📚 Documentação Técnica Completa

Planning Poker

Aplicação Flutter para Sessões de Estimativa Ágil em Tempo Real

---

Versão: 1.0.0
Data: Janeiro 2026
Autor: Daniel Brown
Tecnologias: Flutter 3.9+, Dart 3.9+, Firebase Realtime Database

---

Sumário

---

1. Introdução

1.1 Sobre o Projeto

Planning Poker é uma técnica de estimativa ágil amplamente utilizada por equipes de desenvolvimento de software para estimar o esforço necessário para completar tarefas ou histórias de usuário. Esta aplicação foi desenvolvida para permitir que equipes realizem sessões de Planning Poker de forma remota e em tempo real.

1.2 Objetivos

- ✅ Demonstrar implementação de **Clean Architecture** em Flutter
- ✅ Aplicar **padrões de projeto** modernos e boas práticas
- ✅ Implementar **sincronização em tempo real** com Firebase

- ✅ Criar código **testável** e **manutenível**
- ✅ Suportar **múltiplas plataformas** (Android, iOS, Web, Windows, macOS, Linux)

1.3 Tecnologias Utilizadas

Tecnologia	Versão	Descrição
Flutter	3.9+	Framework UI multiplataforma
Dart	3.9+	Linguagem de programação
Firebase	-	Backend em tempo real
Provider	-	Gerenciamento de estado

Dependências Principais

Pacote	Uso
`provider`	Gerenciamento de estado e DI
`firebase_core`	Core do Firebase
`firebase_database`	Realtime Database
`uuid`	Geração de IDs únicos
`google_fonts`	Fontes customizadas
`lottie`	Animações
`flip_card`	Animação de virar carta

2. Visão Geral do Sistema

2.1 Funcionalidades

Para o Host (Facilitador)

Funcionalidade	Descrição
🎲 Criar Sessão	Criar nova sessão de Planning Poker
🔑 Compartilhar Código	Código de 6 caracteres para outros jogadores
👁️ Revelar Cartas	Revelar todas as cartas simultaneamente
🔄 Resetar Rodada	Reiniciar para nova estimativa
🚪 Encerrar Sessão	Remove todos os participantes

Para Participantes

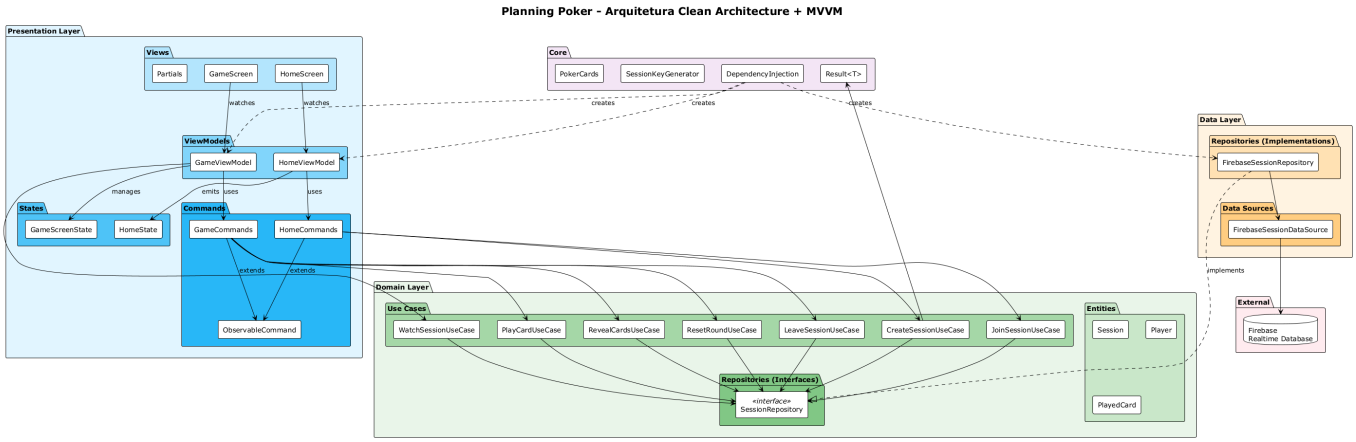
Funcionalidade	Descrição
Entrar em Sessão	Via código de 6 caracteres
Selecionar Carta	Escolher carta de estimativa
Ver Participantes	Lista em tempo real
Ver Resultados	Após revelação das cartas

Cartas Disponíveis

```
0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ?,
```

Baseado na sequência de Fibonacci + cartas especiais (incerteza e pausa)

2.2 Diagrama de Visão Geral da Arquitetura



A imagem acima ilustra a arquitetura completa do sistema, mostrando:

- **Presentation Layer**: Views, ViewModels, States e Commands
- **Domain Layer**: Entities, Use Cases e Repository Interfaces
- **Data Layer**: Repository Implementations e Data Sources
- **External**: Firebase Realtime Database

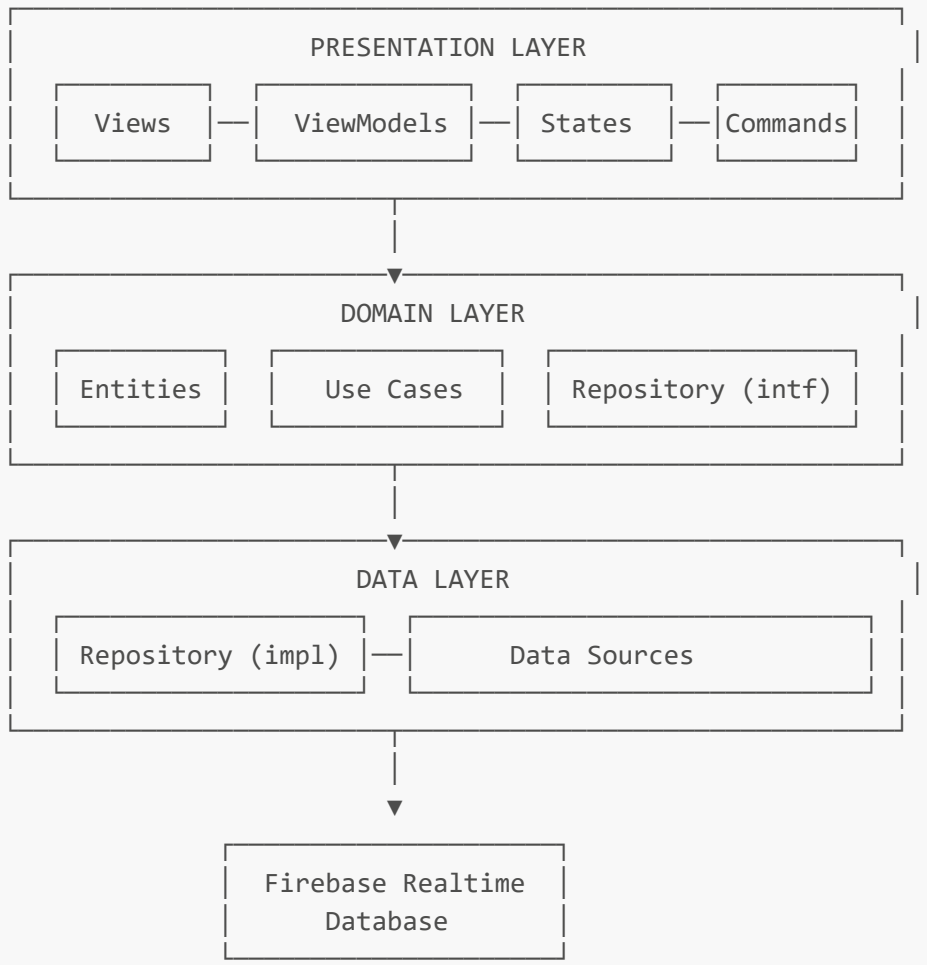
---

3. Arquitetura do Software

3.1 Clean Architecture

O projeto segue os princípios da **Clean Architecture** de Robert C. Martin, combinada com o padrão **MVVM** (Model-View-ViewModel) na camada de apresentação.

