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	09/21/24	Joseph Cassello Jr.	Initial draft of the software design document
2.0	10/04/2024	Joseph Cassello Jr.	Second draft, evaluation table revision

Executive Summary

The purpose of this document is to outline the design and development of a web-based version of "Draw It or Lose It," an interactive game currently available only on Android. This software will serve multiple platforms and involve multiple teams and players, enhancing user engagement and accessibility. Key features include unique team and game identifiers to prevent name collisions and ensuring only one game instance runs at a time. By implementing a robust software design and leveraging object-oriented principles, this document proposes an efficient solution that aligns with the client's requirement.

Requirements

- The game will support one or multiple teams
- Game and team names must be unique
- Only one game instance can exist in memory
- Each team may have multiple players.

Design Constraints

- Scalability: The application must handle latency and connection issues inherent in distributed systems. This requires careful planning around load balancing and resource allocation to ensure smooth performances as the user base grows.

- Network Reliability: The application must be designed to handle potential latency and connection problems. Implementing retry mechanisms and failover strategies will be crucial to maintaining a seamless user experience.
- Security: Sensitive user information must be protected through secure data transmission and storage methods. Regular security audits should also be considered to identify and mitigate vulnerabilities.
- Cross-Platform Compatibility: The solution must function seamlessly across different devices and browsers, complicating design decisions related to technology stacks and user interfaces. A responsive design and thorough testing on various platforms will be necessary to achieve this.

System Architecture View

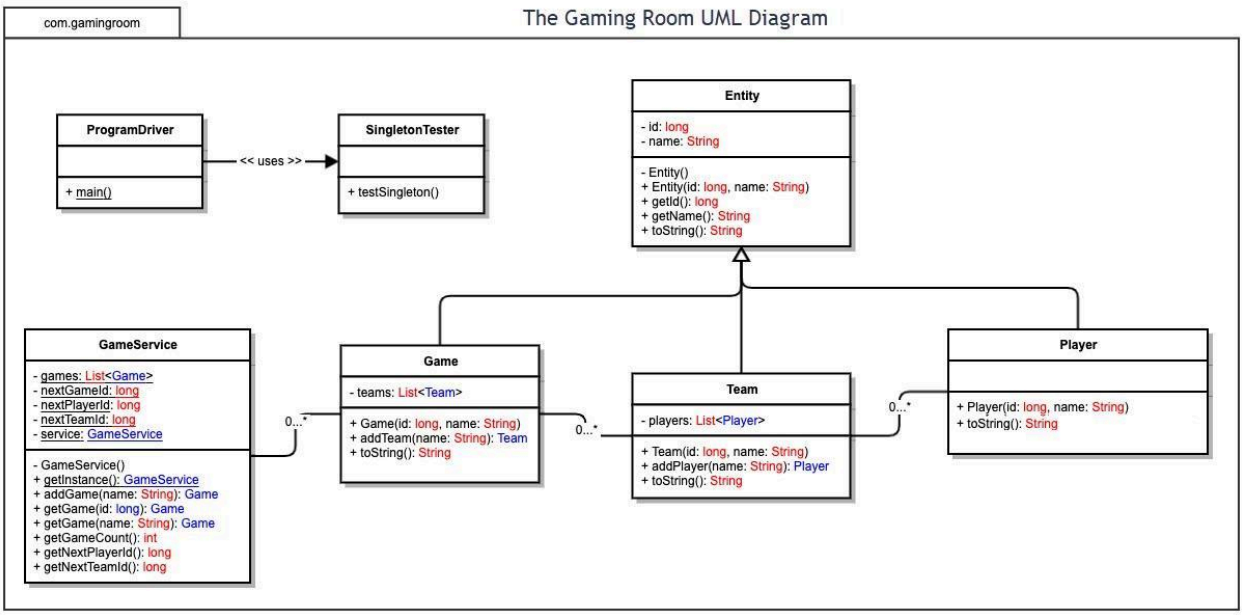
Please note: There is nothing required here for these projects, but this section serves as a reminder that describing the system and subsystem architecture present in the application, including physical components or tiers, may be required for other projects. A logical topology of the communication and storage aspects is also necessary to understand the overall architecture and should be provided.

