Documento de Arquitetura de Software

| **Grupo** |
| --- |
| Gabriel Gomes |
| Ramon Pedro |
| Pedro Umpierre |
| Lucas Vitorino |
| Frederico Antunes |

| **Objetivo deste Documento** |
| --- |
| Este documento tem como objetivo descrever as principais decisões de projeto tomadas pela equipe de desenvolvimento e os critérios considerados durante a tomada destas decisões. Suas informações incluem aparte de *hardware* e *software* do sistema. |

**Sumário**

1. [**INTRODUÇÃO 1**](#_heading=h.c7b8hnp79xey)
2. [**DESCRIÇÃO DOS OBJETIVOS, RESTRIÇÕES E REQUISITOS ARQUITETURAIS 4**](#_heading=h.4obl84l3dr5m)
3. [**JUSTIFICATIVA 6**](#_heading=h.269o1m2tixd3)
4. [**ANÁLISE DO USO DE PADRÕES SOLID, GRASP E GOF 8**](#_heading=h.j13cm514332v)
5. [**DIAGRAMA DE CASOS DE USO 11**](#_heading=h.wyozh4ocw2g4)
6. [**MODELO CONCEITUAL 13**](#_heading=h.rq3rpk50vjp6)
7. [**DIAGRAMA DO PADRÃO ARQUITETURAL 13**](#_heading=h.1m6ttm3tz5f)
8. [**DIAGRAMA DE CLASSES 14**](#_heading=h.2h9j1t2qtrca)
9. [**DIAGRAMAS DE SEQUÊNCIA 15**](#_heading=h.pp7r3rwg7lr6)
10. **DIAGRAMAS DE SEQUÊNCIA DO SISTEMA…………………………………18**
11. **MODELO DE CARACTERÍSTICAS……………………………………………..20**

# 1 - INTRODUÇÃO

## Finalidade

Este documento fornece uma visão arquitetural abrangente do sistema Beluken - RPG Battle Manager, usando diversas visões de arquitetura para **representar** diferentes aspectos do sistema. O objetivo deste documento é capturar e comunicar as decisões arquiteturais significativas que foram tomadas em relação ao sistema.

## Escopo

O sistema proposto é um aplicativo multiplataforma, desenvolvido em Flutter, com suporte para os sistemas operacionais Windows e Android. Seu objetivo principal é oferecer uma ferramenta completa para a criação, gerenciamento e simulação de batalhas em cenários de RPG. Inspirado em sistemas tradicionais de jogos de interpretação de papéis, o aplicativo se propõe a apresentar um sistema próprio de combate, com foco na fluidez e dinamismo das interações durante os confrontos.

O aplicativo permitirá ao usuário criar fichas detalhadas tanto de personagens quanto de inimigos. Essas fichas contém informações como raça, classe, nível inicial, atributos distribuídos, equipamentos e uma descrição geral. No caso dos inimigos, será possível definir também o tipo, além dos outros dados mencionados. Essas informações servirão como base para as batalhas e para o gerenciamento dos grupos que participarão delas.

O sistema fornecerá funcionalidades específicas para o gerenciamento de personagens e inimigos, permitindo ao usuário editar, duplicar ou excluir fichas já criadas. Com base nesses cadastros, será possível criar grupos de personagens com até quatro membros, e grupos de inimigos, que poderão conter um ou mais monstros. Esses grupos poderão ser salvos, modificados ou reutilizados conforme o interesse do usuário.

Para iniciar uma batalha, o usuário deverá selecionar um grupo de personagens e um grupo de inimigos. Esses passos fazem parte do processo de configuração e são essenciais para que o combate seja iniciado de forma adequada.

Durante a batalha, o sistema assumirá o papel de gerenciador do combate, permitindo ao usuário conduzir o andamento da luta por meio do registro das ações realizadas por cada personagem ou inimigo. O sistema fará os cálculos e as operações associadas ao combate e apresentará feedbacks visuais e textuais que reflitam o estado atual do combate. O histórico das ações será mantido para fins de acompanhamento e tomada de decisões táticas.

Todas as informações geradas, personagens, inimigos, grupos e etc serão armazenadas localmente em banco de dados, garantindo a persistência dos dados entre sessões de uso. A escolha da base de dados será compatível com o Flutter, sendo recomendadas soluções como SQLite.

O aplicativo será desenvolvido com uma interface amigável e responsiva, adequada tanto ao uso com mouse e teclado no Windows quanto ao toque em dispositivos Android. A navegação entre as seções será intuitiva, facilitando o acesso às funcionalidades de criação, configuração e gerenciamento.

Embora o sistema de combate seja baseado em estruturas tradicionais de RPG, ele contará com regras e mecânicas próprias, pensadas para tornar as batalhas mais rápidas e acessíveis. Essas regras poderão ser estendidas futuramente, incluindo variações, personalizações ou modos alternativos. O sistema é projetado com flexibilidade para permitir futuras extensões de funcionalidades e regras dentro do ambiente local.

## Levantando Funcionalidades:

O sistema proposto é um aplicativo desenvolvido em Flutter para Windows e Android com a função de gerenciar batalhas entre grupos de fichas de personagens e grupos de monstros. No aplicativo o usuário deve ser capaz de:

* Criar e gerenciar grupos de fichas de personagens que podem variar de 1 membro até 4 membros.
* Criar fichas de personagens para serem colocados nos grupos.
* Criar fichas de inimigos. para serem colocados nos grupos.
* Disponibilizar alta customização estilo RPG para fichas de personagens na hora da criação e posteriormente.
* Disponibilizar alta customização estilo RPG para fichas de monstros na hora da criação e posteriormente.
* Manter os grupos salvos em um banco de dados local.
* Manter as fichas de personagens salvas em um banco de dados local.
* Manter as fichas de inimigos salvas em um banco de dados local.
* Permitir a construção de batalhas ao selecionar grupos de personagens e inimigos
* Gerenciar as batalhas.

# 2 - [DESCRIÇÃO DOS OBJETIVOS, RESTRIÇÕES E REQUISITOS ARQUITETURAIS](#_heading=h.u1anf5u6d8qp)

Esta seção delineia as metas de qualidade que a arquitetura visa alcançar (Objetivos), as condições e decisões preexistentes que devem ser respeitadas (Restrições), e as capacidades específicas que a arquitetura deve prover para atender às necessidades do sistema (Requisitos).