Camadas da Arquitetura



3.2 Princípios SOLID Aplicados

Princípio	Aplicação no Projeto
Single Responsibility	Cada classe tem uma única responsabilidade bem definida
Open/Closed	Extensível via interfaces, fechado para modificação
Liskov Substitution	Repositórios implementam interfaces do domínio
Interface Segregation	Interfaces específicas por contexto de uso
Dependency Inversion	Camadas superiores não dependem de implementações inferiores

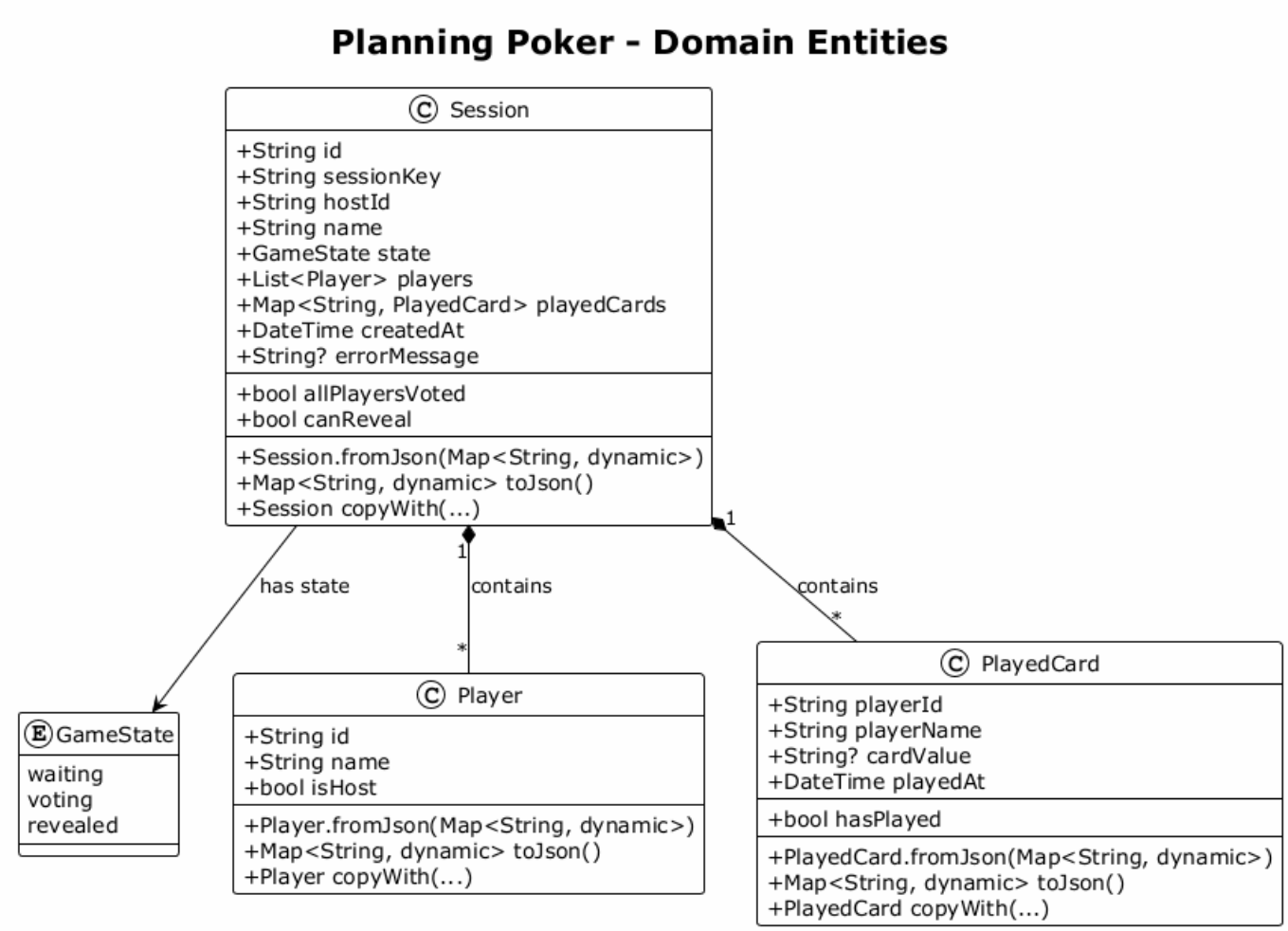
3.3 Benefícios da Arquitetura

1. **Testabilidade**: Cada camada pode ser testada isoladamente
2. **Manutenibilidade**: Mudanças em uma camada não afetam outras
3. **Escalabilidade**: Fácil adicionar novas funcionalidades
4. **Reusabilidade**: Componentes podem ser reutilizados
5. **Independência de Framework**: Domínio não depende de Flutter/Firebase

---

4. Entidades do Domínio

4.1 Diagrama de Entidades



4.2 Descrição das Entidades

4.2.1 Session (Sessão)

A entidade `Session` representa uma sessão de Planning Poker.

Atributo	Tipo	Descrição
`id`	String	Identificador único (UUID)
`sessionKey`	String	Código de 6 caracteres para entrar
`hostId`	String	ID do jogador que criou a sessão
`name`	String	Nome da sessão
`state`	GameState	Estado atual (waiting, voting, revealed)
`players`	List	Lista de jogadores
`playedCards`	Map<String, PlayedCard>	Cartas jogadas por jogador
`createdAt`	DateTime	Data de criação
`errorMessage`	String?	Mensagem de erro (opcional)

Propriedades Computadas:

- `allPlayersVoted`: Verifica se todos jogadores votaram
- `canReveal`: Verifica se é possível revelar as cartas

4.2.2 Player (Jogador)

A entidade `Player` representa um participante da sessão.

Atributo	Tipo	Descrição
`id`	String	Identificador único (UUID)
`name`	String	Nome do jogador
`isHost`	bool	Indica se é o host/facilitador

4.2.3 PlayedCard (Carta Jogada)

A entidade `PlayedCard` representa uma carta jogada por um jogador.

Atributo	Tipo	Descrição
`playerId`	String	ID do jogador que jogou
`playerName`	String	Nome do jogador
`cardValue`	String?	Valor da carta (null = não jogou)
`playedAt`	DateTime	Momento em que jogou

Propriedades Computadas:

- `hasPlayed`: Verifica se o jogador já jogou uma carta

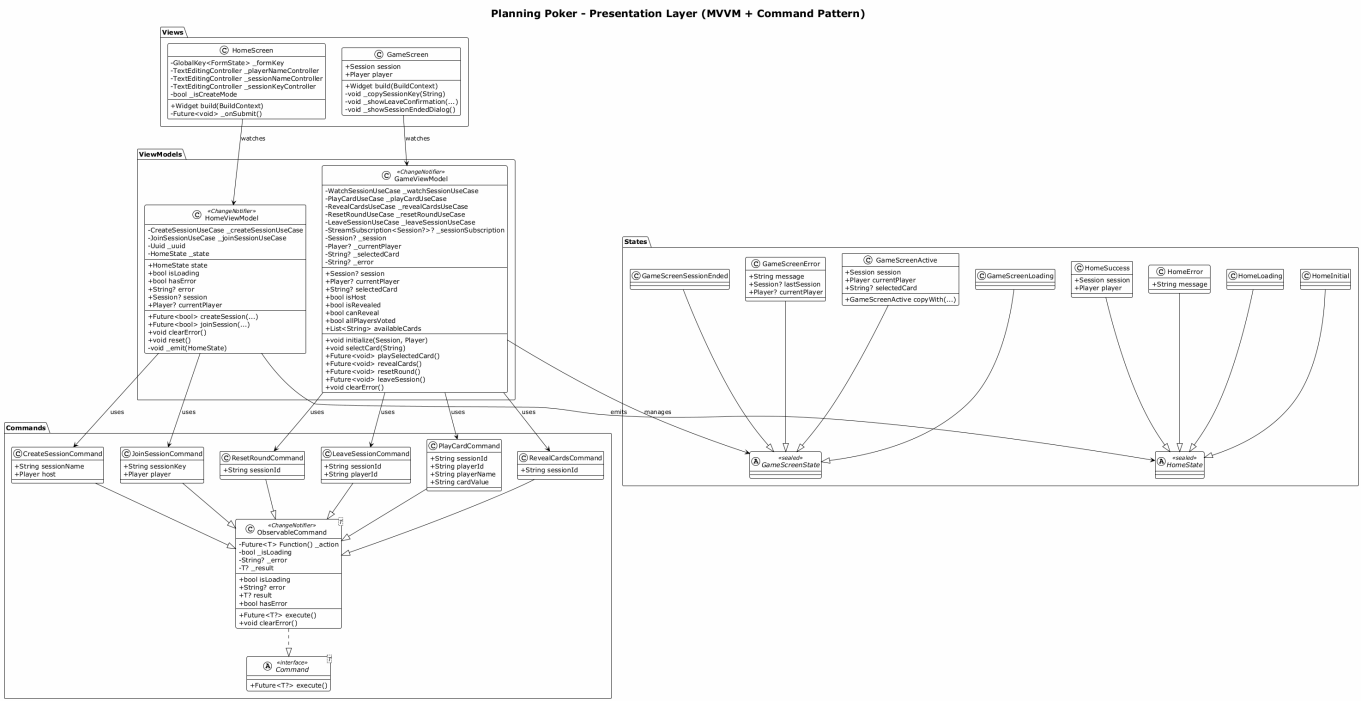
4.2.4 GameState (Enum)

Enumeração que define os estados possíveis do jogo:

Estado	Descrição
`waiting`	Aguardando jogadores
`voting`	Votação em andamento
`revealed`	Cartas reveladas

5. Camada de Apresentação (MVVM)

5.1 Diagrama da Camada de Apresentação



5.2 Componentes

5.2.1 Views (Telas)

As Views são responsáveis pela renderização da UI de forma declarativa.

