Draw It or Lose It

CS 230 Project Software Design Template

Version 1.0

Table of Contents

Document Revision History

Version	Date	Author	Comments
1.0	01/23/2026	Jessica Soler	Executive Summary, Requirements
1.1	01/25/2026	Jessica Soler	Design Constraints, Domain Model
1.2	02/05/2026	Jessica Soler	Evaluation
1.3	02/16/2026	Jessica Soler	Recommendations

Executive Summary

The client, The Gaming Room, wants to expand its existing Android-based game into a web-based application that serves multiple platforms. Unsure of how to set up the environment, the client has requested guidance on designing a solution that meets their technical and business requirements. To address these needs, the goal of this project is to design a solution that manages multiple games, teams, and players while enforcing unique identifiers and names to prevent duplication and data corruption.

A centralized game service is implemented using the Singleton design pattern to ensure only one instance of the game service exists. This service is responsible for creating and managing game instances and assigning unique identifiers to each game, team, and player. Separate subclasses for games, teams, and players inherit attributes from a shared base class and allow each object to store its own ID and name. These objects are stored in lists, and the iterator design pattern is used to verify name uniqueness before creating a new instance.

Requirements

Business

- *Scale the current Android application into a web-based, multiplatform application to reach a broader audience.*
- *Support multiple games that run independently and contain one or more teams.*
- *Each team allows multiple players.*
- *Ensure game names and team names are unique, enabling players to create a new game or join a previous game and prevent duplicate objects and conflicting data.*

Technical

- *Centralized game service that ensures only one instance of the game service exists.*
- *Assign unique identifiers to every game, team, and player to prevent duplicating data.*
- *Store lists of current games, teams, and players using dynamic data structures.*
- *Enforce unique names for games, teams, and players by checking current lists before creating a new object.*

Design Constraints

- **Web-Based, Multiplatform Constraint**
  - *The application must run in a web-based environment and support both mobile and desktop platforms. It must also support Apple, Windows, and Android operating systems. This constrains the design to use platform independent technology. This also adds constraints that allow users to log in from different devices and access the same account.*
- **Singleton Instance Constraint**
  - *Only one instance of the GameService can exist at one time which adds a design constraint. The Singleton pattern is used so that the game creation and ID assignment are handled within a single GameService object. Objects must be created through this service to avoid duplicate games, IDs, and lists.*

- **Unique Identifier and Name constraints**
    - *Each game, team, and player must have a unique name and ID. This constrains the design to include ID counters and verify names are unique before creating a new object using the iteration design pattern.*
- **Data Management Constraint**
    - *The system must store and manage collections of data (games, teams, players) using data structures that are mutable. This constrains the iteration design to check each object name and ID in the collection rather than a reference by ID only.*

System Architecture View

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.

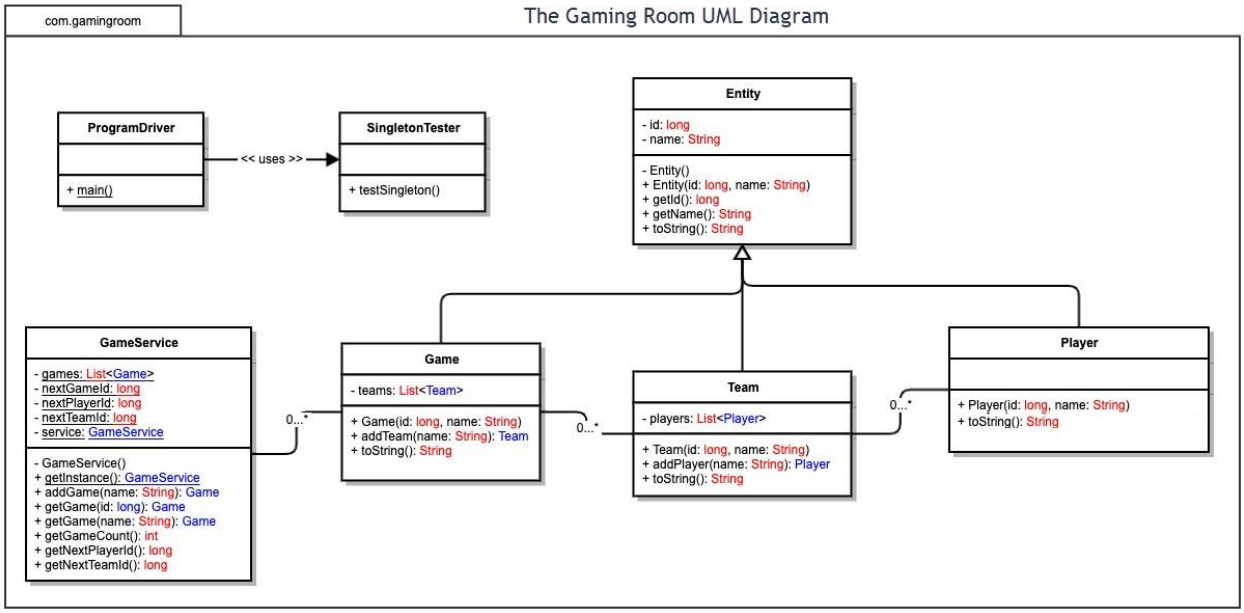
Domain Model

This UML illustrates an inheritance-based structure and a centralized service that manages games, teams, and players. The Entity superclass contains the common attributes id: long and name: String, along with constructors and getter methods. The Game, Team, and Player subclasses all inherit these attributes from Entity. This allows shared fields and behaviors for all subclasses to be defined once in the superclass. These attributes satisfy the requirement that every game, team, and player must have a unique identifier and name.

