Draw It or Lose It

CS 230 Project Software Design Template

Version 1.2

Table of Contents

Document Revision History

Version	Date	Author	Comments
1.0	11.18.2024	Emmalie Cole	Initial draft of the design document
1.1	12.01.2024	Emmalie Cole	• Refined recommendations section to emphasize cross-platform compatibility and mobile considerations. • Added details about specialized teams for mobile development, responsive design, and platform security. • Enhanced Distributed Systems and Networks and Security sections for clarity and alignment with client goals.
1.2	12.12.2024	Emmalie Cole	Expanded on scalability measures, added example system architecture, and discussed trade-offs.

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 purpose of this project is to design and develop a web-based version of the game **"Draw It or Lose It,"** which is currently available as an Android application. The updated application will support multiple platforms while addressing key software requirements such as scalability, uniqueness of game and team names, and maintaining a single instance of the game in memory using the Singleton pattern. This system will leverage responsive web design to deliver a seamless experience across desktop and mobile platforms, ensuring users can access the game from any device

The application will consist of entities such as **Game**, **Team**, and **Player**, all inheriting from a newly created Entity class, which provides shared attributes (**id** and **name**). By leveraging industry-standard design patterns like Singleton and Iterator, the application ensures efficient memory management and unique identification of entities. These patterns, combined with best practices in software development, will provide a robust, maintainable, and scalable solution for the client.

Requirements

The following are the client's business and technical requirements for developing the web-based version of **"Draw It or Lose It"**:

1. **Business Requirements:**
    - The game must support multiple teams and players to ensure a collaborative and competitive gaming experience.
    - Team and game names must be unique to avoid confusion and maintain clarity for users.
    - The application must provide seamless platform independence, allowing users to play across various devices.

2. **Technical Requirements:**
    - Implement a centralized management system to maintain only one instance of the game at any given time, achieved using the Singleton pattern.
    - Ensure scalability by efficiently handling multiple players and teams without performance degradation.
    - Implement a mechanism (using the Iterator pattern) to enforce unique names for games, teams, and players.
    - Adopt secure communication and storage mechanisms to protect user data and support distributed architecture.
    - Develop a modular and maintainable system by adhering to object-oriented principles such as inheritance and encapsulation.

Design Constraints

Developing a web-based distributed application introduces several design constraints:

1. **Scalability:**
    - The system must handle multiple teams and players concurrently without performance degradation.

- Implication: Requires optimized algorithms and efficient data structures to manage entities.

2. **Platform Independence:**
   - The application must function seamlessly across different operating systems and devices.
   - Implication: Use of platform-neutral technologies like Java ensures compatibility.

3. **Security:**
   - User data and communication between platforms must be protected.
   - Implication: Implementation of encryption protocols and secure authentication mechanisms.

4. **Unique Identification:**
   - Names for games, teams, and players must be unique.
   - Implication: Requires validation mechanisms to prevent duplicate entries.

5. **Resource Management:**
   - Only one instance of the game should exist in memory at any given time.
   - Implication: Singleton pattern ensures efficient memory usage and centralized management.

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.

Here is an example of how the system and subsystem architecture for the **"Draw It or Lose It"** application might be structured:

System Architecture

The application will use a **three-tier architecture** to separate concerns and provide scalability:

1. **Presentation Tier (Client-Side):**
   - **Role:** Provides the user interface for players and teams.
   - **Components:**
     - Web browsers or mobile apps to interact with the game.
     - Communicates with the application server through RESTful APIs.
   - **Technologies:** HTML, CSS, JavaScript (front-end framework such as React for web-based clients).
2. **Application Tier (Server-Side Logic):**
   - **Role:** Handles game logic, player and team management, and API requests.
   - **Components:**
     - Java-based game application running on a web server.
     - Implements Singleton and Iterator patterns for efficient entity management.
   - **Technologies:** Java, Spring Framework (for web application development), Apache Tomcat.