* **Restrições Arquiteturais**
  + **Restrição 1: Suporte Multiplataforma com Código Reutilizável**
    - **Descrição:** O projeto exige a entrega de uma aplicação funcional em, no mínimo, duas plataformas distintas (Windows e Android) a partir de um esforço de desenvolvimento unificado. É uma premissa do projeto otimizar os recursos de desenvolvimento e garantir a consistência da experiência do usuário entre as plataformas, evitando a criação e manutenção de bases de código separadas.
    - **Decisão Arquitetural Resultante:** Para atender a esta restrição de eficiência e portabilidade, **foi decidido** utilizar o framework Flutter, que permite o desenvolvimento a partir de uma única base de código em Dart.
  + **Restrição 2: Operação Independente de Conectividade**
    - **Descrição:** O sistema deve ser totalmente funcional em cenários sem acesso à internet. O caso de uso principal envolve o gerenciamento de sessões de RPG de mesa, que frequentemente ocorrem em ambientes onde a conectividade pode ser instável ou inexistente. A capacidade de operar offline é, portanto, um requisito operacional inegociável.
    - **Decisão Arquitetural Resultante:** Para garantir a funcionalidade offline, **foi decidido** que todo o armazenamento de dados (fichas, grupos, batalhas) será estritamente local, no dispositivo do usuário, sem dependência de um backend ou serviços na nuvem.
  + **Restrição 3: Escopo de Lançamento Focado**
    - **Descrição:** Para viabilizar a entrega e validação do produto dentro do cronograma do projeto, o escopo da versão inicial está focado e limitado. A prioridade é validar as funcionalidades centrais em um conjunto definido de ambientes antes de considerar uma expansão mais ampla.
    - **Decisão Arquitetural Resultante:** Em função desta restrição de escopo, **foi decidido** que o lançamento inicial e os testes se concentrarão nas plataformas Windows (desktop) e Android (móvel).
* **Objetivos Arquiteturais:**
  + **Manutenibilidade e Extensibilidade:** Alcançar uma arquitetura de software que seja **altamente fácil de manter e estender**. Isso significa que modificações, correções de bugs e a adição de novas funcionalidades locais (como novas regras de RPG, tipos de personagens ou mecânicas de combate) devem ser possíveis com impacto mínimo nos componentes existentes.
  + **Testabilidade:** Garantir um **alto grau de testabilidade** para todos os componentes críticos do sistema. A arquitetura deve facilitar a criação de testes unitários para a lógica de negócios (regras de combate, criação de fichas) e testes de integração que validem as interações entre as camadas.
  + **Flexibilidade para Evolução Local:** Prover **flexibilidade arquitetural** para permitir a evolução do sistema em termos de regras de jogo, personalizações e modos de operação, mantendo o foco em funcionalidades que operem localmente no dispositivo do usuário.
  + **Desempenho Responsivo:** Assegurar que o sistema apresenta **baixo tempo de resposta e fluidez** nas interações do usuário, especialmente durante operações críticas como a simulação de batalhas, cálculos de atributos e navegação entre telas.
  + **Usabilidade e Experiência do Usuário:** Facilitar, através de uma estrutura bem definida, a construção de uma interface de usuário que seja **intuitiva, agradável e eficiente**, adaptando-se adequadamente a diferentes contextos de uso (desktop e mobile).
* **Requisitos Arquiteturais:**
  + **Interface Adaptativa:** O sistema **deve** suportar layouts de interface que se adaptem de forma responsiva a diferentes tamanhos de tela, densidades de pixel e orientações (retrato/paisagem), visando uma experiência consistente em dispositivos desktop (Windows) e móveis (Android), e potencialmente web e iOS no futuro.
  + **Gerenciamento Eficiente de Dados Locais:** A arquitetura **deve** ser capaz de gerenciar eficientemente um volume crescente de dados do usuário (fichas de personagens e inimigos, grupos, cenários, históricos de batalha) armazenados localmente em um banco de dados SQLite, garantindo a integridade e a rápida recuperação dessas informações.
  + **Segurança dos Dados Locais (Opcional):** O sistema **deve** permitir a futura integração de mecanismos de criptografia para os dados armazenados localmente, caso a proteção da privacidade das informações do usuário contra acesso não autorizado no dispositivo seja identificada como uma necessidade.
  + **Navegação Clara e Consistente:** A arquitetura **deve** suportar um sistema de navegação claro, previsível e guiado, permitindo que os usuários acessem todas as funcionalidades principais com facilidade.
  + **Suporte a Temas e Diretrizes de Design:** O sistema **deve** permitir a aplicação de temas visuais (como claro e escuro) e, na construção dos componentes de interface, considerar as diretrizes de design Material Design (para Android, Windows, Web) e Cupertino (para iOS) para uma integração visual adequada às respectivas plataformas.
  + **Aplicação dos Padrões MVC e Repository:** A estrutura da aplicação **deve** ser organizada seguindo o padrão Model-View-Controller (MVC) para a separação de responsabilidades entre apresentação, lógica de controle e dados. Adicionalmente, **deve** ser utilizado o padrão Repository para abstrair e gerenciar o acesso à camada de persistência de dados local.

# 3 - JUSTIFICATIVA

**Manutenibilidade:** A adoção do padrão MVC (Model-View-Controller) permite a separação clara entre a lógica de negócios, a interface do usuário e o controle da aplicação. Essa separação facilita a localização de erros, a modificação de funcionalidades existentes e a introdução de novas funcionalidades, uma vez que as camadas estão desacopladas. Isso reduz a complexidade do código e melhora a legibilidade e organização do sistema. Além disso, o uso do padrão Repository contribui significativamente para a manutenibilidade ao desacoplar a lógica de acesso a dados do restante da aplicação. Isso permite que o mecanismo de persistência local (SQLite) seja gerenciado ou até mesmo reestruturado internamente sem impactar diretamente outras partes do sistema que consomem os dados.

**Capacidade de Crescimento Local (Escalabilidade):** O sistema foi concebido com foco na capacidade de lidar com um aumento no volume de dados gerados pelo usuário (fichas, grupos, históricos de batalha) de forma eficiente no ambiente local. O uso de Flutter com código base único permite escalar a aplicação para diferentes plataformas (Android, iOS, Windows e web), garantindo consistência funcional e estética em todas elas, caso o escopo de plataformas seja expandido no futuro. A arquitetura modular também visa facilitar a adição de novas funcionalidades locais.

