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	07/16/24	Derricko Swink	We will be developing a web-based, multi-platform version of the game "Draw It or Lose It" using software design patterns to ensure efficient and unique management of game instances, teams, and players.

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 has requested the services of Creative Technology Solutions (CTS) to develop a web-based, multi-platform version of their existing Android game, "Draw It or Lose It."

The game, similar to the classic television game "Win, Lose or Draw," involves teams competing to guess images rendered from a library of stock drawings within a set time limit. The web-based version must support multiple teams and players, ensuring unique game and team names, and must guarantee that only one instance of the game exists in memory at any given time.

To address these requirements, we propose a solution leveraging proven software design patterns. The Singleton pattern will be utilized to ensure a single instance of the game in memory, while the Iterator pattern will manage collections of games, teams, and players to ensure name uniqueness and efficient retrieval.

This approach not only meets the client's requirements but also provides a scalable and maintainable foundation for future enhancements. Critical to this process is the adherence to object-oriented principles, ensuring a robust and extensible design.

By implementing these design patterns and principles, CTS will deliver a streamlined, efficient, and reliable web-based gaming application that meets The Gaming Room's needs and supports their expansion to multiple platforms.

Requirements

Business Requirements:

1. Multi-Platform Web Application: Develop a web-based version of "Draw It or Lose It" that supports multiple platforms.

2. Team-Based Gameplay: Enable the game to involve multiple teams, each with multiple players.

3. Unique Names: Ensure that game, team, and player names are unique for ease of identification and to avoid conflicts.

4. Scalability: Design the game to support growth and potential future enhancements.

Technical Requirements:

1. Single Game Instance: Ensure only one instance of the game exists in memory at any given time using the Singleton pattern.

2. Unique Identifiers: Implement unique identifiers for each game, team, and player to maintain consistency and avoid duplication.

3. Name Verification: Allow users to check the availability of game and team names before selection, ensuring uniqueness.

4. Efficient Management: Use the Iterator pattern to efficiently manage collections of games, teams, and players for seamless addition and retrieval.

5. Robust Error Handling: Implement robust error handling and state management to ensure smooth operation in a distributed web environment.

6. Adherence to Best Practices: Follow industry-standard best practices, including appropriate naming conventions and in-line comments, to enhance code readability and maintainability.

Design Constraints

1. Network Latency & Reliability: Developing a web-based game means dealing with variable network conditions, including latency and reliability. Users may experience delays or interruptions due to network issues, affecting gameplay experience.

2. Data Consistency: In a distributed environment, ensuring data consistency across multiple platforms and sessions is crucial. This includes managing concurrent access to game state data to prevent conflicts and inconsistencies.

3. Session Management: Managing user sessions across different devices and ensuring continuity of gameplay is challenging. Proper session handling mechanisms must be in place to track user progress and maintain game state.

4. Concurrency: Multiple users accessing and modifying game data simultaneously can lead to race conditions and data corruption. Concurrency control mechanisms are necessary to handle simultaneous operations efficiently.

5. Scalability: The application must be designed to scale with an increasing number of users, teams, and games. This requires an architecture that can handle growth without performance degradation.

6. Security: Ensuring secure communication and data storage is critical, especially for user information and game data. Security measures must be in place to protect against unauthorized access and data breaches.

Implications on Application Development:

1. Network Latency & Reliability: The application must include mechanisms to handle network interruptions gracefully. This may involve implementing retry logic, caching strategies, and providing user feedback during latency periods to maintain a seamless user experience.

2. Data Consistency: To ensure data consistency, the application will need to implement synchronization mechanisms, such as locking or version control, when accessing or modifying game state data. This prevents conflicts and ensures all users see the same game state.

3. Session Management: Robust session management strategies, such as token-based authentication and persistent storage, are required to maintain user sessions across different devices. This ensures that users can resume gameplay seamlessly from where they left off.

4. Concurrency: Concurrency control can be achieved through techniques like optimistic or pessimistic locking, ensuring that simultaneous access to game data does not lead to inconsistencies. Additionally, atomic operations and transaction management will be crucial to maintaining data integrity.

5. Scalability: The application must be designed with scalability in mind, using distributed architectures such as microservices or cloud-based solutions. Load balancing and auto-scaling features will help manage increased traffic and user load without compromising performance.