3. **Data Tier (Storage and Persistence):**
    - **Role:** Stores data related to games, teams, players, and their states.
    - **Components:**
        - A relational database for persistent storage.
        - Implements data integrity constraints to ensure uniqueness of names.
    - **Technologies:** MySQL or PostgreSQL.

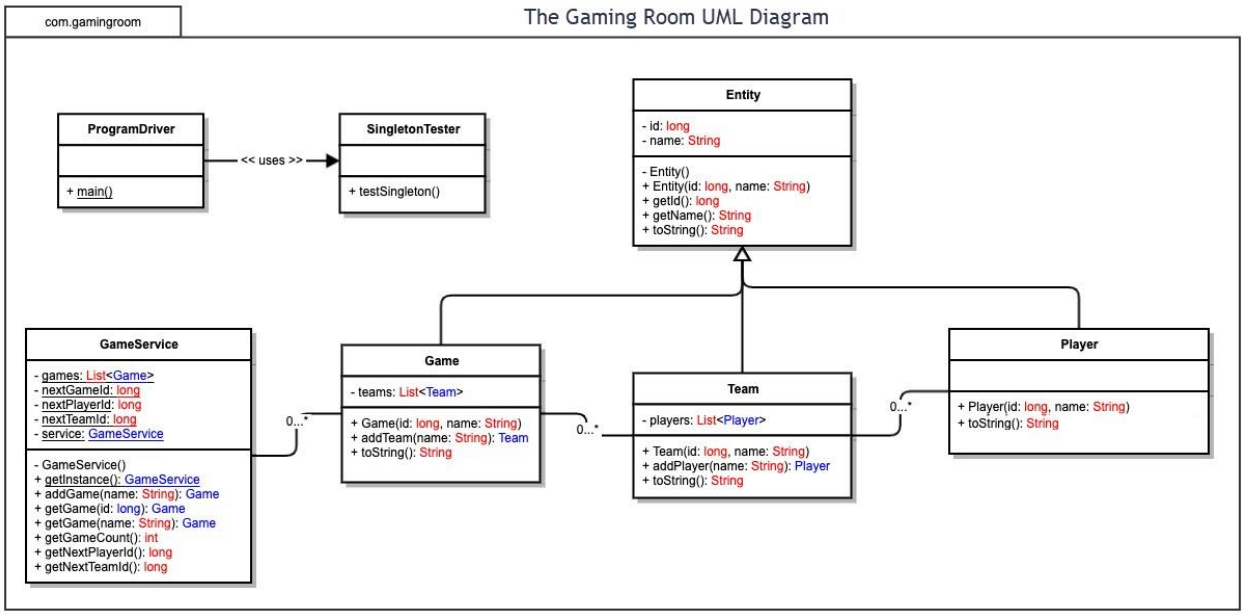
Logical Topology

To ensure scalability and reliability, the following logical components are used:

1. **Communication:**
    - RESTful APIs for client-server communication.
    - Secure HTTPS protocol to ensure encrypted data transfer.
2. **Storage:**
    - Centralized database for all game data.
    - Use of primary keys (id) and indexes for efficient queries.
3. **Network:**
    - Load balancers distribute traffic among multiple application servers.
    - Content delivery networks (CDNs) reduce latency for global users.
4. **Redundancy:**
    - Database replication ensures data availability and backup in case of failure.

Domain Model

The UML class diagram illustrates the structure of the game application and the relationships between the various components. It adheres to object-oriented programming (OOP) principles to efficiently fulfill software requirements.



1. **Class Relationships:**

    - **Entity Class:**

- Serves as a base class for Game, Team, and Player.
- Encapsulates common attributes (id and name) and behaviors (getId(), getName(), and toString() methods).

- **Game Class:**
  - Inherits from Entity and manages a list of teams (List<Team>).
  - Includes functionality to add a new team and ensure each team is uniquely identified.