**Portabilidade:** A escolha pelo framework Flutter com Dart permite a compilação do mesmo código-fonte para múltiplas plataformas, facilitando a manutenção de versões consistentes do aplicativo e reduzindo custos de desenvolvimento. Isso garante portabilidade entre dispositivos móveis, desktop e navegadores modernos, atendendo à restrição de compatibilidade com Android, iOS, Windows, Chrome, Firefox e Safari.

**Testabilidade:** A estruturação do sistema segundo os princípios do MVC e o uso do padrão Repository (com injeção de dependência) favorecem a criação de testes automatizados, tanto unitários quanto de integração. Cada componente da lógica de negócios pode ser testado isoladamente da camada de persistência, mockando-se as interfaces do Repository, garantindo que as regras de negócio, cálculos de combate e manipulação de dados estejam corretos.

**Segurança:** O sistema armazenará todos os dados localmente. Caso seja necessário, a arquitetura permite a integração de mecanismos de criptografia para proteger os dados do usuário armazenados no dispositivo.

**Usabilidade:** A arquitetura propõe o uso de layouts adaptativos e navegação guiada com suporte a temas claro e escuro, o que contribui para uma experiência de usuário consistente, intuitiva e acessível em diferentes dispositivos. A divisão clara das responsabilidades no MVC facilita também a construção de interfaces responsivas e bem organizadas.

**Desempenho:** O uso de banco de dados local leve, como SQLite, e o gerenciamento local de estados e dados garantem tempo de resposta baixo para operações críticas como atualização de atributos durante as batalhas. O padrão Repository pode ser implementado de forma eficiente para não introduzir sobrecarga perceptível, enquanto organiza o acesso aos dados.

**Flexibilidade e Evolução:** O uso do padrão Repository e a organização modular baseada em MVC garantem que o sistema possa ser expandido futuramente com novas funcionalidades *locais*. O Repository, em particular, oferece flexibilidade para adaptar ou evoluir o mecanismo de persistência de dados (atualmente SQLite) ou introduzir novas formas de gerenciamento de dados *localmente*, sem exigir reestruturações profundas na lógica de negócios.

# 4 - ANÁLISE DO USO DE PADRÕES SOLID, GRASP E GOF

### **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.

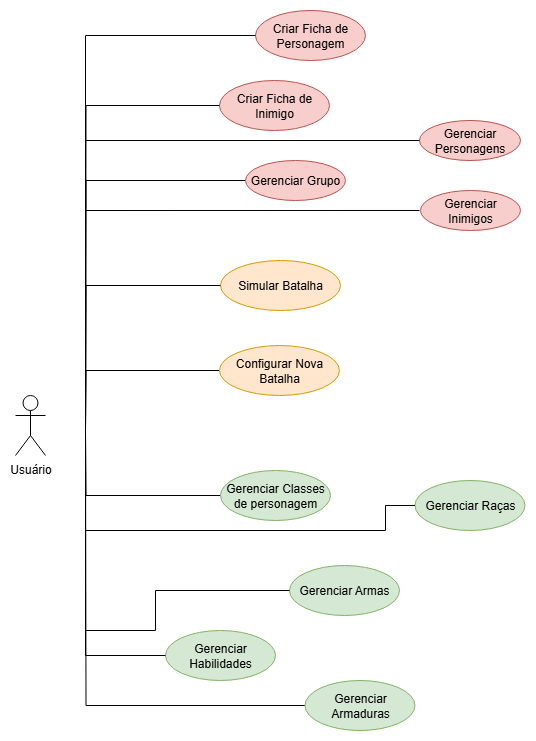
* **Subject (ou Observable):** As classes de ViewModel (como PersonagensViewModel, InimigosViewModel, etc.) atuam como os "Sujeitos". Elas herdam de ChangeNotifier, mantêm o estado (por exemplo, a lista de personagens) e notificam os interessados sobre quaisquer mudanças através da chamada ao método 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.

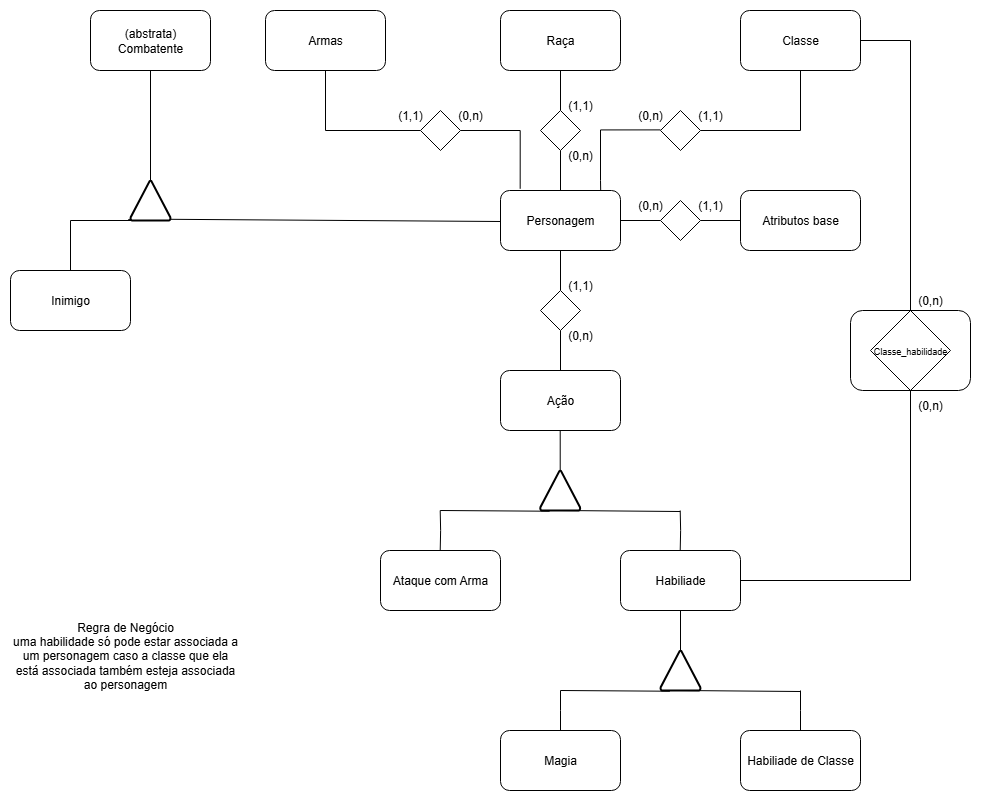
A grande vantagem aqui é o **baixo acoplamento**: o ViewModel (Subject) não precisa conhecer nenhum detalhe sobre os widgets (Observers) que o estão escutando, ele apenas anuncia que seu estado mudou. Isso cria uma separação de responsabilidades limpa e um fluxo de dados unidirecional que é fácil de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