HomeScreen

Tela inicial onde o usuário pode:

- Criar uma nova sessão
- Entrar em uma sessão existente

Responsabilidades:

- Exibir formulário de criação/entrada
- Validar inputs do usuário
- Navegar para GameScreen após sucesso

GameScreen

Tela principal do jogo de Planning Poker.

Responsabilidades:

- Exibir mesa de jogo com participantes
- Permitir seleção de cartas
- Mostrar controles do host (revelar, resetar)
- Exibir resultados quando revelados

5.2.2 ViewModels

Os ViewModels gerenciam o estado e a lógica de apresentação.

HomeViewModel

```
class HomeViewModel extends ChangeNotifier {
    // Estado
    HomeState get state
    bool get isLoading
    bool get hasError
    String? get error
    Session? get session
    Player? get currentPlayer

    // Ações
    Future<bool> createSession(sessionName, playerName)
    Future<bool> joinSession(sessionKey, playerName)
    void clearError()
    void reset()
}
```

GameViewModel

```
class GameViewModel extends ChangeNotifier {
    // Estado
    Session? get session
    Player? get currentPlayer
    String? get selectedCard
    bool get isHost
    bool get isRevealed
    bool get canReveal
    bool get allPlayersVoted
    List<String> get availableCards

    // Ações
    void initialize(Session, Player)
    void selectCard(String)
    Future<void> playSelectedCard()
    Future<void> revealCards()
    Future<void> resetRound()
    Future<void> leaveSession()
}
```

5.2.3 States (Estados)

Estados da UI implementados como **sealed classes** para garantir exaustividade.

HomeState

```
sealed class HomeState {
    const HomeState();
}

class HomeInitial extends HomeState { }
class HomeLoading extends HomeState { }
class HomeError extends HomeState {
    final String message;
}
class HomeSuccess extends HomeState {
    final Session session;
    final Player player;
}
```

Benefícios:

- ✅ Exaustividade garantida pelo compilador
- ✅ Type-safe pattern matching
- ✅ Estados mutuamente exclusivos

5.2.4 Commands (Command Pattern)

Commands encapsulam operações assíncronas com gerenciamento automático de loading/erro.

```
class CreateSessionCommand extends ObservableCommand<Session?> {
    CreateSessionCommand({
        required CreateSessionUseCase useCase,
        required this.sessionName,
        required this.host,
    }) : super(() async {
        final result = await useCase.execute(...);
        return result.when(
            success: (session) => session,
            failure: (message) => throw Exception(message),
        );
    });
}
```

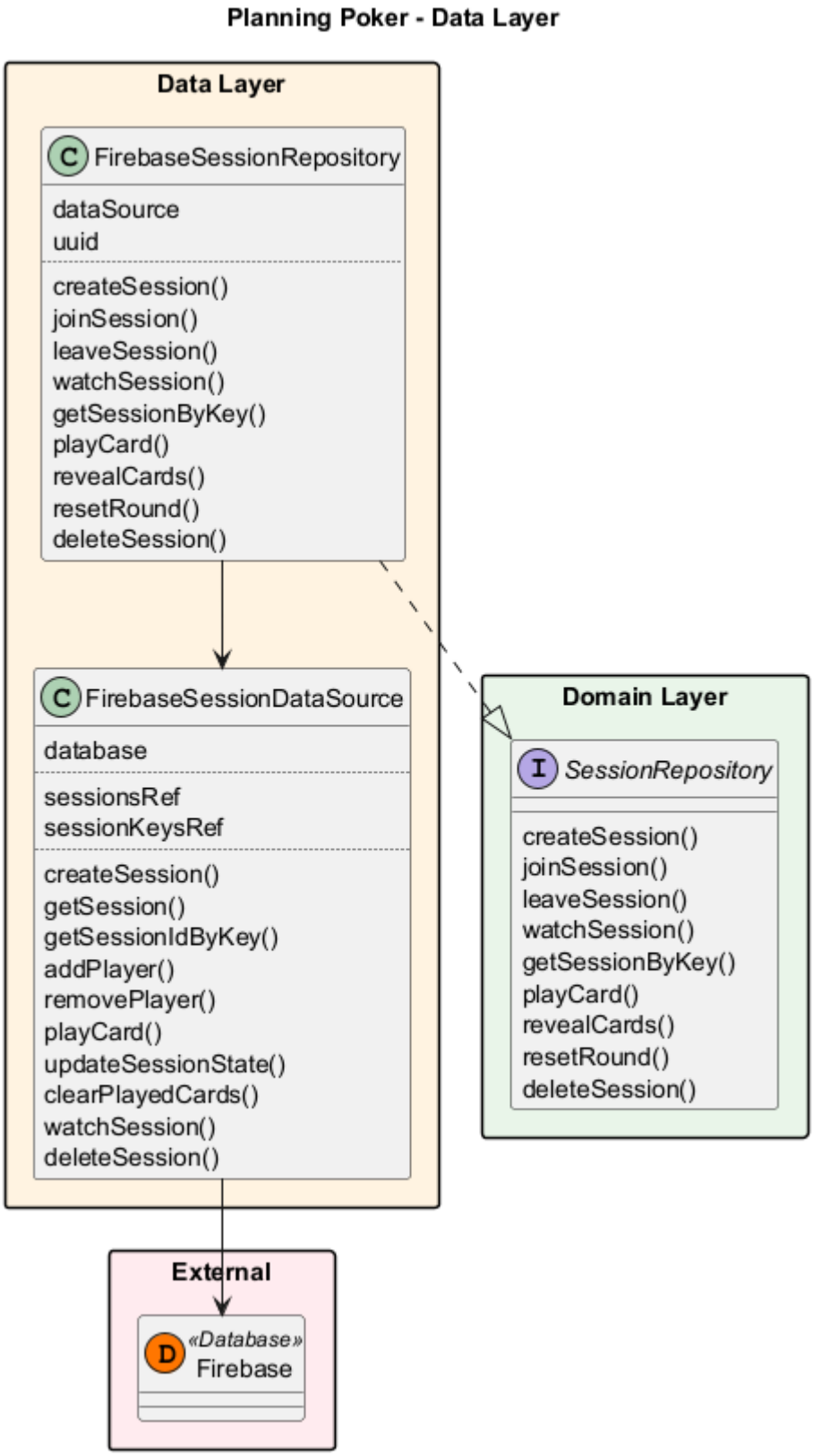
Benefícios:

- ✅ Encapsula lógica de execução
- ✅ Gerencia estados de loading/erro automaticamente
- ✅ Reutilizável entre ViewModels
- ✅ Testável isoladamente

---

6. Camada de Dados

6.1 Diagrama da Camada de Dados



6.2 Repository Pattern

O padrão Repository abstrai a fonte de dados do domínio.