Domain Model

The UML class diagram for the “Draw It or Lose It” application provides a structured representation of the game’s architecture. At the top level, the ProgramDriver class serves at the

entry point, utilizing the SingletonTester to enforce a single instance of the game, ensuring no conflicts arise from multiple instances running simultaneously. The entity class acts as a foundational base, encapsulating common attributes such as ID and name, which promotes code reusability across derived classes.

The GameService class plays a crucial role in managing game instances, teams, and players, while also ensuring the uniqueness of identifiers for each entity. The hierarchical relationships are clearly defined, where each Game contains multiple Teams, and each Team can have several Players. This structure leverages object-oriented principles, particularly encapsulation allowing Game and Team to inherit from the Entity class.



Evaluation

Development Requirements	Mac	Linux	Windows	Mobile Devices

Server Side	Stable, user-friendly, seamless integration with Apple services. macOS Server was discontinued, so there are no licensing fees. Limited scalability, not commonly used for high-performance web hosting.	Flexible, open-source, high performance, strong security. Commonly used for web hosting and supports high scalability. Free to use. Complex for beginners, requires expertise in system management.	Familiar environment, seamless integration with Microsoft services. Strong enterprise support, high scalability and built in security features. Datacenter 16-core license: $6,155 Standard 16-core license: $1,069 Essentials 10 cores and 1 VM license: $501.	Limited resources for hosting on mobile, high user engagement. No direct licensing fees; typically managed through cloud services (Prices can range from $50-$500 a month). Security risks, performance variability.

			Frequent updates; may require more maintenance.	
Client Side	Requires Swift or Objective-C expertise, seamless integration with Apple devices Higher tool costs (around $300-$800). Longer cross-platform development cycles.	Flexible but requires expertise, cost-effective, supports a wide range of browsers. Complexity may increase development time.	Familiar tools, large user base, seamless integration with Microsoft products, large developer community. High licensing fees for tools (Visual Studio can cost around $300-$1,200 per license)., Compatibility issues.	Expertise in Android and iOS, large mobile user base, responsive design. Higher costs for using cross-platform tools (Licenses for cross-platform tools can cost around $400-$1,200.)

Development Tools	Visual Studio Code, Ruby on Rails, Swift and Objective-C.	Python, Java, PHP, Vim, Emacs, and Docker.	C#, VB.NET, ASP.NET, Visual Studio, SQL Server, and Azure.	Android Studio, Xcode, React Native, and Flutter.

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: I recommend Linux as the primary operating platform for developing and hosting “Draw It or Lose It.” I chose Linux because it provides a cost-effective, flexible, environment with strong community support. It is well suited for web applications and can easily scale as the user base grows.

2. Operating Systems Architectures: The architecture will utilize a client-server model. On the server side, the game logic, data processing, and user management will reside on Linux servers. The client side will consist of web and mobile clients that interact with the server via RESTful APIs, allowing for seamless communication and data exchange across different platforms. This architecture supports high availability and load balancing, ensuring a smooth user experience.

3. Storage Management: I recommend implementing a NoSQL database, such as MongoDB or firebase. These databases efficiently handle unstructured data and scale easily as more teams and players join. They also support real-time data updates, which are crucial for a dynamic game environment.

4. Memory Management: Linux employs various memory management techniques, including paging and segmentation, to optimize the use of RAM. The game can utilize in-memory caching solutions like Redis to store frequently accessed data, minimizing latency and enhancing performance. Additionally, automatic garbage collection in languages like Java or Python can help manage memory without developer intervention, reducing memory leaks and enhancing stability.

5. Distributed Systems and Networks: To facilitate communication between clients and the server, a RESTful API will be utilized. The server will handle requests from multiple clients, using WebSocket for real-time features such as drawing and guessing interactions. It is essential to design for fault tolerance to address connectivity issues, implementing retries and fallbacks for network failures to maintain user experience. Load balancers can be employed to distribute requests among servers, reducing the impact of outages on any single server.

6. Security: To protect user information, HTTPS should be implemented for secure data transmission between clients and the server, safeguarding user data from eavesdropping. Additionally, utilizing OAuth 2.0 for user authentication and authorization ensures that

only authorized users can access sensitive game functionalities. Regular updates and patches for the Linux operating system and third-party libraries will help mitigate vulnerabilities. It is also crucial to encrypt sensitive user data both in transit and at rest, using strong encryption standards.