The Gaming Room

Project Software Design: Draw It or Lose It

Version 1.0

Table of Contents

Document Revision History

Version	Date	Author	Comments
1.0	03/21/2022	Lewis Quick	Updated Executive Summary, Design Constraints, Domain Modal, Evaluation, and Recommendations.

Executive Summary

The Gaming Room wants to expand their game Draw It or Lose It, which is currently available exclusively on Android, to a web-based platform that can operate across multiple platforms. The staff at The Gaming Room are unfamiliar with setting up an environment for a web-based application and need guidance on streamlining development. This document outlines the design constraints, system architecture, and recommendations for transitioning to a web-based distributed environment.

Design Constraints

The game Draw It or Lose It is currently developed for Android and needs to be adapted for a web-based distributed environment. The software requirements are:

1. The game must support multiple teams, each with multiple players.
2. Game and team names must be unique.
3. Only one instance of the game can exist in memory at any given time.

To meet these requirements, the singleton design pattern will be implemented to ensure that the game and team names remain unique and that only one instance of the game exists in memory.

System Architecture View

The system architecture for the web-based version of Draw It or Lose It will consist of the following components:

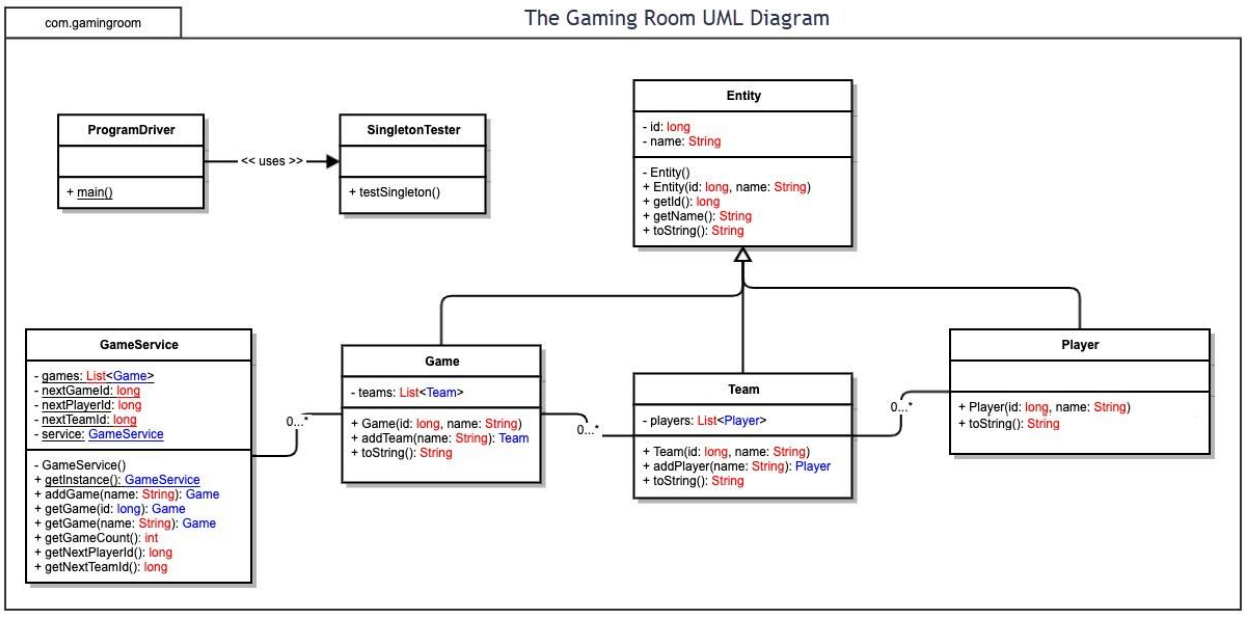
1. **Frontend**: Developed using HTML, CSS, and JavaScript frameworks like React or Angular to ensure responsiveness and user engagement across different devices.
2. **Backend**: Implemented with a robust framework such as Node.js or Django, providing RESTful APIs for client-server communication.

3. **Database**: A relational database like PostgreSQL or a NoSQL database like MongoDB will be used to store game data, ensuring scalability and reliability.
4. **Server Infrastructure**: Hosted on a cloud platform like AWS or Azure, providing scalability, load balancing, and high availability.
5. **Security**: Implementing SSL/TLS for data encryption, user authentication mechanisms, and regular security audits to ensure data integrity and privacy.