6.2.1 Interface (Domínio)

```
abstract class SessionRepository {
    Future<Session> createSession({
        required String name,
        required Player host
    });

    Future<Session> joinSession({
        required String sessionKey,
        required Player player
    });

    Stream<Session?> watchSession(String sessionId);

    Future<void> playCard({
        required String sessionId,
        required String playerId,
        required String playerName,
        required String? cardValue,
    });

    Future<void> revealCards(String sessionId);

    Future<void> resetRound(String sessionId);

    Future<void> leaveSession({
        required String sessionId,
        required String playerId,
    });
}
```

6.2.2 Implementação (Dados)

```
class FirebaseSessionRepository implements SessionRepository {
    final FirebaseSessionDataSource _dataSource;
    final Uuid _uuid;

    // Implementa todos os métodos da interface
    // usando o DataSource para acessar Firebase
}
```

6.3 Data Source

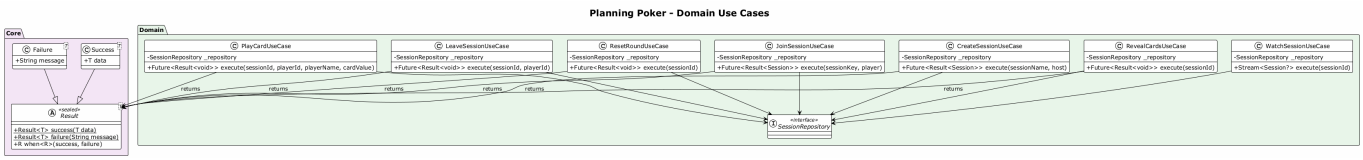
O `FirebaseSessionDataSource` é responsável pelo acesso direto ao Firebase.

```
class FirebaseSessionDataSource {
    final FirebaseDatabase database;

    // Referências do banco
```

```
    DatabaseReference get sessionsRef
    DatabaseReference get sessionKeysRef

    // Operações CRUD
    Future<void> createSession(Session session)
    Future<Session?> getSession(String id)
    Future<String?> getSessionIdByKey(String key)
    Future<void> addPlayer(String sessionId, Player player)
    Future<void> removePlayer(String sessionId, String playerId)
    Future<void> playCard(String sessionId, PlayedCard card)
    Future<void> updateSessionState(String sessionId, GameState state)
    Future<void> clearPlayedCards(String sessionId)
    Stream<Session?> watchSession(String sessionId)
    Future<void> deleteSession(String sessionId)
}
```

7. Casos de Uso

7.1 Diagrama de Casos de Uso



7.2 Lista de Casos de Uso

7.2.1 CreateSessionUseCase

Propósito: Criar uma nova sessão de Planning Poker.

Entrada:

- `sessionName`: Nome da sessão
- `host`: Jogador que será o host

Saída: `Result<Session>`

Fluxo:

1. Gera ID único para a sessão
2. Gera código de 6 caracteres
3. Cria objeto Session
4. Persiste no Firebase
5. Retorna sessão criada

7.2.2 JoinSessionUseCase

Propósito: Entrar em uma sessão existente.

Entrada:

- `sessionKey`: Código de 6 caracteres
- `player`: Dados do jogador

Saída: `Result<Session>`

Fluxo:

1. Busca sessão pelo código
2. Valida se sessão existe
3. Adiciona jogador à sessão
4. Retorna sessão atualizada

7.2.3 PlayCardUseCase

Propósito: Jogar uma carta na sessão.

Entrada:

- `sessionId`: ID da sessão
- `playerId`: ID do jogador
- `playerName`: Nome do jogador
- `cardValue`: Valor da carta

Saída: `Result<void>`

7.2.4 RevealCardsUseCase

Propósito: Revelar todas as cartas (apenas host).

Entrada:

- `sessionId`: ID da sessão

Saída: `Result<void>`

7.2.5 ResetRoundUseCase

Propósito: Resetar a rodada para nova votação.

Entrada:

- `sessionId`: ID da sessão

Saída: `Result<void>`

7.2.6 LeaveSessionUseCase

Propósito: Sair de uma sessão.

Entrada:

- `sessionId`: ID da sessão

- `playerId`: ID do jogador

Saída: `Result<void>`

Comportamento especial: Se o host sair, a sessão é encerrada para todos.

7.2.7 WatchSessionUseCase

Propósito: Observar mudanças na sessão em tempo real.

Entrada:

- `sessionId`: ID da sessão

Saída: `Stream<Session?>`

7.3 Result Pattern

O padrão Result (similar ao Either) trata explicitamente sucesso e falha:

```
sealed class Result<T> {
    factory Result.success(T data) = Success<T>;
    factory Result.failure(String message) = Failure<T>;

    R when<R>({
        required R Function(T data) success,
        required R Function(String message) failure,
    });
}
```

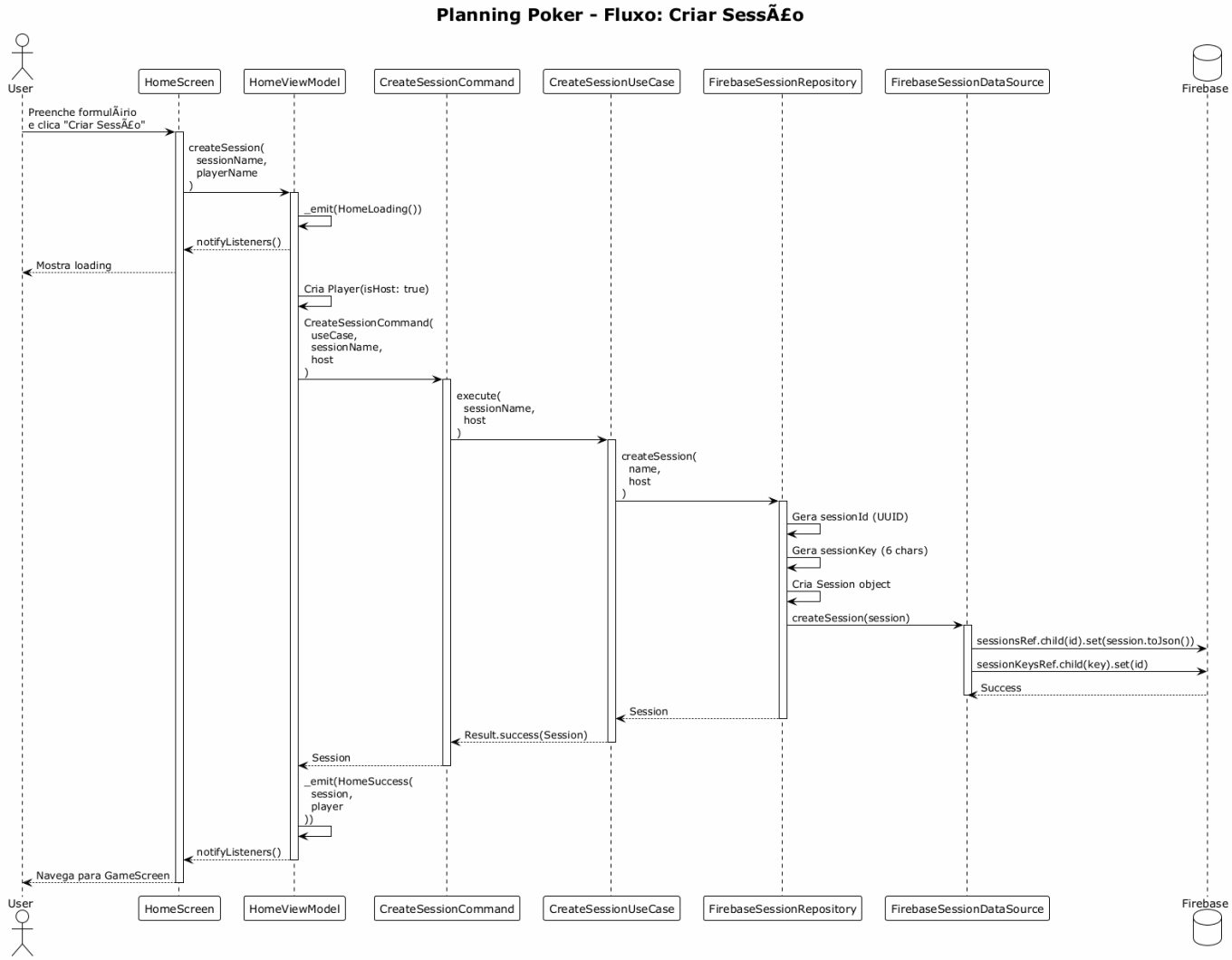
Uso:

```
final result = await useCase.execute(...);
return result.when(
    success: (session) => session,
    failure: (message) => throw Exception(message),
);
```

