Draw It or Lose It

**CS 230 Project Software Design Template**

Version 2.0

**Table of Contents**

**Document Revision History**

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 9/24 | Thomas C |  |
| 2.0 | 10/24 | Thomas C |  |
| 3.0 | 10/24 | Thomas C |  |

**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**

The Gaming Room is trying to expand on their Android-based game, Draw It or Lose It, to build it as a web-based platform accessible across multiple devices. This transition will require significant changes to the architecture and design of the game application to support a distributed, web-based environment while ensuring key features are retained. The new system must support multiple teams, each with unique names and multiple players, with safeguards to ensure no duplication of game or team names. Additionally, only one instance of a game will exist in memory at a time, necessitating the implementation of a Singleton design pattern. This document outlines the proposed solution, addressing the client’s software requirements and design constraints, with a focus on maintaining performance, scalability, and security.

**Requirements**

*<* Please note: While this section is not being assessed, it will support your outline of the design constraints below. *In your summary, identify each of the client’s business and technical requirements in a clear and concise manner.>*

**Design Constraints**

Singleton Pattern for Game Management

As the client requires that only one instance of the game be running at a time, a Singleton design pattern will be implemented. This constraint ensures that the game instance can be accessed globally but restricts the system to a single active game, reducing memory usage but introducing challenges in managing concurrent user requests.

Unique Naming for Games and Teams

Since both game and team names must be unique, a strategy for real-time name validation will be implemented. This will involve checking the database or in-memory storage whenever a user selects a name to avoid conflicts. However, the need for constant name checking can introduce performance overhead, especially with a large number of users.

Team and Player Management

Since each team can consist of multiple players, efficient data structures will be needed to manage players within teams and track their activities. This adds an additional layer of complexity, especially when handling a large number of concurrent teams and players in different games.

Web Application Scalability

The system needs to be designed to scale to accommodate a potentially large number of users without impacting game performance or user experience.

Security Considerations

Given the web-based nature, ensuring secure communication between clients and the server (such as implementing SSL/TLS) and preventing unauthorized access to game sessions and player information is critical. This adds to development time and resource allocation for security testing.

**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 diagram represents the domain model for the game application. It consists of several key classes: Entity, Game, Team, Player, GameService, ProgramDriver, and SingletonTester.

**Class Relationships**

Entity (Base Class)

- Acts as a superclass for Game, Team, and Player.

- Contains common attributes (id, name) and methods shared by its subclasses.

Game

- Inherits from Entity.

- Has a composition relationship with Team (teams: List<Team>).

- Manages multiple teams within a single game instance.

Team

- Inherits from Entity.

- Has a composition relationship with Player (players: List<Player>).

- Manages multiple players within a team.

Player

- Inherits from Entity.

- Represents individual players in the game.

GameService

- Central class managing game instances, teams, and players.

- Implements the Singleton pattern (getInstance() method).

- Maintains lists of games and handles unique ID generation for games, teams, and players.

ProgramDriver

- Contains the main() method to run the application.

- Uses GameService to manage game instances.

SingletonTester

- Utility class to test the Singleton implementation of GameService.

**Object-Oriented Programming Principles Demonstrated**

Inheritance

- The Entity class serves as a base class for Game, Team, and Player.

- This demonstrates the principle of inheritance, allowing common attributes (id, name) and methods to be shared among subclasses.

- Fulfills the requirement of unique identifiers for game instances, teams, and players efficiently.

Encapsulation

- Classes encapsulate their data (e.g., private attributes) and provide public methods for interaction.

- For example, GameService encapsulates the game management logic, providing controlled access through methods like addGame() and getGame().

Abstraction

- The Entity class abstracts common properties and behaviors, simplifying the overall design.

- GameService abstracts the complexities of game management, providing a simple interface for other parts of the system.

Polymorphism

- While not explicitly shown, the use of a common base class (Entity) allows for polymorphic behavior in handling games, teams, and players.

