString(): String: Returns a string representation of the game
   * **Description**: The Game class represents a game with a unique ID and name. It contains a list of teams that participate in the game and provides methods to add teams and get a string representation of the game.
4. **Team Class**
   * **Attributes**:
     + players (List<Player>)
   * **Methods**:
     + Team(id: long, name: String): Constructor with parameters
     + addPlayer(name: String): Player: Adds a new player
     + toString(): String: Returns a string representation of the team
   * **Description**: The Team class represents a team within a game. It has a unique ID and name, and it contains a list of players. It provides methods to add players and get a string representation of the team.
5. **Player Class**
   * **Attributes**:
     + id (long)
     + name (String)
   * **Methods**:
     + Player(id: long, name: String): Constructor with parameters
     + toString(): String: Returns a string representation of the player
   * **Description**: The Player class represents a player within a team. Each player has a unique ID and name, and it provides a method to get a string representation of the player.
6. **ProgramDriver Class**
   * **Methods**:
     + main(): The main method to start the application
   * **Description**: The ProgramDriver class contains the main method that initializes the game application and tests its functionality.
7. **SingletonTester Class**
   * **Methods**:
     + testSingleton(): Tests the singleton behavior of the GameService class
   * **Description**: The SingletonTester class tests the singleton behavior to ensure that only one instance of GameService exists.

**Object-Oriented Programming Principles**

1. **Encapsulation**: Each class has its own attributes and methods, which are hidden from other classes. This helps in managing the complexity by keeping the internal workings of each class private.
2. **Inheritance**: The Entity class is a base class, and Game, Team, and Player classes inherit common attributes (id and name) and methods from it. This promotes code reuse and a hierarchical class structure.
3. **Polymorphism**: The toString() method is overridden in Game, Team, and Player classes to provide specific string representations for each class. This allows objects to be treated as instances of their base class Entity while still using their specific implementations.
4. **Singleton Pattern**: The GameService class uses the singleton pattern to ensure that only one instance of the game service exists. This is crucial for managing the game state consistently across the application.

By using these principles, the software design ensures that the game application is modular, easy to maintain, and scalable. The relationships between classes are clearly defined, and common functionalities are centralized, making the codebase efficient and organized.

**"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 servers are known for their reliability and security, but they are expensive and not commonly used for web hosting. They are typically used for specific applications that require high performance and security. | Linux servers are very reliable, cost-effective, and widely used in web hosting. They offer excellent performance, scalability, and security, making them a popular choice for web-based applications. | Windows servers are easy to use and integrate well with other Microsoft products. They offer good performance and user-friendly management tools, but they can be more expensive and less secure than Linux. | Mobile devices are not typically used for hosting web applications. They are primarily used to run the client-side of applications, allowing users to interact with the game. Hosting on mobile devices is impractical due to limited resources and scalability. |
| **Client Side** | Developing for Mac requires specific expertise in macOS, which can be time-consuming and costly. However, Macs are popular among creative professionals and offer a consistent user experience. | Developing for Linux clients can be cost-effective and requires knowledge of different distributions. Linux offers flexibility and is preferred for environments that need customization. | Developing for Windows is straightforward, with many tools and resources available. It is the most widely used desktop operating system, which makes it essential to support. Development costs are moderate, and there is a large pool of expertise available. | Developing for mobile devices requires knowledge of both iOS and Android platforms. It can be expensive and time-consuming due to the need to create and maintain apps for multiple platforms. However, mobile apps are crucial for reaching a broad audience. |
| **Development Tools** | Tools like Xcode and Swift are used for Mac development. Xcode is a powerful IDE that supports all phases of development for macOS and iOS applications, but it requires a Mac to run. | Tools like GCC, Make, and various IDEs (e.g., Eclipse, IntelliJ IDEA) are used for Linux development. These tools are often free and open-source, providing a cost-effective development environment. | Tools like Visual Studio and .NET are commonly used for Windows development. Visual Studio is a comprehensive IDE that supports a wide range of programming languages and project types. | Tools like Android Studio for Android and Xcode for iOS are used for mobile development. These tools are platform-specific and provide the necessary features to build, test, and deploy mobile applications. |

## 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**: Use Linux for the main operating platform because it is cost-effective, reliable, and widely used for web hosting.
2. **Operating Systems Architectures**: Linux has a strong architecture that supports high availability, scalability, and security, making it suitable for web-based applications.
3. **Storage Management**: Use cloud storage services like AWS S3 or Google Cloud Storage. These services are reliable, scalable, and provide easy integration with web applications.
4. **Memory Management**: Linux uses effective methods to manage memory, such as virtual memory and paging, ensuring that the game runs smoothly even with many users.
5. **Distributed Systems and Networks**: Use a setup where different parts of the game can communicate over the internet. RESTful APIs can allow various devices to interact with the game, ensuring seamless connectivity and handling any network issues.
6. **Security**: Protect user information by using encryption (like SSL/TLS) for data transfer and secure login methods (like OAuth). Regularly check and update security measures to keep the game safe.