---

8. Fluxos de Operação

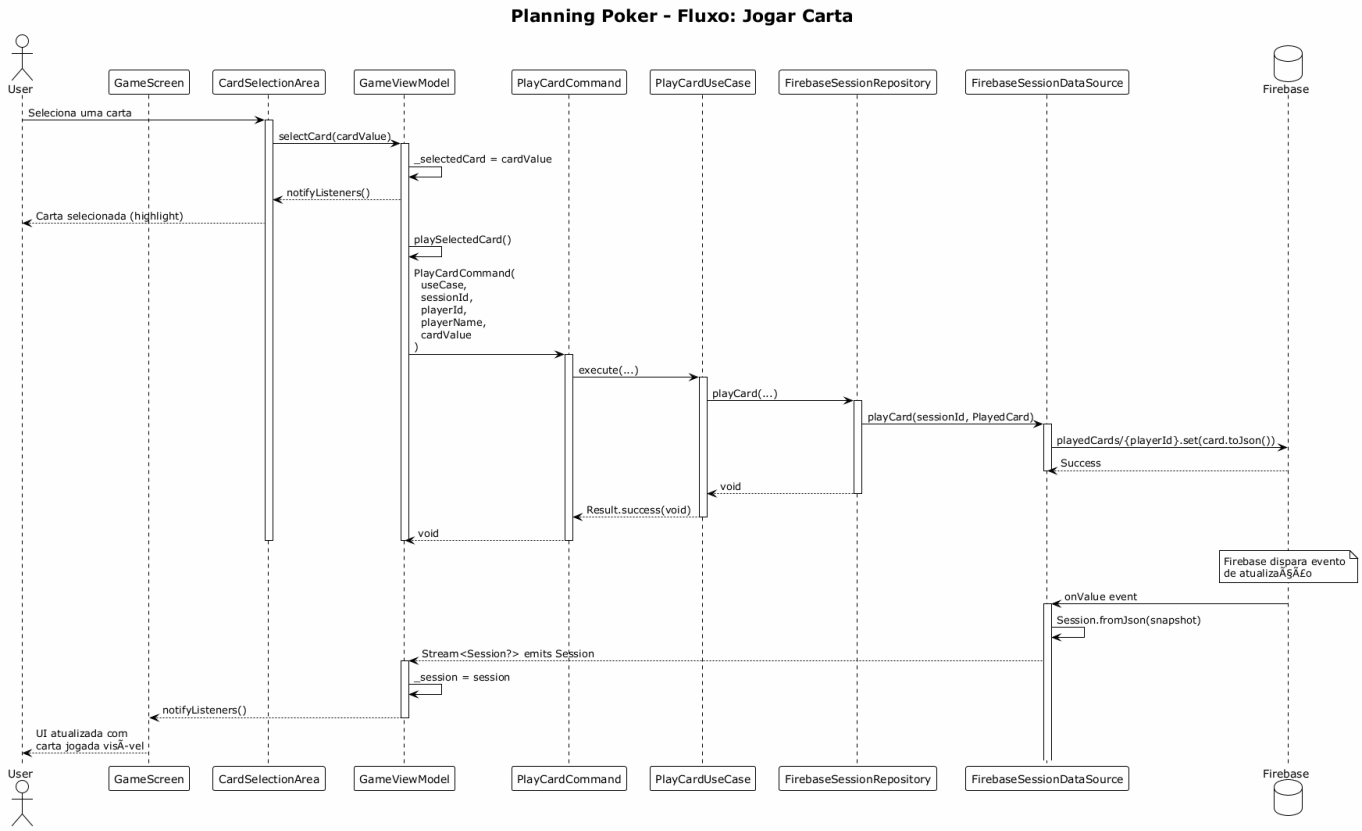
8.1 Fluxo: Criar Sessão



Descrição do Fluxo

1. **Usuário** preenche formulário e clica "Criar Sessão"
2. **HomeScreen** chama `createSession()` no ViewModel
3. **HomeViewModel** emite estado `HomeLoading`
4. **HomeViewModel** cria Player como host
5. **CreateSessionCommand** executa o UseCase
6. **CreateSessionUseCase** gera IDs e chama Repository
7. **FirebaseSessionRepository** usa DataSource para persistir
8. **FirebaseSessionDataSource** salva no Firebase
9. Retorno sucesso propaga de volta
10. **HomeViewModel** emite `HomeSuccess`
11. **HomeScreen** navega para GameScreen

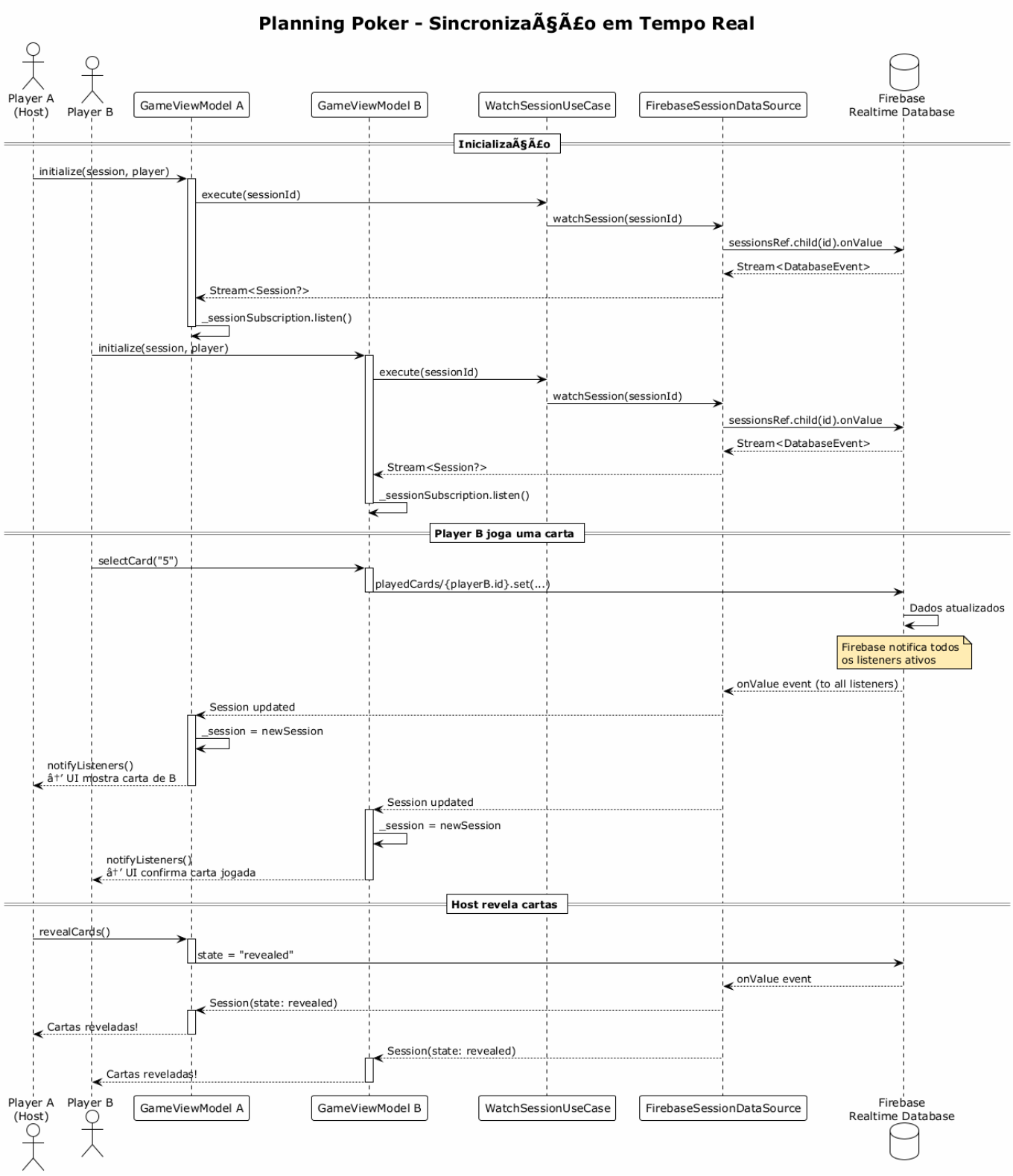
8.2 Fluxo: Jogar Carta



Descrição do Fluxo

1. **Usuário** seleciona uma carta
2. **CardSelectionArea** chama `selectCard()` no ViewModel
3. **GameViewModel** armazena carta selecionada
4. **PlayCardCommand** executa o UseCase
5. **UseCase** chama Repository para salvar
6. **Firebase** persiste a carta jogada
7. **Firebase** dispara evento de atualização
8. **Stream** propaga sessão atualizada
9. **GameViewModel** atualiza estado
10. **UI** reflete carta jogada

