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	10/17/2024	Tanisha Wilson	Final draft of the design document.

Executive Summary

The Gaming Room wants to broaden its game, Draw It or Lose It, from Android to a web-based application that works on desktops and mobile devices. This change brings some challenges, like managing many teams and players and making certain each game instance has a unique identifier. To solve these issues, we plan to create a web application that can support many users at once. We'll use a RESTful API to handle game interactions and user logins. It's important that each game instance is unique, allowing only one game to run at a time to keep everything organized. We'll also implement security measures, such as Basic Authentication, to protect user information. This approach will help us build a scalable and maintainable application that can grow with future updates. In summary, we need to focus on these software design elements to ensure the game meets user needs and runs smoothly on various platforms.

Requirements

The Gaming Room has identified several important business and technical requirements for the web-based version of Draw It or Lose It. First, the application needs to support multiple teams, each consisting of several players, to allow for interactive gameplay. It is essential that each game instance, team, and player has a unique identifier to prevent name conflicts and maintain clarity during the game. Additionally, only one instance of the game can be active at any given time, which is crucial for avoiding conflicts and making certain for a smooth gaming experience. The application should be designed as a web-based solution, making it usable across numerous platforms. A RESTful API must be implemented to facilitate communication between the client and server, enabling easy interactions during gameplay. Furthermore, Basic Authentication should be integrated to protect user credentials and data, making sure everything is secure. Lastly, the design should be scalable, allowing for future enhancements and new features without the need for a complete system overhaul. These requirements will shape the software design and development process, making sure that the final product meets the client’s needs and delivers a robust gaming experience.

Design Constraints

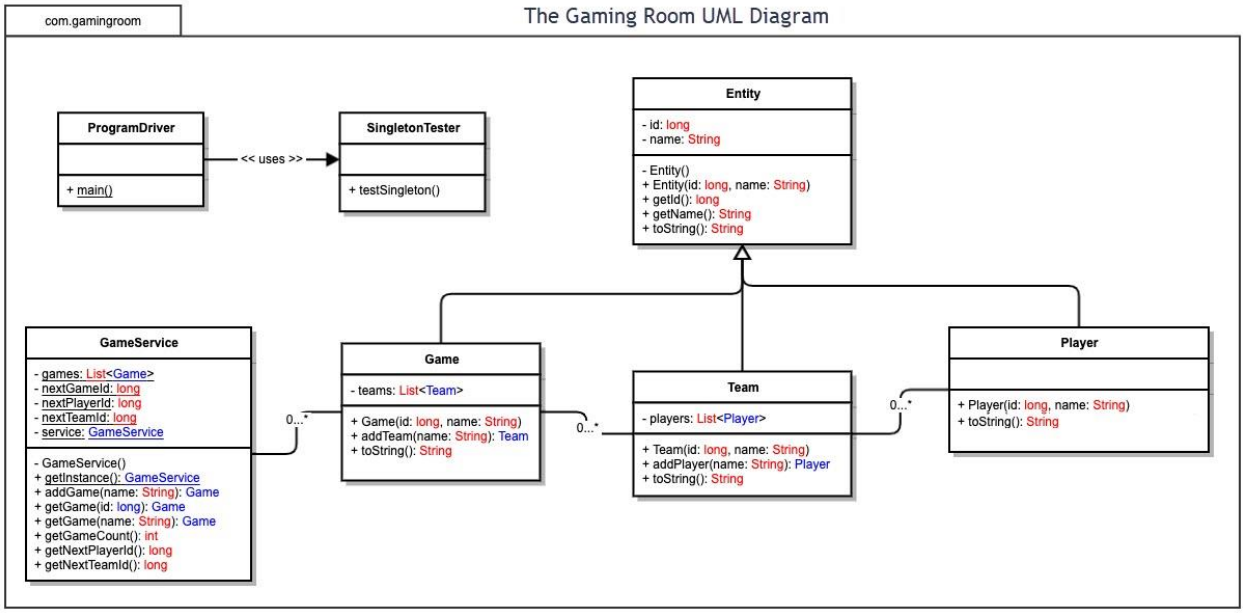
When creating the game application for a web-based distributed environment, several key design constraints must be addressed. First, performance is critical. The app needs to support multiple users simultaneously without lag, requiring careful resource planning and efficient coding to handle growth. Poor performance could frustrate users and lead to attrition. Second, security is essential, as the application will handle sensitive user data. Implementing measures like data encryption and secure authentication adds complexity to development but is necessary to protect user trust. Cross-platform compatibility is another constraint. The application must function well across various devices and browsers, which necessitates thorough testing and potentially more development time to ensure a consistent user interface. Network reliability is also vital. The app relies on internet connectivity for client-server communication, so it must handle interruptions gracefully to maintain user enjoyment. Finally, maintenance and scalability are important considerations. The architecture should facilitate easy updates and new features without requiring a complete redesign, emphasizing the need for a modular structure.

