Architecture Planning

Overview

Da'math is an educational two-player board game that combines the classic mechanics of checkers with math operations to create an engaging and interactive learning experience. It is designed to help students strengthen their math skills while having fun. Each player takes turns capturing opponent pieces, just like in checkers, but with an added twist: each capture involves solving a math problem.

This game encourages students to think critically, solve problems, and plan strategically, making it an ideal tool for classroom use or practice at home. It's especially effective for making math enjoyable for learners who struggle with traditional teaching methods. The application comprises three primary layers:

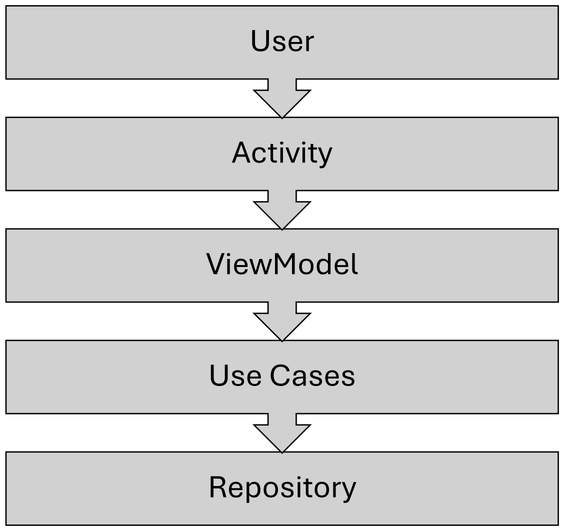
1. **Presentation Layer:** Handles UI rendering and user interactions.
2. **Domain Layer:** Contains business logic and use cases.
3. **Data Layer:** Manages data sources and repositories.

Presentation Layer		
Components		**Responsibilities**
Activities	`UsernameActivity`: Captures player usernames	• Render UI components based on the state exposed by ViewModels. • Handle user interactions and delegate actions to ViewModels. • Observe LiveData or StateFlow from ViewModels to update the UI reactively.
	`MainActivity`: Manages the game board and gameplay	
	`HowToPlayActivity`: Displays game instructions	
ViewModels	`UsernameViewModel`: Processes and validates username inputs	
	`GameViewModel`: Manages game state, including the board, scores, and turn logic	
Domain Layer		
Use Cases	`ValidateUsernamesUseCase`: Ensures entered usernames meet criteria	• Encapsulate business logic specific to the Damath game. • Serve as an intermediary between the Presentation and Data layers. • Ensure that business rules are applied consistently across the application.
	`InitializeGameBoardUseCase`: Sets up the initial game board state	
	`ProcessMoveUseCase`: Validates and processes player moves	
	`CalculateScoreUseCase`: Computes scores based on game rules	
Data Layer		
Repositories	`GameRepository`: Provides access to game data and operations	• Abstract data operations and provide a clean API for the Domain layer. • Manage data retrieval and persistence, even though this application uses in-memory storage.
Data Sources	`LocalDataSource`: Manages in-memory data storage for the game state	

Data Flow

The application follows a unidirectional data flow:

1. **User Interaction:** The user interacts with the UI (e.g., entering usernames, making moves).
2. **ViewModel Processing:** The corresponding ViewModel handles the interaction, invoking appropriate use cases.
3. **Use Case Execution:** Use cases execute business logic and interact with repositories as needed.
4. **Data Retrieval/Update:** Repositories fetch or update data from the data sources.
5. **UI Update:** ViewModels expose updated state via LiveData or StateFlow, which the UI observes to render changes.



Advantages of This Architecture

- **Separation of Concerns:** Each layer has distinct responsibilities, making the codebase more manageable.
- **Testability:** Business logic in use cases can be unit tested independently of the UI.
- **Scalability:** The architecture supports adding new features with minimal impact on existing code.
- **Maintainability:** Clear boundaries between components facilitate easier maintenance and updates.

Future Enhancements

- **Persistence:** Implement persistent storage (e.g., Room database) to save game states.
- **Networking:** Add multiplayer capabilities using network communication.
- **Dependency Injection:** Integrate a DI framework like Hilt for better management of dependencies.
- **Jetpack Compose:** Transition to Jetpack Compose for modern UI development.