6. Security: Implementing security measures such as HTTPS, encryption, and secure authentication mechanisms will protect user data and communications. Regular security audits and compliance with industry standards are necessary to safeguard against potential threats.

NOTE: By addressing these design constraints effectively, the development process will ensure the creation of a robust, scalable, and user-friendly web-based game application that meets the client's requirements and provides a seamless gaming experience.

System Architecture View

The system architecture for the web-based version of "Draw It or Lose It" is designed using a multi-tier architecture, ensuring scalability, maintainability, and robust performance. The architecture is divided into three primary tiers: the Presentation Tier, the Application Tier, and the Data Tier.

1. Presentation Tier:

Components:

Web Browser/Client Application: This is the front-end interface where users interact with the game. It can be accessed via various web browsers on multiple platforms.

User Interface (UI): Built using HTML, CSS, and JavaScript frameworks (such as React or Angular) to provide a responsive and interactive user experience.

Responsibilities:

- Rendering game screens and elements.
- Capturing user inputs (e.g., guesses, team names).
- Communicating with the application tier via HTTP/HTTPS requests.

2. Application Tier:

Components:

Web Server: Hosts the web application and serves static content to clients. Examples include Apache, Nginx, or cloud-based services like AWS Elastic Beanstalk.

Application Server: Executes the business logic of the game. This can be implemented using Java-based frameworks such as Spring Boot or Java EE.

GameService: Singleton service managing game state, game instances, teams, and players. Ensures that only one instance of the game exists in memory.

Responsibilities:

- Processing user requests (e.g., creating a new game, adding a team).
- Managing game logic and state.
- Ensuring data consistency and handling concurrency.
- Communicating with the data tier to retrieve and store game data.

3. Data Tier:

Components:

Database Server: Stores all persistent data related to games, teams, and players. Examples include MySQL, PostgreSQL, or cloud-based databases like Amazon RDS.

Data Access Layer: Manages communication between the application server and the database, using ORM (Object-Relational Mapping) frameworks like Hibernate.

Responsibilities:

- Persisting game state, team information, and player data.
- Ensuring data integrity and security.
- Providing efficient data retrieval and storage operations.

Logical Topology:

Client-Server Communication:

- Clients (web browsers) communicate with the web server using HTTP/HTTPS protocols.
- The web server forwards client requests to the application server for processing.
- The application server processes the requests, executes business logic, and interacts with the database server as needed.
- Responses are sent back through the web server to the clients, ensuring a seamless user experience.

Application Server and Database Communication:

- The application server uses JDBC or ORM frameworks to communicate with the database server.
- Database transactions are managed to ensure consistency, especially in concurrent access scenarios.

Physical Components and Tiers:

Web Server:

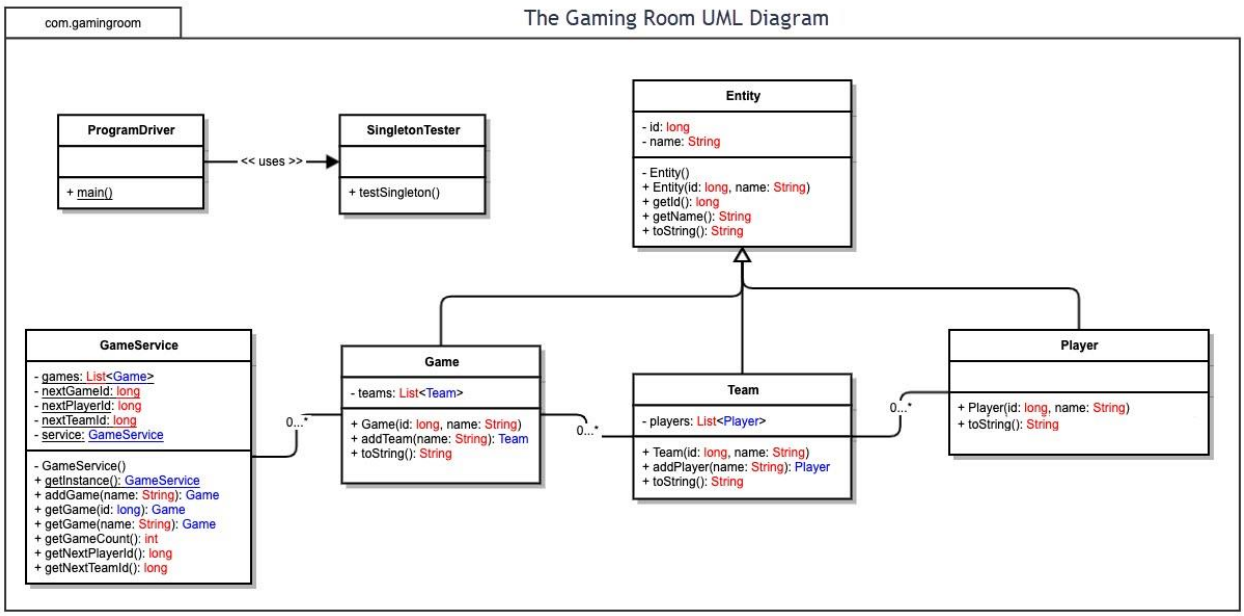
- Could be hosted on a cloud service (e.g., AWS, Azure) or on-premises.
- Responsible for serving static content and routing requests to the application server.