8.3 Fluxo: Sincronização em Tempo Real



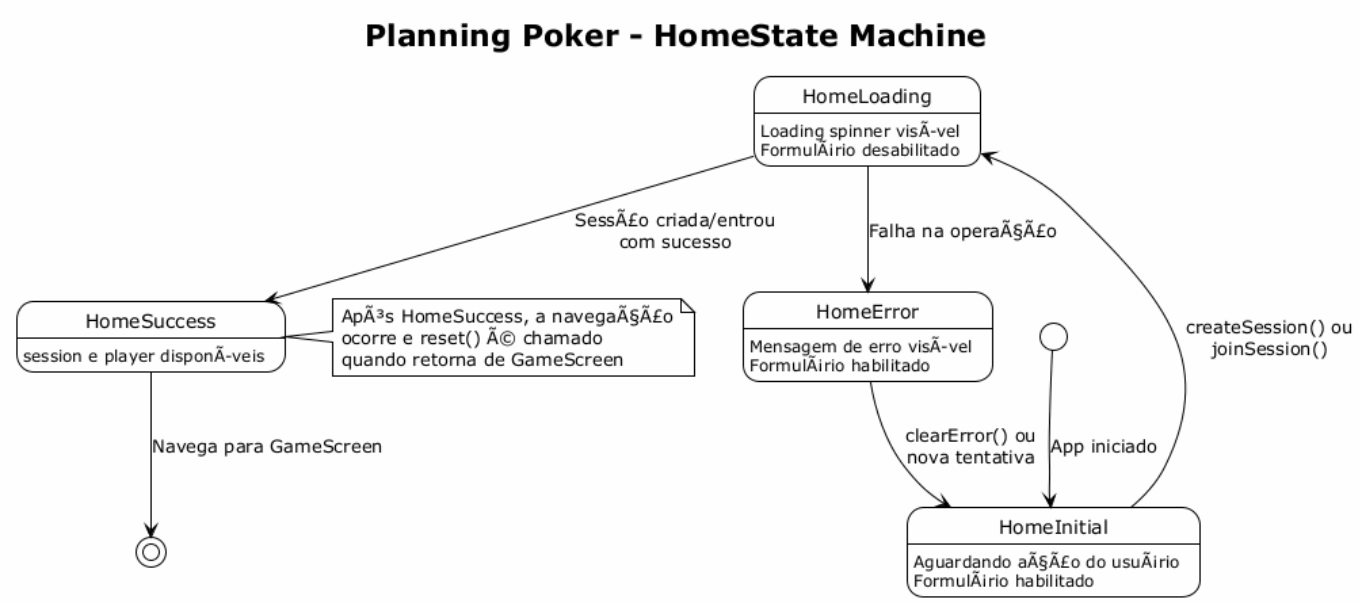
Descrição do Fluxo

1. **Inicialização:** Cada ViewModel se inscreve no Stream da sessão
2. **Player B joga carta:** Dados atualizados no Firebase
3. **Firebase notifica listeners:** Todos os ViewModels recebem atualização
4. **UIs atualizam:** Ambos jogadores veem a mudança
5. **Host revela:** Estado muda para "revealed"
6. **Todos veem cartas:** Sincronização instantânea

---

9. Máquinas de Estado

9.1 Estado da Home Screen



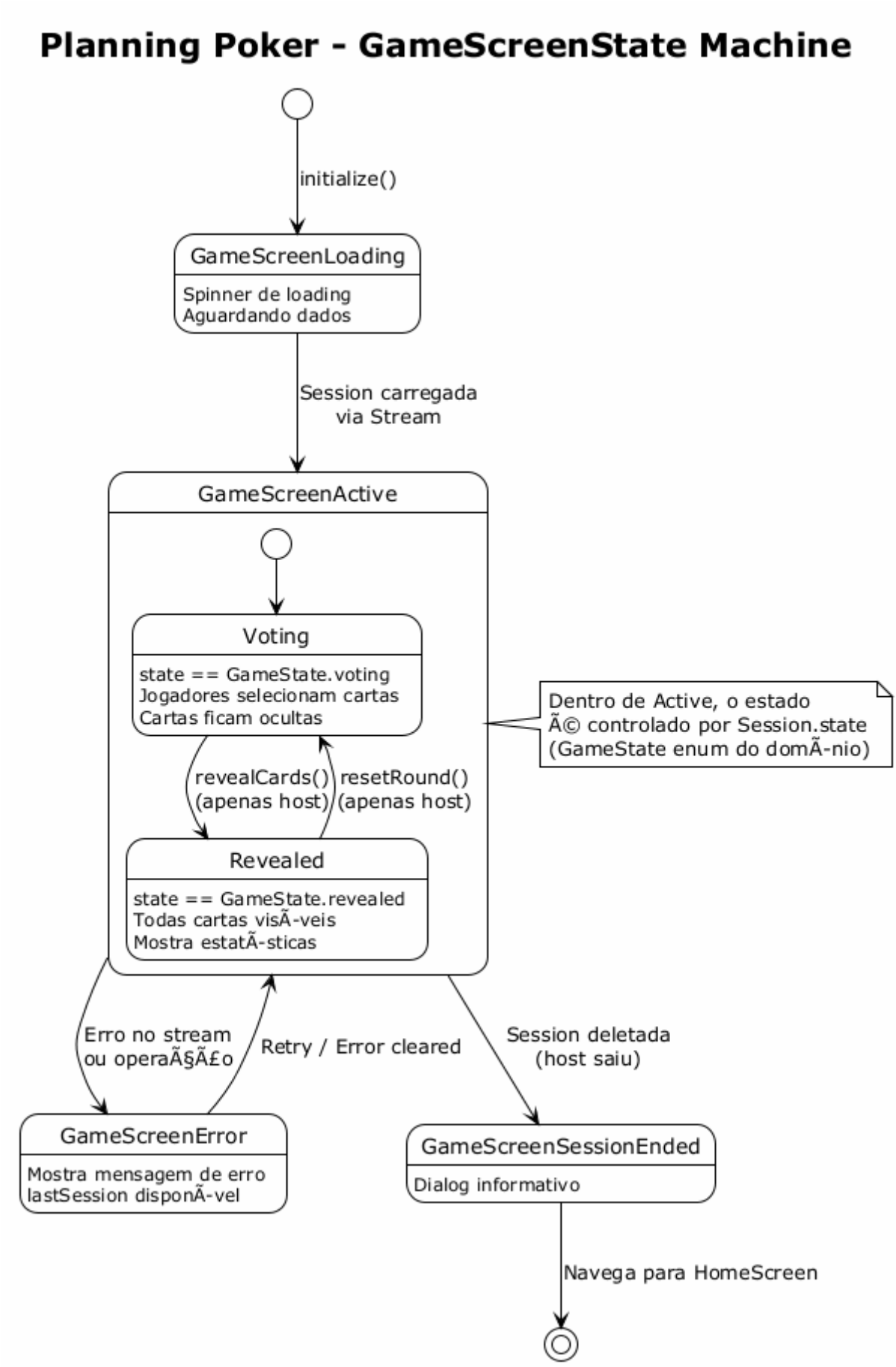
Transições de Estado

Estado Atual	Evento	Próximo Estado
`HomeInitial`	createSession() / joinSession()	`HomeLoading`
`HomeLoading`	Sucesso	`HomeSuccess`
`HomeLoading`	Falha	`HomeError`
`HomeSuccess`	Navegação	(fim)
`HomeError`	clearError() / nova tentativa	`HomeInitial`

Comportamento dos Estados

Estado	UI
`HomeInitial`	Formulário habilitado, aguardando ação
`HomeLoading`	Spinner visível, formulário desabilitado
`HomeSuccess`	Navega para GameScreen
`HomeError`	Mensagem de erro, formulário habilitado