- **Team Class:**
  - Inherits from Entity and manages a list of players (List<Player>).
  - Provides methods for adding a player and ensuring player names are unique within the team.

- **Player Class:**
  - Inherits from Entity and represents individual players in a team.

- **GameService Class:**
  - Implements the Singleton pattern to ensure only one instance exists in memory.
  - Manages the lifecycle of Game, Team, and Player objects, including the use of unique identifiers (nextGameId, nextTeamId, nextPlayerId) for all entities.
  - Provides methods like addGame(), addTeam(), and addPlayer() to efficiently manage these entities, utilizing the Iterator pattern to ensure uniqueness.

- **ProgramDriver Class:**
  - Acts as the main entry point for the application and interacts with GameService to test and demonstrate its functionality.

- **SingletonTester Class:**
  - Tests the Singleton pattern implementation for GameService to verify only one instance is created during runtime.

2. **OOP Principles:**

- **Inheritance:**
  - The Entity class acts as a parent for Game, Team, and Player, reducing redundancy and promoting reusability.

- **Encapsulation:**
  - Attributes like id and name are private, with public getter methods providing controlled access.

- **Abstraction:**
  - The Entity class abstracts common attributes and behaviors, simplifying the design.

- **Polymorphism:**

        - The toString() method is overridden in each subclass (Game, Team, and Player) to provide specific string representations of each entity.

    - **Singleton Pattern:**
        - The GameService class ensures centralized and consistent management of game entities across the application.

    - **Iterator Pattern:**
        - Used in GameService methods (addGame(), addTeam(), addPlayer()) to traverse and validate collections of entities.

3. **How the Design Fulfills Requirements:**

    - **Unique Names:**
        - The addGame(), addTeam(), and addPlayer() methods leverage the Iterator pattern to prevent duplicate names.

    - **Centralized Management:**
        - The Singleton pattern ensures a single GameService instance for managing all game-related operations.

    - **Scalability:**
        - The use of List collections for teams and players allows dynamic growth and efficient management.

    - **Code Maintainability:**
        - The modular design and adherence to OOP principles make the system easy to extend and maintain.

Evaluation

Development Requirements	Mac	Linux	Windows	Mobile Devices
Server Side	Mac provides reliable hosting performance but is expensive due to hardware and licensing costs. It is less commonly used in server environments compared to Linux and Windows. While it offers robust security and stability, it lacks the widespread adoption and tooling required for cost-efficient large-scale hosting.	Linux is the most widely used server platform for hosting due to its scalability, cost-effectiveness, and open-source nature. It supports high performance for web-based applications and includes robust security measures. Its flexibility and customizability are ideal for scaling up to thousands of players, though expertise in Linux administration is required..	Windows offers compatibility with enterprise tools and Microsoft ecosystems, making it a strong choice for businesses already using Microsoft products. However, its licensing costs and slightly lower scalability compared to Linux make it less cost-efficient for large-scale hosting. Windows servers are user-friendly but may require significant investment..	Mobile devices are unsuitable for hosting large-scale applications due to hardware and processing limitations. They can only support localized or peer-to-peer hosting scenarios, making them impractical for a web-based application serving thousands of users.