Application Server:
- Runs on a separate machine or virtual instance.
- Handles all game-related processing and business logic.

Database Server:
- Stores game data, running on a dedicated machine or managed database service in the cloud.
- Ensures data persistence and integrity.

In conclusion, this architecture ensures that "Draw It or Lose It" as a game is scalable, maintainable, and provides a robust user experience across multiple platforms. Each tier is designed to handle specific responsibilities, ensuring clear separation of concerns and efficient communication between components.

Domain Model



The UML class diagram outlines several classes and their relationships, demonstrating key object-oriented programming principles to efficiently fulfill software requirements:

1. *Entity class*: Represents a foundational class with properties `id` of type `long` and `name` of type `String`. It serves as a base class for other entities, providing common attributes and behaviors such as `getId()`, `getName()`, and `toString()`.

2. *Game class*: Manages a collection of `Team` instances through a `List<Team> teams`. Each `Game` instance has properties like `id` and `name`, along with methods such as `addTeam(name: String)` to create and add teams, and `toString()` to represent the game's details as a string.

3. *Team class*: Represents a team within a game, containing a `List<Player> players`. It has properties like `id` and `name`, and methods such as `addPlayer(name: String)` to create and add players to the team, and `toString()` to provide a string representation of the team's details.

4. *Player class*: Represents individual players within a team, characterized by properties `id` and `name`, along with a `toString()` method to describe the player's details.

5. *GameService class*: Demonstrates the singleton pattern (`getInstance()` method), ensuring only one instance (`service`) exists throughout the application. It manages a collection of `Game` instances (`games`) and maintains counters (`nextGameId`, `nextPlayerId`, `nextTeamId`) to generate unique identifiers for games, players, and teams.

Relationships and Object-Oriented Principles:

Inheritance: The `Entity` class serves as a base class for `Game`, `Team`, and `Player`, demonstrating inheritance where common attributes and methods (`id`, `name`, `toString()`) are inherited by subclasses, promoting code reusability and maintenance.

Composition: The `Game` class uses composition to manage multiple `Team` instances (`List<Team> teams`). Similarly, `Team` manages multiple `Player` instances (`List<Player> players`). This promotes modular design, where complex objects are built from simpler ones, enhancing flexibility and scalability.

Singleton Pattern: The `GameService` class exemplifies the singleton pattern (`getInstance()` method), ensuring a single global point of access to its instance (`service`). This pattern guarantees that resources managed by `GameService` (such as `games` collection and `next...Id` counters) are centralized and consistently accessed across the application.

Encapsulation: Each class encapsulates its data and behavior within defined boundaries (`private` fields and `public` methods), promoting data integrity and modularity. This encapsulation ensures that classes interact with each other through well-defined interfaces (`public` methods), shielding internal implementations from external interference.

By adhering to these object-oriented principles, the UML class diagram facilitates efficient software design and development for Draw It or Lose It. It promotes code organization, reusability, and maintainability, crucial for building a robust and scalable game application with clear separation of concerns and efficient resource management.