**Evaluation**

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** | - Advantages: Stable Unix-based environment, good security, reliable for development.  - Weaknesses: Higher cost of hardware, limited in enterprise-scale server hosting.  - Suitability: Not commonly used for large-scale hosting but good for local development/testing. | - Advantages: Open-source, highly customizable, cost-effective, popular for web hosting (e.g., LAMP stack).  - Weaknesses: Requires more expertise to manage.  - Suitability: Excellent for hosting due to flexibility, security, and widespread server use (e.g., Ubuntu, CentOS). | - Advantages: Familiar environment for many developers, good integration with Microsoft Azure and enterprise solutions.  - Weaknesses: Licensing costs, security issues, more resource-heavy.  - Suitability: Suitable for hosting with Windows Server, but higher cost and complexity than Linux. | - Advantages: Can host smaller-scale mobile versions using platforms like Firebase, AWS.  - Weaknesses: Limited performance for server-side hosting, more focused on client-side mobile apps.  - Suitability: Not ideal for primary hosting, but cloud services like AWS can help integrate mobile access. |
| **Client Side** | Considerations: High development cost, fewer games developed for Mac, but users expect a polished experience. Requires expertise in macOS-specific features.  - Time: Moderate to high (due to fewer tools). | Considerations: Widely used open-source system; flexible but requires more time and expertise due to varied distributions and compatibility issues.  - Time: Medium, depends on distribution compatibility and testing. | - Considerations: High user base, strong support from Microsoft development ecosystem, easy to develop and test.  - Time: Moderate; Windows is a popular gaming platform with solid tools and documentation. | - Considerations: Most development here is focused on mobile platforms (iOS and Android). Requires expertise in mobile frameworks.  - Time: High for cross-platform mobile games. Lower if using cross-platform frameworks like React Native or Flutter. |
| **Development Tools** | Languages/Tools: Xcode for macOS/iOS development, Swift, Objective-C, web tools (HTML5, JavaScript, etc.). | Languages/Tools: Common languages (Python, PHP, JavaScript). Tools like Visual Studio Code, PyCharm, LAMP stack. | Languages/Tools: Visual Studio, .NET, C#, JavaScript frameworks, Azure tools. | Languages/Tools: React Native, Flutter for cross-platform mobile apps. Android Studio for Android, Xcode for iOS. |

**Recommendations**

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

* **Operating Platform**: I recommend a cloud-based platform utilizing Linux servers for The Gaming Room's expansion of Draw It or Lose It. This choice offers scalability, cost-effectiveness, and broad compatibility across different client platforms.
* **Operating Systems Architectures**: The Linux-based cloud platform will likely use a microservices architecture, allowing different components of the application to be developed, deployed, and scaled independently. This architecture supports containerization technologies like Docker, enabling consistent deployment across development and production environments.
* **Storage Management**: For storage, I recommend a combination of relational and NoSQL databases. Use a relational database (e.g., PostgreSQL) for structured data like user accounts and game statistics, and a NoSQL database (e.g., MongoDB) for flexible, scalable storage of game state and user-generated content.
* **Memory Management:** The Linux platform will use virtual memory management, efficiently allocating physical memory. For the application level, implement caching mechanisms (e.g., Redis) to store frequently accessed data in memory, reducing database load and improving response times.
* **Distributed Systems and Networks**: Implement a RESTful API for communication between client applications and the server. Use WebSockets for real-time game state updates. Employ load balancers to distribute traffic across multiple server instances. Implement retry mechanisms and circuit breakers to handle network instabilities and service outages gracefully.
* **Security**: Implement HTTPS for all client-server communications. Use OAuth 2.0 for user authentication and JWT for session management. Employ rate limiting to prevent abuse. Regularly update all server software and dependencies. Implement input validation and sanitization to prevent injection attacks. Use encryption for sensitive data at rest and in transit. Regularly perform security audits and penetration testing.