A Game contains a collection of Team objects using teams: List<Team>. This satisfies the game-to-team relationship requirement that states a game can have one or more teams. The Game subclass manages teams and includes a method to create or add teams into the game's list, represented in the UML as addTeam(name: String): Team.

A Team contains a collection of Player objects using players: List<Player>. This satisfies the team-to-player relationship requirement that each team will have one or more players. The Team subclass manages players and includes a method to add players into the team's list, represented in the UML as addPlayer(name: String): Player.

The GameService class manages the collection of games (games: List<Game>) and sums the next identifiers (nextGameId, nextTeamId, nextPlayerId). The singleton pattern requirement is represented in the UML as service: GameService and accessed through getInstance(): GameService. This satisfies the requirement that only one instance of the game service can exist in memory. This ensures IDs remain unique and that games are created and retrieved from one source. The addGame and getGame methods in GameService are designed to search the existing list before adding or returning an object to ensure data reliability.


com.gamingroom
The Gaming Room UML Diagram
ProgramDriver
+ main()
<< uses >>
SingletonTester
+ testSingleton()
Entity
- id: long
- name: String
- Entity()
+ Entity(id: long, name: String)
+ getId(): long
+ getName(): String
+ toString(): String
GameService
- games: List<Game>
- nextGameId: long
- nextPlayerId: long
- nextTeamId: long
- service: GameService
- GameService()
+ getInstance(): GameService
+ addGame(name: String): Game
+ getGame(id: long): Game
+ getGame(name: String): Game
+ getGameCount(): int
+ getNextPlayerId(): long
+ getNextTeamId(): long
0...*
Game
- teams: List<Team>
+ Game(id: long, name: String)
+ addTeam(name: String): Team
+ toString(): String
0...*
Team
- players: List<Player>
+ Team(id: long, name: String)
+ addPlayer(name: String): Player
+ toString(): String
0...*
Player
+ Player(id: long, name: String)
+ toString(): String

Evaluation

Development Requirements	Mac	Linux	Windows	Mobile Devices
Server Side	macOS can host a web-based app and supports Java and Unix-based tools, making it suitable for development and testing. However, higher hardware costs and limited scalability make it less ideal for long-term hosting.	Linux is a very popular platform for web servers as it is stable, secure, and low cost. It integrates well with Java app servers and cloud environments, making it ideal for hosting a distributed web-based game.	Windows Server supports web apps and integrates well with Microsoft tools but has higher licensing costs. It is less commonly used for Java based web hosting than Linux.	Mobile devices are not suitable for server-side hosting due to limited resources. They are intended to function only as clients.
Client Side	macOS clients are well supported through current browsers, which require minimal platform specific development. Testing requires Apple hardware which increases costs.	Linux clients vary by distribution, which would require additional testing. However, the app is able to run with minimal platform-specific changes.	Windows clients represent the largest user base and are well supported by current browsers.	Mobile clients require responsive design and would require additional testing for touch input and the many different screen sizes. Development effort is increased, but necessary for expanded accessibility.
Development Tools	macOS supports Java, build tools like Maven and Gradle, and IDEs like IntelliJ IDEA and Eclipse. Built in Unix tools make development efficient.	Linux provides strong support for Java, open-source build tools, containers, and cloud deployment. It is well suited for both development and production environments.	Windows supports Java development using IDEs such as IntelliJ IDE, Eclipse, and Visual Studio Code.	Mobile development relies on web technologies and testing tools such as browser dev tools and emulators. Tooling focuses on client optimization rather than app hosting.

Recommendations

1. ***Operating Platform****: The recommended operating platform for hosting the Draw It or Lose It web-based application is Linux. Linux is widely used for web servers because it is stable, secure, and low cost. It works well with Java-based server applications and is supported by cloud hosting providers. It is a strong fit for a distributed application that must serve multiple platforms and operating systems.*

2. ***Operating Systems Architectures****: Linux supports the most common server processor architectures, allowing the application to run on a wide range of hardware and cloud platforms. This allows the application to use system resources efficiently and handle multiple users at the same time.*

3. ***Storage Management****: A relational database system is recommended for storing game, team, and player data. MySQL or PostgreSQL allows data to be organized, searched, and protected using constraints like ID and unique name. Using a database ensures data is not lost if the server goes down.*

4. ***Memory Management****: Linux uses virtual memory management. This helps applications run smoothly by managing how memory is shared between running processes.* This allows the operating system to dynamically manage how physical memory is shared between running processes, ensuring applications run smoothly even under load. By relying on Linux's virtual memory capabilities, the *Draw It or Lose It* game server will be able to seamlessly handle multiple simultaneous client requests and rapid image rendering without consuming excessive hardware resources.

5. ***Distributed Systems and Networks****: The game should use a distributed web-based design in which the server handles game logic and client devices connect to the network. This allows game data on the server to keep information consistent between devices.*

   Dependency: *By keeping the game data on the server, information remains consistent across all connected devices. However, this creates a strict dependency on network connectivity. If the network goes down or the server experiences an outage, client devices will not be able to access the game.*

6. ***Security****: Linux provides strong built in security features like user permissions and process isolation. The application should protect data sent between clients and the server using secure network connections. User information should be limited to what is necessary, and input should be validated, allowing user data to be safe across all platforms.*