### Análise Arquitetural do Projeto

### Análise dos Princípios SOLID

Os princípios SOLID foram a bússola para a nossa arquitetura. Veja como cada um foi aplicado:

##### S - Princípio da Responsabilidade Única (SRP)

*Cada classe tem uma, e apenas uma, razão para mudar.*

* **Exemplo 1: Repositórios.** A classe RacaRepositoryImpl tem a única responsabilidade de gerenciar a persistência de entidades Raça. Ela não sabe como salvar um Personagem ou uma Arma. Se a forma de armazenar raças mudar, esta é a única classe que precisará ser alterada.
* **Exemplo 2: DatabaseHelper.** Sua única responsabilidade é gerenciar a conexão e o esquema do banco de dados (criação e migração de tabelas). Ele não tem conhecimento algum sobre as entidades de negócio.
* **Exemplo 3: Factories.** A PersonagemFactoryImpl tem a única responsabilidade de construir um objeto Personagem válido, encapsulando a lógica complexa de buscar suas dependências. Ela não o salva no banco; ela delega essa responsabilidade ao repositório.

##### O - Princípio Aberto/Fechado (OCP)

*O software deve ser aberto para extensão, mas fechado para modificação.*

* **Exemplo Principal: Padrão Strategy.** Nossa implementação do Strategy com a classe Habilidade é o exemplo perfeito. O código que executa uma habilidade (um futuro "Gerenciador de Batalha") está fechado para modificação. Para adicionar uma nova habilidade com um efeito completamente novo (ex: HabilidadeDeVenenoModel), nós apenas criamos uma nova classe que herda de Habilidade (aberto para extensão). Não precisamos alterar nenhuma linha do código que consome as habilidades.

##### L - Princípio da Substituição de Liskov (LSP)

*Subclasses devem ser substituíveis por suas classes base sem quebrar o sistema.*

* **Exemplo Principal: Combatente.** Tanto Personagem quanto Inimigo herdam de Combatente. Qualquer parte do nosso sistema que espera receber um Combatente (como a lista de membros de um Grupo<Combatente>) pode receber uma instância de Personagem ou Inimigo sem distinção e sem causar erros. O código cliente não precisa saber o tipo concreto.

##### I - Princípio da Segregação de Interfaces (ISP)

*Clientes não devem ser forçados a depender de interfaces que não utilizam.*

* **Exemplo Principal: Nossos Repositórios.** A nossa decisão de criar interfaces específicas como IRacaRepository e IPersonagemRepository em vez de uma única interface IRepository genérica é a aplicação direta deste princípio. Uma parte do código que precisa apenas listar raças disponíveis dependerá *somente* da interface IRacaRepository, não sendo forçada a conhecer os métodos de save de um personagem ou de delete de um grupo.

##### D - Princípio da Inversão de Dependência (DIP)

*Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.*

* **Exemplo Principal: Injeção de Dependência.** Nossas classes de alto nível (os repositórios) não dependem das implementações concretas de suas dependências. Por exemplo, PersonagemRepositoryImpl depende da **abstração** IHabilidadeRepository, não da classe concreta HabilidadeRepositoryImpl. Nós "injetamos" a dependência através do construtor. Isso inverte o controle e permite, por exemplo, injetar um repositório falso (Mock) nos testes, provando a flexibilidade da arquitetura.

### Análise dos Padrões GRASP

GRASP nos ajuda a atribuir responsabilidades de forma lógica.

* **Information Expert (Especialista da Informação):** A responsabilidade é atribuída à classe que tem a informação necessária para cumpri-la.
  + **Exemplo:** A classe Combatente é a especialista em seus próprios pontos de vida (vidaAtual, vidaMax). Portanto, a responsabilidade e a lógica de como o dano e a cura são aplicados (receberDano, receberCura) residem nela.
  + **Exemplo 2:** Cada classe de habilidade concreta (HabilidadeDeDanoModel) é a especialista sobre como seu próprio efeito deve ser calculado e aplicado, por isso ela contém o método execute.
* **Creator (Criador):** Define quem deve ser responsável por criar um objeto.
  + **Exemplo:** O padrão **Factory Method** (PersonagemFactoryImpl) é um Criador especializado. Ele foi designado para criar objetos Personagem porque ele agrega e possui acesso a todas as informações necessárias para isso (os repositórios de Raca e Classe).
* **Low Coupling & High Cohesion (Baixo Acoplamento e Alta Coesão):**
  + **Baixo Acoplamento:** Nossa arquitetura em camadas (Domain, Data) e o uso de interfaces de repositório garantem que as camadas sejam o mais independentes possível. A camada de Domínio não sabe nada sobre o sqflite.
  + **Alta Coesão:** Nossas classes têm responsabilidades focadas. ArmaRepositoryImpl só lida com armas. HabilidadeDeDanoModel só lida com a lógica de causar dano. Isso as torna fáceis de entender e manter.
* **Controller (Controlador):** Atua como um intermediário entre a UI e o domínio.
  + **Exemplo:** Embora não tenhamos construído a UI, o CrudTestRunner age como um Controlador nos nossos testes. Ele recebe a "intenção" (ex: "testar o CRUD de personagem") e coordena as chamadas aos diferentes repositórios e factories para executar a tarefa. Em uma aplicação real, este papel será das classes ViewModel ou Provider.