Client Side	Developing client-side applications for Mac requires tools like Xcode and expertise in Swift or Objective-C. Hardware costs are high, but macOS provides excellent stability, performance, and browser compatibility. Development times may be longer due to platform-specific requirements.	Linux is cost-effective for client-side development and offers robust tools like Eclipse, IntelliJ IDEA, and VS Code. However, developers must be proficient with Linux environments, and browser compatibility testing can be challenging due to the diversity of Linux distributions..	Windows offers a broad range of development tools such as Visual Studio, making it highly accessible for client-side development. Licensing costs are moderate, and compatibility with most web browsers ensures ease of deployment. Its wide adoption supports a large talent pool of developers, reducing development time.	Mobile platforms require native development tools like Android Studio (for Android) and Xcode (for iOS). While these tools are free, separate development teams may be needed for each platform, as Android uses Java/Kotlin and iOS uses Swift/Objective-C. Testing and deployment times may vary based on app store approval processes.).
Development Tools	Development on Mac involves tools like Xcode, IntelliJ IDEA, and Visual Studio Code. Licensing costs are tied to Apple hardware, which increases the overall expense. While macOS ensures seamless integration, it limits development to Apple devices, potentially restricting team flexibility.	Linux supports a wide range of free, open-source tools like Eclipse, IntelliJ IDEA, and NetBeans. These tools minimize costs, but the initial learning curve for Linux environments can slow down development. Linux also requires thorough cross-browser testing to ensure compatibility.	Windows leverages versatile tools like Visual Studio, IntelliJ IDEA, and Eclipse. While licensing costs can be significant, the tools are highly user-friendly, enabling rapid development and debugging. Windows is also widely adopted, simplifying team coordination and scaling development efforts.	Mobile development requires platform-specific tools: Android Studio for Android and Xcode for iOS. Both tools are free, but expertise in platform-specific programming languages is essential. This often necessitates separate development teams, increasing costs and timelines for cross-platform compatibility.

Recommendations

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

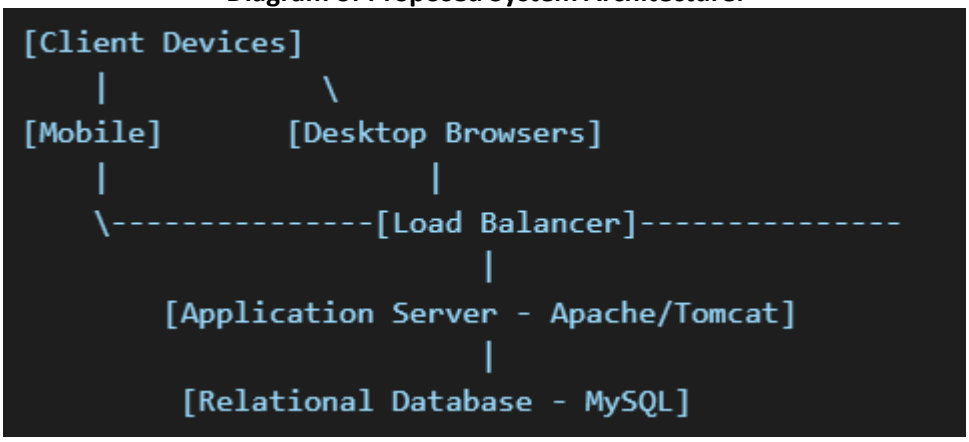
1. **Operating Platform**: The recommended operating platform for hosting the web-based version of **Draw It or Lose It** is **Linux**.

    a. **Reasoning:**
        i. Linux is cost-effective, open-source, and widely used for web applications.
        ii. It supports high scalability and reliability, which are essential for handling multiple teams and players.
        iii. Its flexibility and compatibility with most development tools make it an ideal choice for distributed applications.

2. **Operating Systems Architectures**: The Linux platform utilizes a **monolithic kernel architecture**, which integrates key services (e.g., memory, process, and file system management) into the kernel for optimized performance.

    a. It supports **modular extensions**, allowing customization for specific application needs.
    b. A common architecture for the application would involve:
        i. **Web Server Tier:** Hosting the application using technologies like Apache or Nginx.
        ii. **Application Layer:** Running the game logic implemented in Java.
        iii. **Database Layer:** A relational database such as MySQL for storing game, team, and player data.

Diagram of Proposed System Architecture:



3. **Storage Management**: **Recommended System:** MySQL or PostgreSQL.

    a. Both provide robust relational database management systems (RDBMS) that align with the requirements of **Draw It or Lose It**.
    b. **They ensure**:
        i. Data integrity through referential constraints.
        ii. Scalability to handle increasing amounts of player and team data.
        iii. Compatibility with Java's JDBC for seamless integration.
    c. **Features**:
        i. Supports transactional consistency (important for game state changes).
        ii. Easy backup and recovery options to safeguard data.