9.2 Estado da Game Screen



Estados Principais

Estado	Descrição

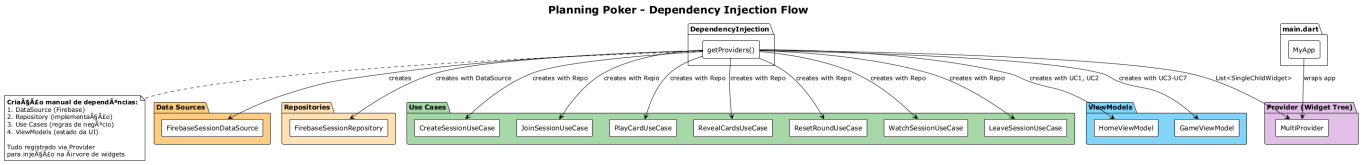
Estado	Descrição
`GameScreenLoading`	Carregando dados da sessão
`GameScreenActive`	Jogo em andamento
`GameScreenError`	Erro na operação
`GameScreenSessionEnded`	Sessão encerrada pelo host

Sub-estados de GameScreenActive

Sub-estado	Condição	Comportamento
`Voting`	state == voting	Jogadores selecionam cartas (ocultas)
`Revealed`	state == revealed	Todas cartas visíveis + estatísticas

10. Injeção de Dependência

10.1 Diagrama de Injeção de Dependência



10.2 Fluxo de Criação

A injeção de dependência é feita manualmente seguindo a ordem:

```
1. DataSource (Firebase)
       ↓
2. Repository (implementação)
       ↓
3. Use Cases (regras de negócio)
       ↓
4. ViewModels (estado da UI)
       ↓
5. Provider (Widget Tree)
```

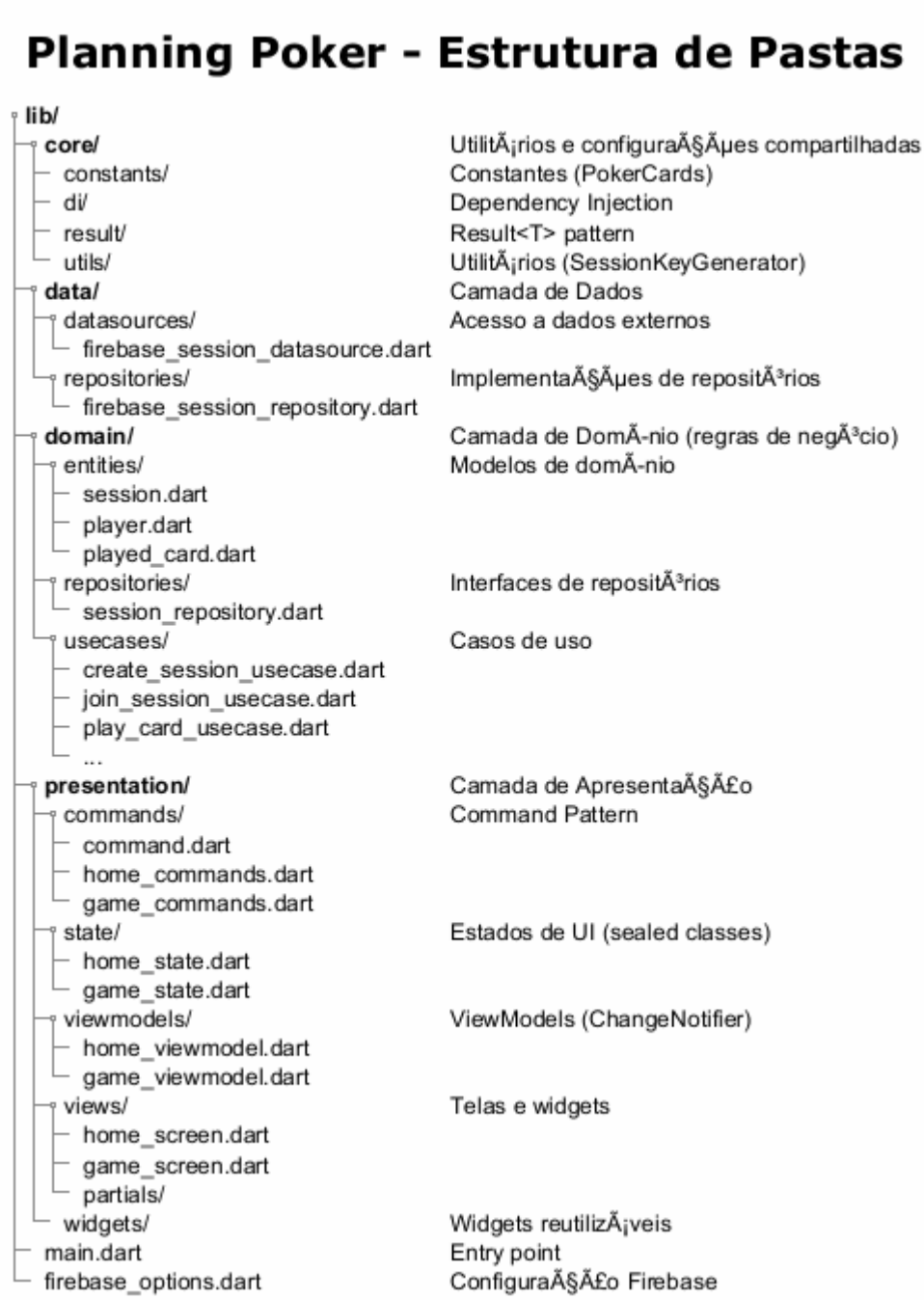
10.3 Código de Configuração

```dart
List<SingleChildWidget> getProviders() {
    // 1. Data Sources
    final dataSource = FirebaseSessionDataSource(
        database: FirebaseDatabase.instance,
    );
```

```
    // 2. Repositories
    final repository = FirebaseSessionRepository(
        dataSource: dataSource,
        uuid: const Uuid(),
    );

    // 3. Use Cases
    final createSessionUseCase = CreateSessionUseCase(repository);
    final joinSessionUseCase = JoinSessionUseCase(repository);
    // ... outros use cases

    // 4. ViewModels via Provider
    return [
        ChangeNotifierProvider(
            create: (_) => HomeViewModel(
                createSessionUseCase: createSessionUseCase,
                joinSessionUseCase: joinSessionUseCase,
            ),
        ),
        ChangeNotifierProvider(
            create: (_) => GameViewModel(
                watchSessionUseCase: watchSessionUseCase,
                playCardUseCase: playCardUseCase,
                // ... outros use cases
            ),
        ),
    ];
}
```

11. Estrutura de Pastas

11.1 Diagrama de Estrutura



11.2 Descrição Detalhada

```
lib/
├── core/                          # Utilitários e configurações
│   ├── constants/                 # Constantes (cartas de poker)
│   │   └── poker_cards.dart
│   ├── di/                        # Injeção de dependência
│   │   └── dependency_injection.dart
│   ├── result/                    # Result pattern
│   │   └── result.dart
│   └── utils/                     # Utilitários
│       └── session_key_generator.dart
│
├── data/                          # Camada de dados
│   ├── datasources/               # Fontes de dados
```

```
│   │   └── firebase_session_datasource.dart
│   └── repositories/              # Implementações
│       └── firebase_session_repository.dart
│
├── domain/                        # Camada de domínio
│   ├── entities/                  # Entidades de negócio
│   │   ├── session.dart
│   │   ├── player.dart
│   │   └── played_card.dart
│   ├── repositories/              # Interfaces
│   │   └── session_repository.dart
│   └── usecases/                  # Casos de uso
│       ├── create_session_usecase.dart
│       ├── join_session_usecase.dart
│       ├── play_card_usecase.dart
│       ├── reveal_cards_usecase.dart
│       ├── reset_round_usecase.dart
│       ├── leave_session_usecase.dart
│       └── watch_session_usecase.dart
│
├── presentation/                  # Camada de apresentação
│   ├── commands/                  # Command pattern
│   │   ├── command.dart           # ObservableCommand base
│   │   ├── home_commands.dart     # Commands da Home
│   │   └── game_commands.dart     # Commands do Game
│   ├── state/                     # Estados da UI
│   │   ├── home_state.dart
│   │   └── game_state.dart
│   ├── viewmodels/                # ViewModels
│   │   ├── home_viewmodel.dart
│   │   └── game_viewmodel.dart
│   ├── views/                     # Telas
│   │   ├── home_screen.dart
│   │   ├── game_screen.dart
│   │   └── partials/              # Componentes de tela
│   └── widgets/                   # Widgets reutilizáveis
│
├── main.dart                      # Entry point
└── firebase_options.dart          # Config Firebase
```

---

12. Padrões de Projeto Utilizados

12.1 MVVM (Model-View-ViewModel)

Descrição: Separa lógica de UI da lógica de negócio.