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	**MacOS** is known for its stability and efficient resource management, making it a reliable choice for hosting web-based software applications. It offers a robust development environment with tools like Xcode and supports a wide range of development and deployment tools due to its Unix-based system. Security features like Gatekeeper and system integrity protection enhance its suitability for secure hosting. However, macOS can be costly compared to other operating systems, and its server role limitations and compatibility with certain server software may restrict its scalability and deployment in enterprise-level environments.	Linux is renowned for its stability, performance, and security, making it a top choice for hosting web-based software. It offers cost-effectiveness due to its open-source nature, avoiding licensing fees. Linux's flexibility allows customization through various distributions like Ubuntu and CentOS, optimizing performance and resource use. It supports a wide range of software essential for web hosting, including web servers and databases, ensuring comprehensive support for web development needs. However, Linux can be challenging for beginners due to its command-line interface and initial setup complexity. Compatibility issues with specific hardware and software may arise, requiring careful consideration of distribution choices.	Windows is known for its user-friendly interface and broad compatibility with hardware and software, making it accessible for developers and administrators. It integrates well with enterprise applications and includes the powerful .NET Framework for building robust web applications. Active Directory integration simplifies user management and access control. Microsoft provides strong vendor support with regular updates and technical assistance. However, hosting web applications on Windows can be expensive due to licensing fees. Security concerns exist due to historical vulnerabilities, requiring diligent patching. Windows Server may require higher hardware resources compared to alternatives like Linux. It may also have fewer open-source options.	Mobile devices, such as smartphones and tablets, are known for their portability and touch interfaces, running on various operating systems like iOS and Android. They provide ubiquitous access to web-based applications, making them convenient for users on-the-go. The touch interface enhances interaction, while built-in connectivity options ensure continuous access via Wi-Fi or cellular networks. Mobile devices support native app development, enabling web applications to be packaged for enhanced performance and offline use. They seamlessly integrate with device features like GPS and cameras, enhancing functionality and user engagement. However, mobile devices have limitations in processing power, memory, and screen size compared to desktops or servers. This can impact the performance and user interface of complex web

Client Side	Supporting multiple client types on Mac involves considering several critical factors. Firstly, there's the cost aspect: while Xcode, the primary development tool, is free, the requirement for macOS hardware adds initial investment. Ensuring compatibility across various client configurations, such as different macOS versions and iOS devices via macOS, may also necessitate investing in testing infrastructure like VMware Fusion or Parallels Desktop. Secondly, time is crucial: developing for Mac entails learning and implementing frameworks like Cocoa and Swift, which can extend development timelines. Testing and debugging across diverse client setups further adds to the time investment. Expertise is equally vital, as developers need specialized knowledge in	Supporting multiple client types on Linux involves considering several key software development factors. Firstly, cost considerations are favorable as Linux development tools are typically open-source, minimizing initial expenses compared to proprietary alternatives. However, ensuring compatibility across various Linux distributions like Ubuntu and CentOS may require investment in testing infrastructure, such as virtual machines or cloud platforms, to cover diverse client environments effectively. Secondly, time and expertise are crucial. Developing for Linux involves understanding and testing applications across different distributions and versions, which can extend development timelines	Supporting multiple client types on Windows involves navigating several key software development considerations. Firstly, there are cost implications: while widely-used tools like Visual Studio incur licensing fees, they provide robust development environments crucial for Windows application development. Ensuring compatibility across various Windows versions, from desktop editions to server environments, may require investment in diverse testing infrastructure such as virtual machines or physical devices, impacting initial setup costs. Secondly, time and expertise are critical factors. Developing for Windows entails testing applications across different versions and integrating with Microsoft technologies like	Supporting multiple client types on mobile devices requires careful consideration of key software development factors. Cost management involves leveraging free IDEs like Android Studio and Xcode while potentially investing in additional tools or cloud-based testing platforms to ensure compatibility across diverse devices and operating system versions. Time considerations revolve around platform-specific development using languages like Java/Kotlin for Android and Swift for iOS, alongside designing responsive user interfaces that work seamlessly across various screen sizes. Expertise in mobile-specific APIs, design guidelines, and familiarity with cross-platform frameworks such as Flutter or React Native is essential for optimizing application performance and user experience across different mobile platforms. Successful development hinges on balancing these factors to deliver