# 

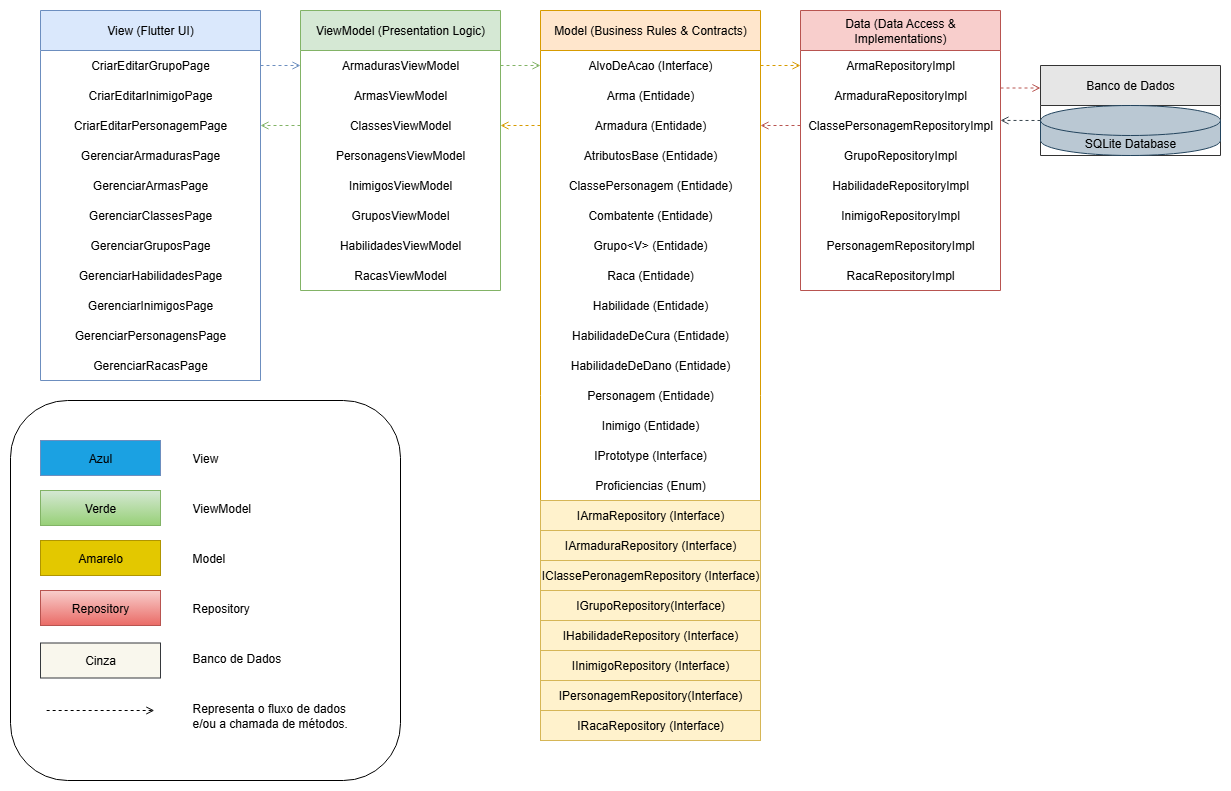
# 5 - DIAGRAMA DE CASOS DE USO



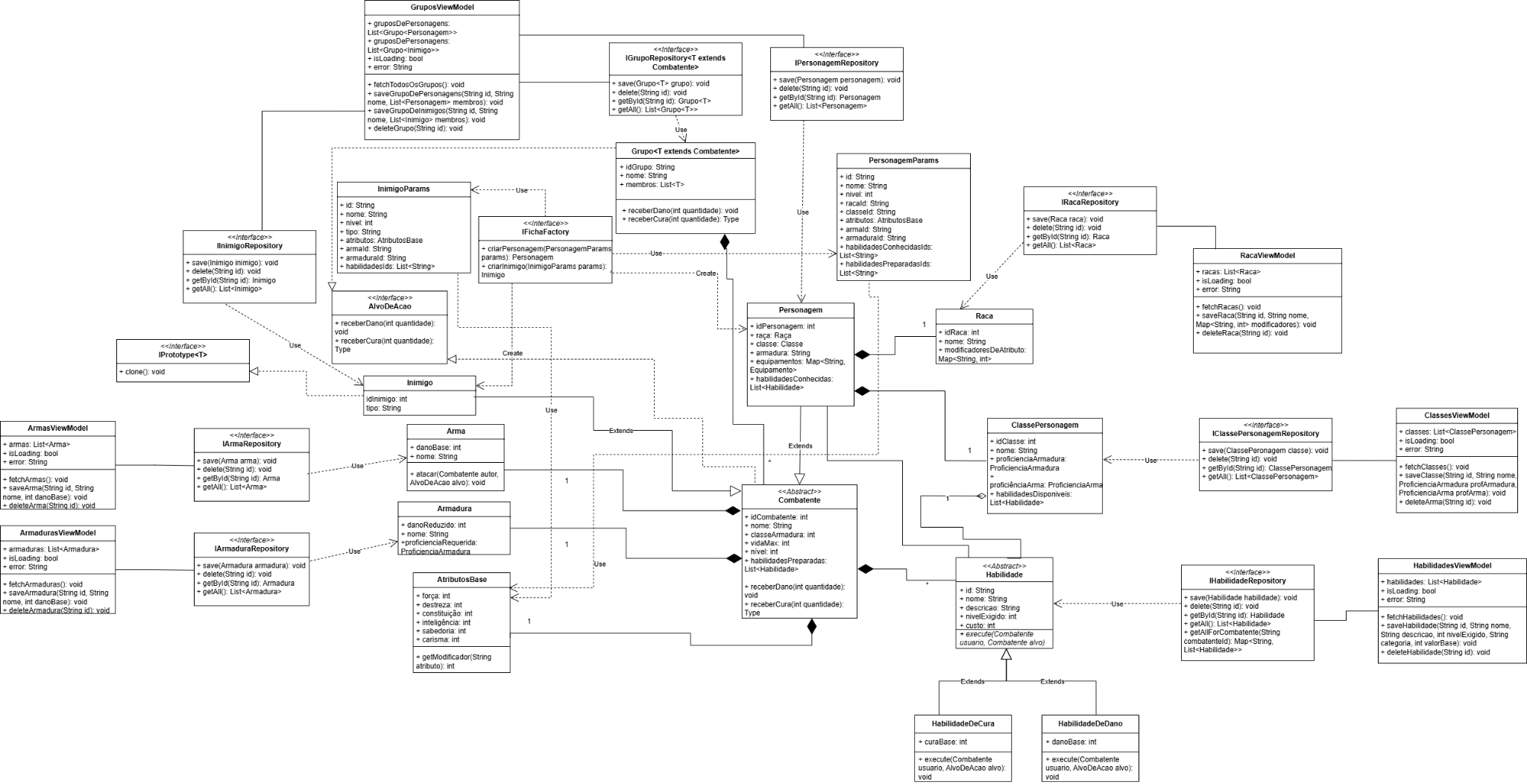
# 6 - MODELO CONCEITUAL



# 7 - DIAGRAMA DO PADRÃO