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

The UML class diagram application outlines the structure of the game's components and their relationships. The diagram begins with the ProgramDriver, which contains the main method and interacts with the SingletonTester to check the singleton pattern used in the application. The "uses" arrow indicates that ProgramDriver relies on SingletonTester for its functionality. Next, the GameService class acts as the central management component for the game, holding lists of Game, Team, and Player objects. It keeps track of unique identifiers like nextGamesId, nextPlayerId, and nextTeamId to ensure each instance is distinct. Key methods like getInstance(), addGame(), and getGame() are crucial for managing game instances and making certain only one instance exists in memory at any time. The Game class maintains a list of Team objects and is linked to GameService, indicating that it can retrieve or modify games through this service. The addTeam() method allows teams to be added to the game, reinforcing the connection between a game and its teams. Each Team contains a list of Player objects, and the relationship is zero-to-many, meaning a game can have multiple teams. The addPlayer() method allows for the addition of players to the team, supporting the dynamic nature of gameplay.
The Player class represents individual players and inherits common attributes from the Entity class, which contains basic properties like id and name. This ensures that every player can be uniquely identified. The Entity class serves as a base for Game, Team, and Player, encapsulating shared properties and promoting code reuse across the different game components.
In terms of object-oriented programming principles, the diagram showcases several key concepts. Encapsulation is evident as classes like Game, Team, and Player protect their data and methods. Inheritance is utilized, allowing Game, Team, and Player to share attributes and methods from the Entity class, cutting down on code duplication. The singleton pattern makes certain only one instance exists, maintaining a consistent game state. Composition is reflected in how classes relate to each other, with Game containing Team and Team containing Player. Lastly, association is shown through multiplicity notation, indicating how many instances can be associated, which is essential for supporting multi-user gameplay.


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 run server software but is mainly designed for desktop use. It has a user-friendly interface, strong security features, and good development tools, making it suitable for small teams. However, it's not as common for large-scale web hosting, limiting its scalability.	Linux is popular for server environments due to its open-source nature and configurability. It offers excellent performance, scalability, and security. However, it has a steeper learning curve and requires more expertise for setup and maintenance.	Windows supports various server configurations, including Internet Information Services (IIS), and integrates well with Microsoft products, making it familiar for many developers. While it provides good support for .NET applications, licensing costs can be high, and it may be less stable under heavy loads compared to Linux.	Mobile supports various server configurations, including Internet Information Services (IIS), and integrates well with Microsoft products, making it familiar for many developers. While it provides good support for .NET applications, licensing costs can be high, and it may be less stable under heavy loads compared to Linux.
Client Side	Development for Mac can be costly due to hardware and licensing, with longer development times required for different OS versions. Developers need to be familiar with frameworks like Swift and Objective-C.	Linux generally has lower development costs, and deployment can be quicker with common frameworks, although familiarity with command-line tools is helpful.	Windows can incur high licensing fees, but development can be faster with tools like Visual Studio, requiring knowledge of .NET.	Mobile devices costs include app store fees, and expenses can vary. Using cross-platform frameworks can speed up development, but developers must know mobile languages like Swift for iOS and Kotlin for Android.
Development Tools	Mac uses Swift and Objective-C with Xcode for iOS and HTML/CSS/JavaScript for web apps	Linux supports Python, Node.js, Java, and PHP with frameworks like Django and Express.	Windows typically involves C#, ASP.NET, and Java, using frameworks like Angular and React, with IDEs such as Visual Studio and JetBrains Rider.	Mobile development utilizes Swift, Kotlin/Java, and HTML/CSS/JavaScript, with tools like Android Studio and Xcode.

Recommendations

Analyze the characteristics of techniques specific to various systems architectures and recommend to The Gaming Room. Specifically, address the following:

1. **Operating Platform**: To expand " Draw It or Lose It," I recommend using Linux as the operating platform. It's recognized for its stability, security, and scalability. Which makes it an ideal choice for a web-based application that needs to support a large number of users on different devices. Additionally, since it is open source, we can customize and optimize it to meet the specific needs of the project.

2. **Operating Systems Architectures**: Linux offers several architectures, but I suggest going with a microservices architecture. This allows different parts of the application like user authentication, game logic, and data management to function independently, which improves maintainability and scalability. Using Docker for containerization will help ensure consistent deployment across both development and production environments, simplifying dependency management and updates.

3. **Storage Management**: For storage management, I propose using PostgreSQL as the relational database management system. It can handle complex queries and maintain transactional integrity, which is essential for a game that requires reliable data management.

4. **Memory Management**: Linux utilizes dynamic memory allocation, which is helpful for "Draw It or Lose It," as the number of active players and teams can change frequently. This approach allows the application to request and free up memory as needed, optimizing resource usage. Also, Linux's memory overcommitment feature helps manage memory efficiently, allowing the application to handle varying loads without running into allocation issues.

5. **Distributed Systems and Networks**: To allow communication between different platforms application should make use of RESTful APIs, which allow for stateless interactions and effectively manage requests and responses across devices.

6. **Security**: Security is crucial for this project. To safeguard user information, I recommend using Basic Authentication along with HTTPS to secure communications between the client and server. Data encryption should be applied to sensitive information both when stored (in the database) and when transmitted (during API requests).