Development Tools	To develop software for deployment on macOS, developers primarily use Swift as the main programming language, designed for seamless integration with Apple's Cocoa frameworks. Objective-C, although less commonly used now, remains relevant for legacy macOS applications. The essential Integrated Development Environment (IDE) for macOS development is Xcode, offering comprehensive tools for coding, debugging, and testing Swift and Objective-C applications, along with Interface Builder for designing user interfaces and Instruments for performance analysis. Cocoa and Cocoa Touch frameworks provide essential APIs for building macOS and iOS applications respectively. Alternatively, developers can use JetBrains' AppCode as an IDE for	To develop software for Linux deployment, developers commonly use versatile programming languages such as C/C++, Python, Java, and Go, each serving different purposes from system-level programming to web development. Integrated Development Environments (IDEs) like Visual Studio Code, Eclipse, and IntelliJ IDEA support these languages, offering robust coding environments on Linux. Text editors like Sublime Text, Vim, and Emacs are also popular for their customization and scripting capabilities. Essential tools include GCC for compiling C/C++ programs, Docker and Kubernetes for containerization and orchestration, and Git for version control. These tools empower developers to create diverse applications—	To develop software for Windows deployment, developers commonly rely on versatile programming languages such as C#, C++, JavaScript/TypeScript, Python, and Java, each serving different purposes from desktop applications to web development. Key Integrated Development Environments (IDEs) include Visual Studio for comprehensive Windows development with C# and C++, and Visual Studio Code for lightweight, cross-platform coding across various languages. Eclipse and JetBrains' IDEs like PyCharm and IntelliJ IDEA support Java and Python development on Windows, enhancing productivity with debugging and testing capabilities. Essential tools encompass .NET Framework and .NET Core for building Windows applications	To develop software for mobile devices, developers typically use Java and Kotlin for Android apps, Swift for iOS, and JavaScript/TypeScript for cross-platform frameworks like React Native. C/C++ is also used for performance-intensive tasks and game development. Key IDEs include Android Studio for Android development, Xcode for iOS, and Visual Studio with Xamarin for cross-platform apps. React Native and Flutter are popular frameworks for cross-platform development, leveraging JavaScript/TypeScript and Dart respectively. Unity is used for mobile game development. Tools like Firebase provide backend services. These tools enable developers to create diverse mobile applications, leveraging platform-specific features and cross-platform frameworks for efficient development and deployment across iOS, Android, and other mobile platforms.

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**:

   Based on our goal to expand Draw It or Lose It to various computing environments, I recommend leveraging a cross-platform development approach. Platforms like React Native or Flutter offer excellent solutions. React Native allows us to develop applications using JavaScript/TypeScript, sharing code between iOS and Android, thereby reducing development time.

2. **Operating Systems Architectures**:

   React Native employs a bridge architecture where JavaScript runs in a background thread and communicates with native modules through a bridge. This setup allows React Native apps to render UI components using native APIs provided by iOS and Android platforms. By leveraging this approach, React Native ensures that the apps maintain performance and a native-like experience. Developers bundle JavaScript code along with the native components, enabling deployment of the app to both iOS (via Xcode) and Android (via Android Studio) platforms seamlessly.

3. **Storage Management**:

   For our React Native application, I would recommend using AsyncStorage for local storage and Firebase Firestore for cloud-based storage. AsyncStorage is perfect for storing small amounts of data locally on the device, such as user preferences and cached data, due to its simplicity and asynchronous nature.

   On the other hand, Firebase Firestore offers a scalable NoSQL database solution for structured data storage. It provides real-time syncing, offline support, and robust security, making it ideal for managing user profiles, game states, and other application data that needs to be accessed across devices. Integrating these storage solutions will ensure efficient data management and a seamless experience for users of Draw It or Lose It across various platforms.

4. **Memory Management**:

   React Native utilizes effective memory management techniques to optimize performance for the Draw It or Lose It software. Memory management primarily relies on the JavaScript engine's garbage collection mechanism, which automatically deallocates memory used by objects no longer in use, thereby preventing memory leaks and ensuring efficient memory usage.