ARQUITETURAL

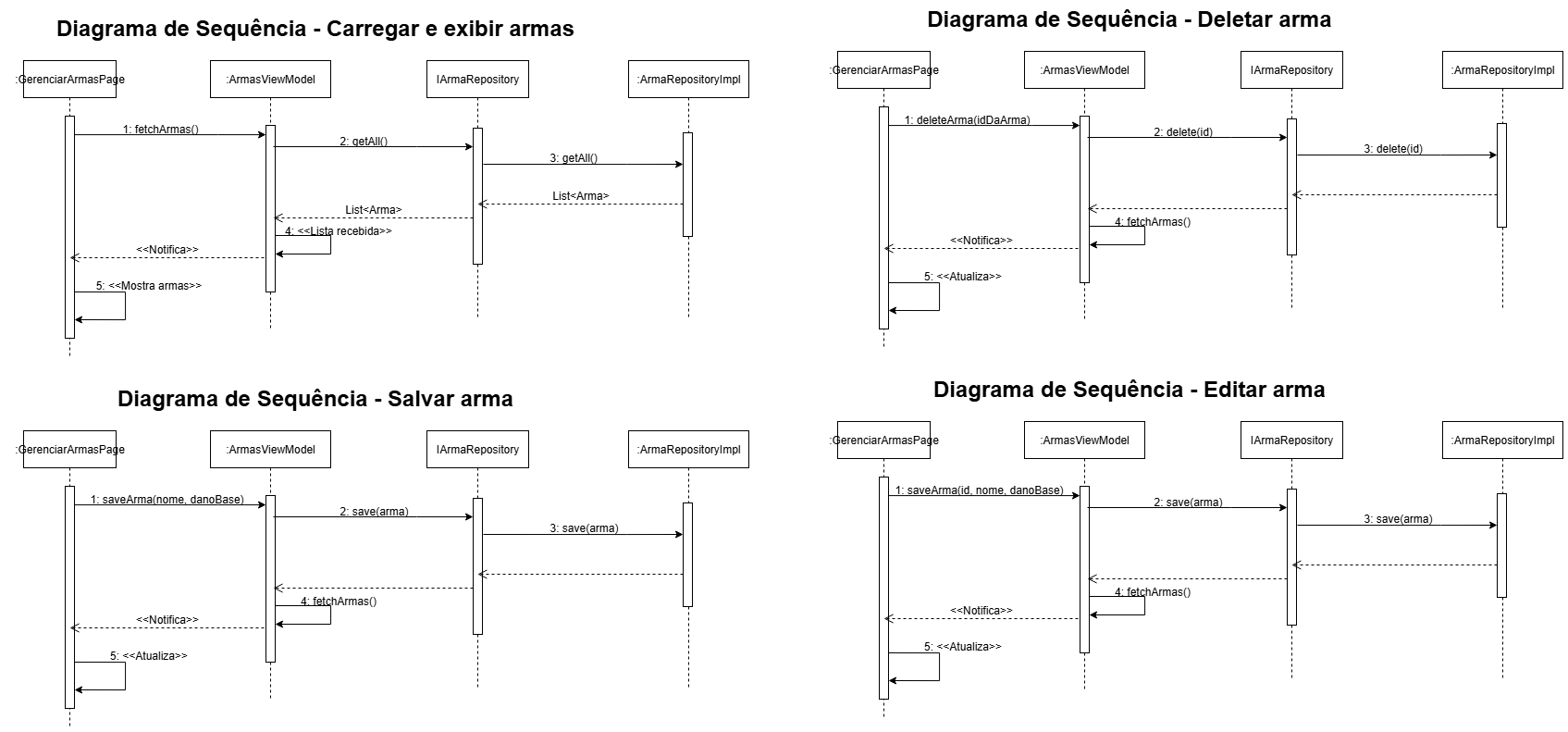


# 8 - DIAGRAMA DE CLASSES

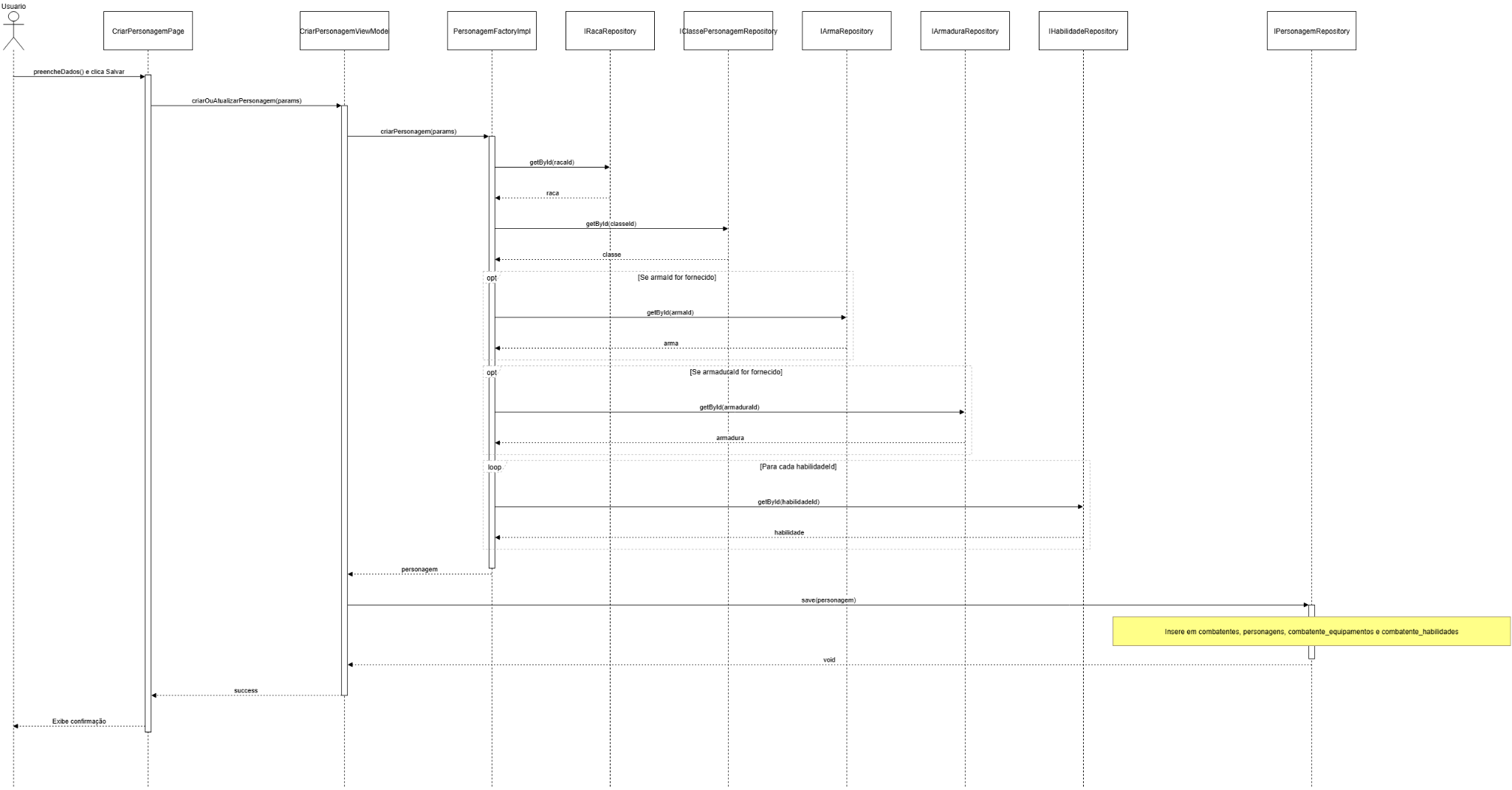


# 9 - DIAGRAMAS DE SEQUÊNCIA

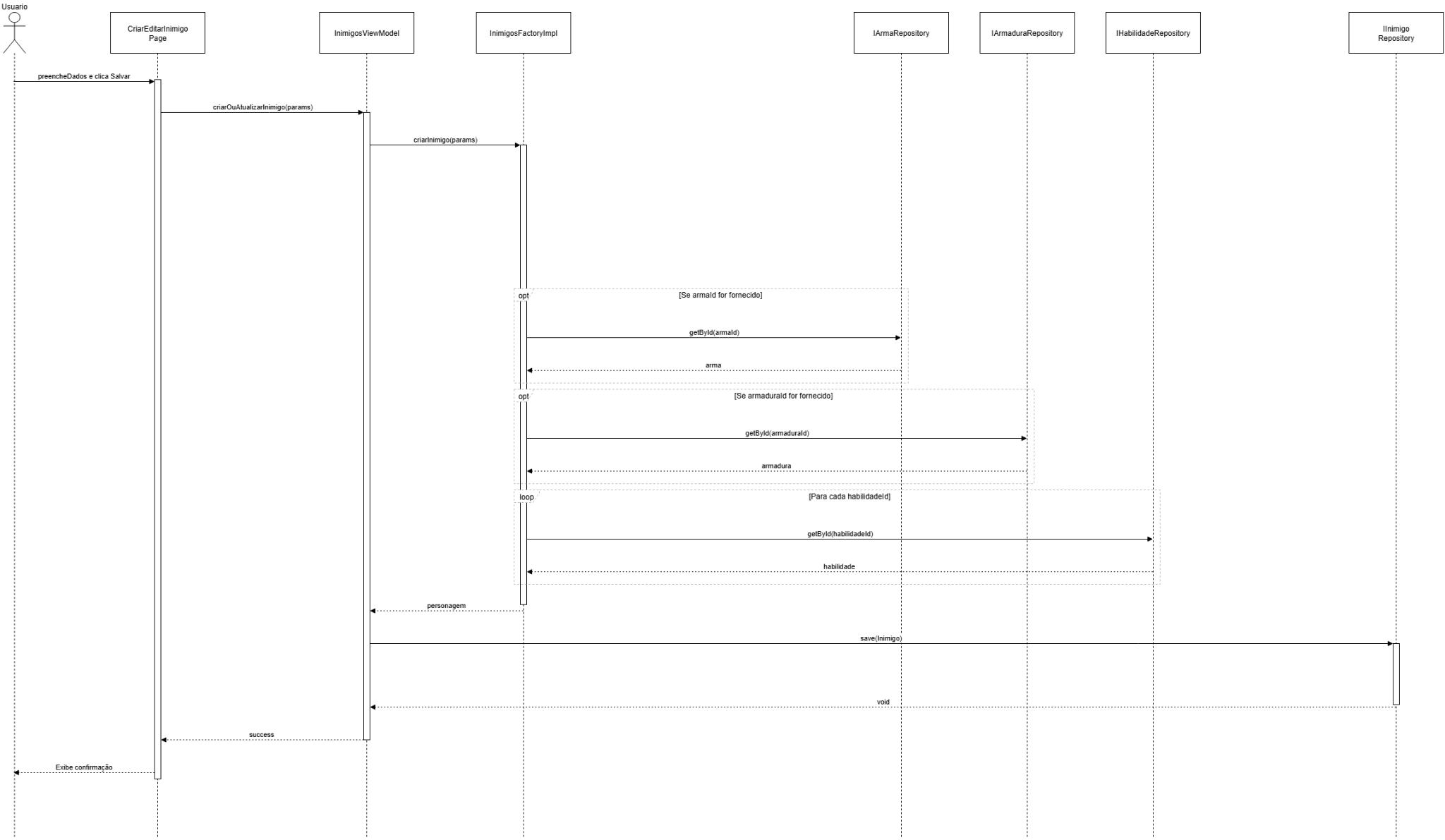