Implementação:

```
// View observa ViewModel
final viewModel = context.watch<HomeViewModel>();

// ViewModel expõe estado
HomeState get state => _state;

// ViewModel emite novos estados
void _emit(HomeState state) {
    _state = state;
    notifyListeners();
}
```

Componentes:

- **View** (`HomeScreen`, `GameScreen`) - UI declarativa
- **ViewModel** (`HomeViewModel`, `GameViewModel`) - Gerencia estado
- **State** (`HomeState`, `GameScreenState`) - Representa estado

12.2 Command Pattern

Descrição: Encapsula operações como objetos.

Benefícios:

- Encapsulamento de lógica
- Gerenciamento automático de loading/erro
- Reutilização entre ViewModels
- Testabilidade isolada

12.3 Repository Pattern

Descrição: Abstrai acesso a dados.

Benefícios:

- Desacoplamento domínio/dados
- Facilita troca de fonte de dados
- Testabilidade com mocks

12.4 Result Pattern (Either-like)

Descrição: Tratamento explícito de sucesso/falha.

Benefícios:

- Type-safe error handling
- Elimina exceções não tratadas
- Código mais expressivo

12.5 Sealed Classes (Discriminated Unions)

Descrição: Estados mutuamente exclusivos.

Benefícios:

- Exaustividade pelo compilador
- Pattern matching seguro
- Impossível estados inválidos

12.6 Observer Pattern

Descrição: Notificação de mudanças de estado.

Implementação: Via `ChangeNotifier` e `Provider`.

---

13. Guia de Instalação

13.1 Pré-requisitos

- Flutter SDK 3.9+
- Dart SDK 3.9+
- Conta Firebase
- Editor (VS Code recomendado)

13.2 Configuração do Firebase

1. Criar projeto no Firebase Console
2. Configurar Realtime Database
3. Executar:

```
# Instalar FlutterFire CLI
dart pub global activate flutterfire_cli

# Configurar Firebase
flutterfire configure
```

13.3 Instalação

```
# Clone o repositório
git clone https://github.com/DanielBrown1998/planning_poker.git
cd planning_poker

# Instale as dependências
flutter pub get

# Execute o projeto
flutter run
```

13.4 Build para Produção

```
# Android
flutter build apk --release

# iOS
flutter build ios --release

# Web
flutter build web --release

# Windows
flutter build windows --release
```

---

14. Referência de Diagramas

Lista de Diagramas Disponíveis

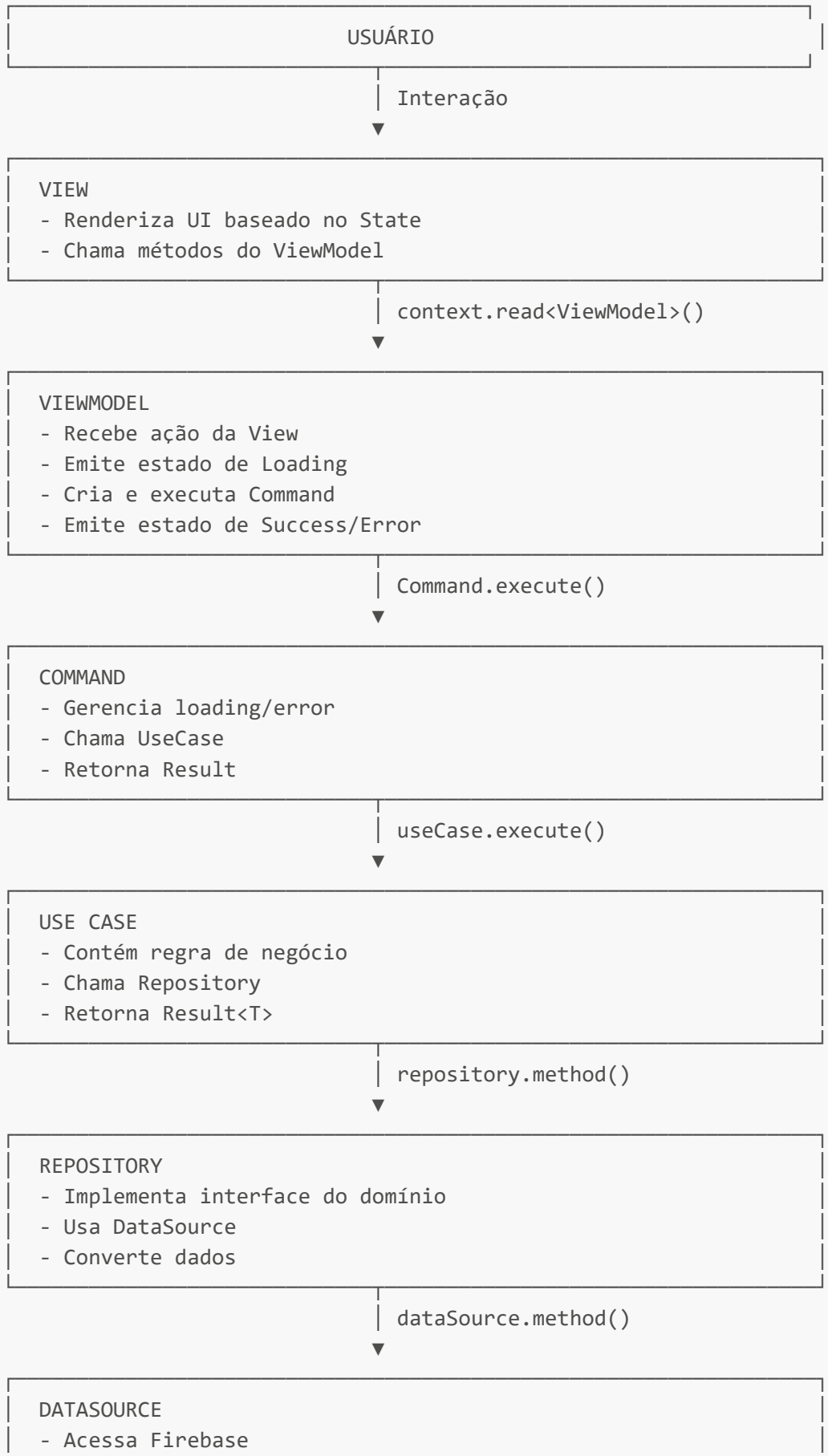
Diagrama	Arquivo	Descrição
Arquitetura Geral	`architecture_overview.png`	Visão geral das camadas
Entidades do Domínio	`domain_entities.png`	Classes de domínio
Camada de Apresentação	`presentation_layer.png`	MVVM e Components
Camada de Dados	`data_layer.png`	Repository e DataSource
Casos de Uso	`use_cases.png`	Use Cases do domínio
Fluxo: Criar Sessão	`flow_create_session.png`	Sequência de criação
Fluxo: Jogar Carta	`flow_play_card.png`	Sequência de jogar
Fluxo: Sync Tempo Real	`flow_realtime_sync.png`	Sincronização
Estado: Home	`state_home.png`	Máquina de estados Home
Estado: Game	`state_game.png`	Máquina de estados Game
Injeção de Dependência	`dependency_injection.png`	Fluxo de DI
Estrutura de Pastas	`folder_structure.png`	Organização do código

Localização

Todos os diagramas estão em:

- **Fonte (PlantUML):** `docs/diagrams/src/*.puml`
- **Imagens (PNG):** `docs/diagrams/images/*.png`

Apêndice A: Fluxo de Dados Completo



```
│  - CRUD operations                                                    │
│  - Streams para tempo real                                           │
└──────────────────────────────────┬──────────────────────────────────┘
                                   │
                                   ▼
                          ┌──────────────────┐
                          │    FIREBASE      │
                          │ Realtime Database│
                          └──────────────────┘
```

Apêndice B: Glossário

Termo	Definição
Clean Architecture	Arquitetura de software que separa preocupações em camadas
MVVM	Model-View-ViewModel, padrão de apresentação
Repository	Padrão que abstrai acesso a dados
Use Case	Caso de uso, representa uma ação do sistema
Entity	Objeto de domínio com identidade
ViewModel	Gerencia estado e lógica de apresentação
Command	Objeto que encapsula uma operação
Sealed Class	Classe que define um conjunto fechado de subtipos
Result Pattern	Padrão para tratamento de sucesso/falha
Provider	Biblioteca de gerenciamento de estado Flutter
Stream	Sequência assíncrona de dados
Firebase	Plataforma de desenvolvimento de apps Google

Documento gerado automaticamente
Planning Poker v1.0.0