   Components in React Native are dynamically mounted and unmounted based on user interactions and application state changes. When a component is unmounted, React Native releases associated memory resources, including event listeners and state variables, to maintain optimal performance. Images and assets are managed carefully to minimize memory consumption, employing techniques like lazy loading and caching. Additionally, state

management tools such as Redux are used to centralize application state and optimize memory usage by controlling data retention. React Native also provides profiling and optimization tools to monitor memory allocations, detect memory leaks, and enhance application performance through continuous memory management improvements. These practices collectively contribute to a responsive and stable user experience for Draw It or Lose It across various mobile platforms.

5. **Distributed Systems and Networks**:

   To enable Draw It or Lose It to communicate across multiple platforms using distributed software and networks, strategic architectural decisions and technical implementations are essential. A service-oriented architecture (SOA) or microservices approach would be beneficial, dividing the application into independent services that handle specific functionalities like game logic and user management. Each service communicates through well-defined APIs, facilitating seamless interaction between platforms such as iOS, Android, and web clients.

   Robust APIs and communication protocols play a crucial role in ensuring reliable data exchange. Utilizing RESTful APIs for standard data operations and WebSocket for real-time updates can accommodate diverse communication needs. These protocols manage data transmission formats and reliability over networks, supporting efficient and responsive interactions across devices.

   Network connectivity poses a critical dependency. Implementing strategies like offline-first design, local data caching, and asynchronous communication helps mitigate the impact of network outages. Retry mechanisms for failed requests and fallback strategies, such as displaying cached data, enhance user experience continuity during connectivity disruptions.

   Scalability is another consideration. Load balancing techniques distribute incoming traffic among backend servers to optimize performance and maintain reliability under varying user loads. Cloud infrastructure services offer scalability features like auto-scaling and serverless computing, ensuring the application can handle increasing user demands effectively.

   Security and data integrity are paramount. Implementing encryption, authentication, and authorization mechanisms safeguard user data and ensure compliance with security standards. Secure communication protocols like HTTPS and SSL/TLS encrypt data during transmission, preventing unauthorized access and maintaining data confidentiality across distributed systems.

   By integrating these architectural strategies and technical measures, Draw It or Lose It can achieve robust cross-platform communication capabilities while addressing dependencies on network connectivity, ensuring scalability, enhancing security, and maintaining data integrity. This approach supports a seamless and reliable gaming experience across different devices and platforms for users.

6. **Security**:
   Ensuring robust security measures for protecting user information across various platforms, particularly with React Native, involves implementing several key strategies. First and foremost is data encryption, which should be applied rigorously using strong encryption standards such as

AES-256. This ensures that sensitive user data, whether stored locally using AsyncStorage or in cloud databases like Firebase Firestore, remains securely encrypted both at rest and in transit. Encryption plays a crucial role in safeguarding data integrity and confidentiality, mitigating risks associated with unauthorized access.

Authentication and authorization mechanisms are equally vital components of a secure system. Implementing secure authentication protocols such as OAuth or JWT helps verify user identities and control access to application resources. Multi-factor authentication (MFA) adds an extra layer of security by requiring users to authenticate through multiple verification methods, further fortifying access controls and reducing the risk of unauthorized account access.

Secure communication protocols, like HTTPS, are essential for protecting data transmitted between clients and servers. React Native inherently supports HTTPS, ensuring that all data exchanges are encrypted to prevent interception and tampering by malicious actors. This foundational security measure strengthens the overall integrity of communications and protects sensitive user information during transit over the network.

Regular security audits and updates are critical to maintaining a secure environment. Conducting thorough security assessments and vulnerability scans helps identify and mitigate potential weaknesses in the application and infrastructure. Keeping software dependencies and frameworks up to date with the latest security patches and fixes is essential to addressing known vulnerabilities promptly, thereby reducing the likelihood of exploitation.

Educating users about best security practices enhances overall protection. Providing clear guidelines on creating strong passwords, recognizing phishing attempts, and understanding data handling policies empowers users to actively participate in safeguarding their own information. Transparent communication about privacy practices and data usage policies builds trust and confidence among users, reinforcing their perception of security within the application.

By integrating these comprehensive security measures and leveraging the robust security capabilities of React Native, Draw It or Lose It can establish a resilient security framework. This framework not only protects user data across different platforms but also strengthens user trust and satisfaction, ensuring a secure and reliable experience for all users of the application.