Gerenciar Armas



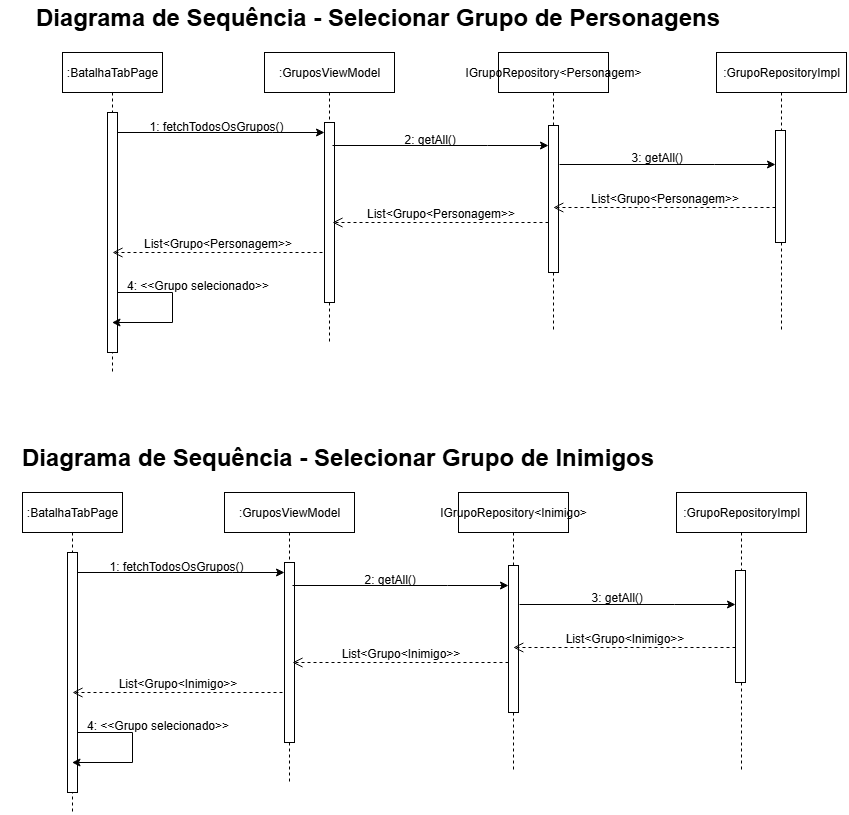
Criar ficha de personagem



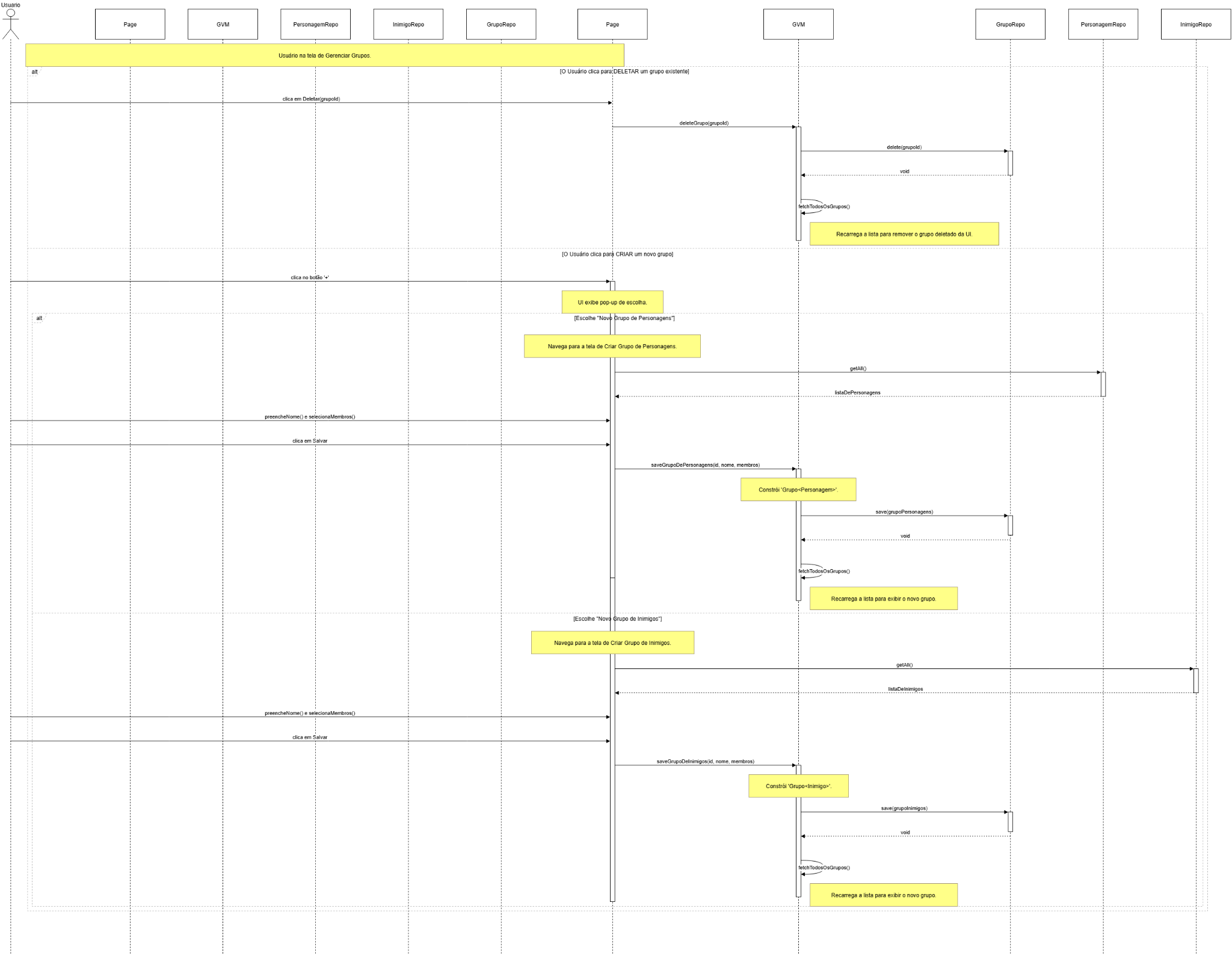
Criar ficha de inimigo