4. **Memory Management**: The Linux operating system employs **efficient memory management techniques**:

    a. **Virtual Memory:** Ensures efficient allocation of memory to processes, even if physical RAM is limited.
    b. **Caching and Buffering:** Speeds up data access by storing frequently accessed data in memory.
    c. **Garbage Collection:** In Java, the garbage collector automatically manages memory by reclaiming unused objects, reducing the risk of memory leaks.
    d. This combination ensures that **Draw It or Lose It** operates smoothly without interruptions due to memory bottlenecks.

5. **Distributed Systems and Networks**: To enable communication between platforms, the following architecture is recommended:

    a. **RESTful API:**
        i. Provides a lightweight, scalable interface for communication between devices and servers.
        ii. Supports JSON data, which is widely accepted across platforms and ensures compatibility with front-end frameworks like React and Angular for browser-based clients.
        iii. The proposed architecture leverages RESTful APIs for client-server communication, ensuring compatibility across web browsers (desktop) and mobile platforms. The system delivers seamless functionality across all platforms by adopting JSON data formats and supporting modern frameworks like React or Angular.
    b. **WebSocket Protocol:**
        i. For real-time updates (e.g., game progress), WebSockets enable low-latency, bidirectional communication.
    c. **Dependencies and Failures:**
        i. Employ **load balancers** to distribute traffic evenly across servers.
        ii. Implement **failover mechanisms** to redirect requests in case of server downtime.
        iii. Use **content delivery networks (CDNs)** to reduce latency for distributed players.
    d. **Specialized Teams:**
        i. For mobile platforms, development may require separate teams for Android and iOS to address platform-specific requirements. This may increase initial project costs but ensures optimal performance and user experience on both platforms.
        ii. Separate teams for Android and iOS development will ensure each platform's unique requirements are addressed. While this may increase upfront costs and require additional coordination, it ensures platform-optimized performance and a polished user experience, which is crucial for mobile users. Teams can collaborate on shared aspects such as API integration, which reduces redundancy and improves efficiency.
    e. **Scalability Measures:**
        i. Employ load balancers to distribute traffic evenly across servers.

ii. Implement failover mechanisms to redirect requests in case of server downtime.
iii. Use content delivery networks (CDNs) to reduce latency for distributed players.

6. **Security**: To protect user information and ensure secure communication between platforms:

    a. **Data Encryption:**
        i. Use HTTPS with SSL/TLS protocols to encrypt data in transit.
        ii. Additional measures will be implemented for mobile platforms, including the use of secure app sandboxing and adherence to platform-specific guidelines for iOS and Android to ensure user data remains protected.
    b. **Authentication and Authorization:**
        i. Implement OAuth or JWT for user authentication.
        ii. Role-based access control to restrict sensitive actions to authorized users.
    c. **Database Security:**
        i. Use parameterized queries to prevent SQL injection attacks.
        ii. Encrypt sensitive data (e.g., passwords, player data) at rest.
    d. **Firewalls and Monitoring:**
        i. Employ firewalls to block unauthorized access.
        ii. Use tools like Fail2Ban or Snort to monitor and prevent malicious activity.
        iii. These measures ensure robust security for users and the application.

7. **Trade-offs:**

- The use of Linux requires expertise in administration, which may lead to additional training costs.
- Separate development teams for Android and iOS may increase project costs but ensure platform-specific optimizations.

By adopting Linux as the operating platform, leveraging its architectural strengths, implementing relational storage management, and focusing on distributed systems, client-side compatibility, and robust security, ***Draw It or Lose It!*** can successfully expand to support multiple computing environments while maintaining scalability, reliability, cost-effectiveness, and security. Furthermore, by adopting responsive design and prioritizing cross-platform testing, the application ensures a consistent user experience across all devices.