### Análise dos Padrões GoF Implementados

1. **Factory Method:** Implementado através da interface IFichaFactory e suas classes concretas (PersonagemFactoryImpl, InimigoFactoryImpl). Ele encapsula a lógica complexa de criação de fichas, recebendo parâmetros simples (como IDs) e orquestrando a busca de dependências nos repositórios para montar um objeto completo e válido.
2. **Strategy:** Implementado com a classe abstrata Habilidade como a interface da Estratégia e seu método execute. Classes como HabilidadeDeDanoModel e HabilidadeDeCuraModel são as Estratégias Concretas, cada uma fornecendo um algoritmo diferente para o método execute. Isso permite que o código cliente use uma habilidade sem conhecer sua lógica interna.
3. **Composite:** Implementado com a interface AlvoDeAcao como o "Componente". A classe Combatente atua como a "Folha" (objeto individual) e a classe Grupo atua como o "Compósito" (coleção de objetos). Ambas implementam a mesma interface, permitindo que o cliente (como o método execute de uma habilidade) trate um alvo individual e um grupo de alvos de maneira uniforme e transparente.
4. **Singleton:** Este padrão foi implementado na classe DatabaseHelper para garantir que exista uma única e centralizada instância de conexão com o banco de dados durante toda a execução do aplicativo. A implementação utiliza um construtor privado (\_privateConstructor) para impedir a criação externa de novos objetos e um ponto de acesso estático e global (DatabaseHelper.instance) que retorna sempre a mesma instância. Isso evita problemas de concorrência e garante que todos os repositórios e partes do sistema conversem com o mesmo banco de dados de forma consistente e segura.
5. **Prototype:** Este padrão foi implementado na entidade Inimigo através da interface IPrototype, que define um método clone(). A implementação do método clone cria uma nova instância de Inimigo, copiando os atributos do objeto original (o protótipo), mas garantindo a atribuição de um novo e único ID. Isso permite a replicação rápida e eficiente de fichas de inimigos, ideal para cenários onde o mestre do jogo precisa criar múltiplos inimigos idênticos para uma batalha sem ter que recriar cada um do zero, complementando o Factory Method que é usado para a criação inicial do protótipo.
6. **Observer:** Este padrão é fundamental para a arquitetura de UI reativa do aplicativo e é implementado através da combinação das classes ChangeNotifier e dos widgets Consumer (ou do método context.watch) do pacote Provider.

* **Subject (ou Observable):** As classes de ViewModel (como PersonagensViewModel, InimigosViewModel, etc.) atuam como os "Sujeitos". Elas herdam de ChangeNotifier, mantêm o estado (por exemplo, a lista de personagens) e notificam os interessados sobre quaisquer mudanças através da chamada ao método notifyListeners().
* **Observers:** Os widgets da camada de apresentação (UI), como o ListView.builder dentro de um widget Consumer na GerenciarPersonagensPage, atuam como os "Observadores". Eles se inscrevem para ouvir as notificações de um ViewModel específico.

O fluxo funciona da seguinte maneira: quando uma ação do usuário (como deletar um personagem) aciona um método no PersonagensViewModel, o ViewModel atualiza sua lista interna de personagens e, em seguida, chama notifyListeners(). Isso transmite uma notificação para todos os widgets "observadores" registrados. O Consumer, ao receber essa notificação, reconstrói sua árvore de widgets, fazendo com que a ListView seja redesenhada com a lista de personagens atualizada, refletindo a mudança na UI de forma automática e eficiente.

A grande vantagem aqui é o **baixo acoplamento**: o ViewModel (Subject) não precisa conhecer nenhum detalhe sobre os widgets (Observers) que o estão escutando, ele apenas anuncia que seu estado mudou. Isso cria uma separação de responsabilidades limpa e um fluxo de dados unidirecional que é fácil de manter e depurar.