Domain Model

The UML diagram for The Gaming Room illustrates the implementation of the singleton design pattern, ensuring only one instance of the Game object exists at any given time. The diagram includes:

- **Entity Class**: A superclass with attributes and methods to manage unique IDs and names.
- **Game, Team, and Player Classes**: Inheriting from the Entity class, these subclasses manage the core elements of the game.
- **Associations**: Bidirectional associations between GameService and Game, Game and Team, and Team and Player, indicating how these objects interact.



Evaluation

Development Requirements	Mac	Linux	Windows	Mobile Devices
Server Side	macOS server configuration is simple but expensive.	Linux is cost-effective and stable.	Windows is user-friendly but prone to crashes.	Mobile OSs increase device functionality but are limited.
Client Side	Requires MAC devices, costly, and moderate expertise needed.	Cost-effective, requires significant expertise.	Minimal effort needed, widely used software development tools.	Extra time for formatting due to diverse hardware specs.
Development Tools	Uses Swift/Objective-C and Xcode.	Supports multiple languages and IDEs.	Supports nearly all languages and IDEs, dominant in C++.	Best with Visual Studio for cross-platform development.

Recommendations

1. **Operating Platform**: Recommend using Windows for its robust tools and software, simplifying the transition from Android to web-based.
2. **Operating Systems Architectures**: Windows OS has a layered architecture facilitating user applications and system resource access.
3. **Storage Management**: Use Windows Disk Management utility for advanced storage tasks.
4. **Memory Management**: Windows Memory Manager maps virtual addresses to physical addresses efficiently.
5. **Distributed Systems and Networks**: Utilize a shared database and network to enhance speed, functionality, and reliability.
6. **Security**: Windows OS includes comprehensive security applications to protect against threats, ensuring secure user information management.

Performance Considerations

To ensure the application performs optimally under various loads, the following strategies will be implemented:

1. **Load Balancing:** Distribute incoming network traffic across multiple servers using a load balancer (e.g., AWS ELB, Nginx).
2. **Caching:** Utilize caching mechanisms (e.g., Redis, Memcached) to store frequently accessed data and reduce database load.
3. **Optimization:** Optimize database queries, minimize API response times, and reduce client-side rendering time.
4. **Benchmarking:** Regular performance benchmarking using tools like JMeter to identify and mitigate bottlenecks.

Scalability Plan

To handle increasing load and ensure high availability, the application will employ the following scalability strategies:

1. **Horizontal Scaling:** Add more instances of application servers behind a load balancer to distribute traffic.
2. **Vertical Scaling:** Increase the capacity of existing servers by upgrading hardware resources.
3. **Containerization:** Use Docker containers to deploy applications, enabling easy scaling and orchestration using Kubernetes.
4. **Cloud Services:** Utilize cloud services such as AWS Auto Scaling and Azure Scale Sets to automatically adjust capacity based on demand.

Testing Strategy

Our testing strategy ensures the reliability, performance, and security of the application through the following methodologies:

1. **Unit Testing:**

- Scope: Test individual components or functions to ensure they work as intended.
- Tools: JUnit for Java, pytest for Python, and Jest for JavaScript.
- Process: Write test cases for each component and run them with each code commit.

2. **Integration Testing:**

- Scope: Test the interaction between integrated components to ensure they work together correctly.
- Tools: Postman for API testing, Selenium for web UI testing.
- Process: Develop integration test cases that cover various interaction scenarios, focusing on data flow and communication between modules.

3. **End-to-End (E2E) Testing:**

- Scope: Simulate real user scenarios to test the entire application workflow from start to finish.
- Tools: Cypress for web applications, Appium for mobile applications.
- Process: Create E2E test scripts that cover critical user paths and execute them in a staging environment.

4**. Performance Testing:**

- Scope: Evaluate the application's performance under load and stress conditions.
- Tools: Apache JMeter, Gatling.
- Process: Set up performance test scenarios, execute tests under different loads, and analyze response times, throughput, and resource utilization.

5. **Security Testing:**

- Scope: Identify and mitigate security vulnerabilities.
- Tools: OWASP ZAP, Burp Suite.
- Process: Conduct vulnerability assessments and penetration testing, focusing on common security issues such as SQL injection, XSS, and CSRF.

6**. Regression Testing:**

- Scope: Ensure that new code changes do not negatively impact existing functionality.
- Tools: Automated test suites with tools like Selenium, Jenkins.
- Process: Maintain a comprehensive suite of regression tests and run them after every major code change.

7**. User Acceptance Testing (UAT):**

- Scope: Validate the application against user requirements and expectations.
- Tools: Manual testing by endusers.
- Process: Prepare UAT scenarios, involve stakeholders in testing, gather feedback, and make necessary adjustments.