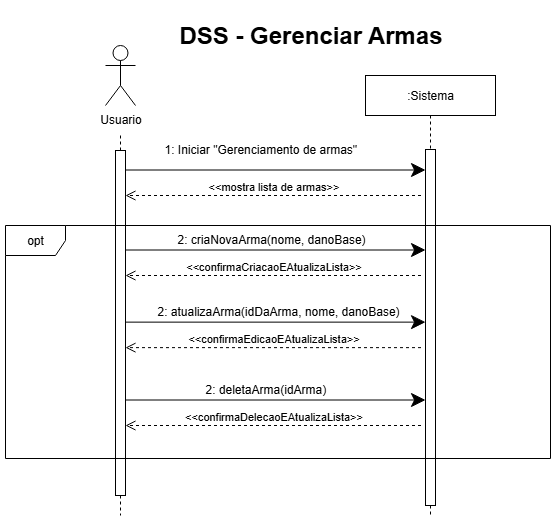
Configurar Batalha

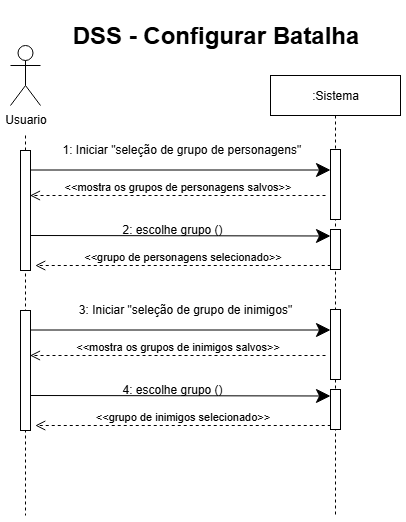


Gerenciar Grupo

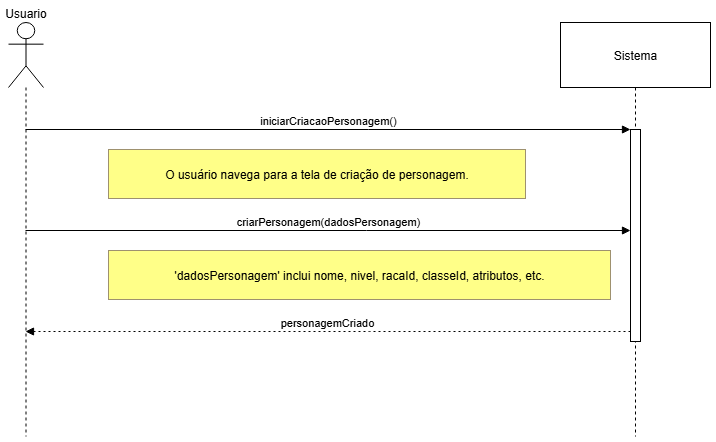


# 10 - DIAGRAMAS DE SEQUÊNCIA DE SISTEMA

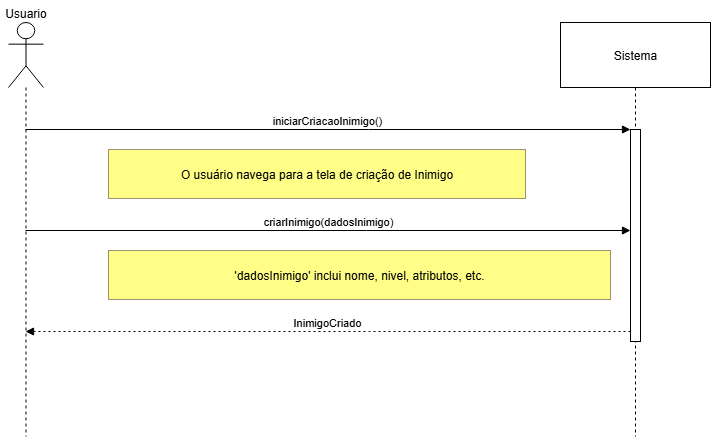




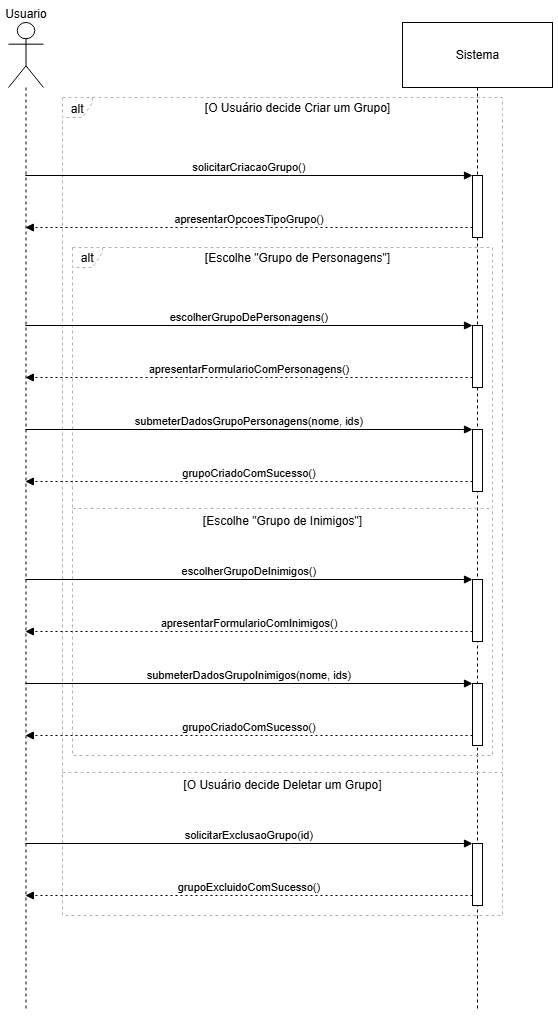
Criar Ficha Personagem



Criar Ficha Inimigo



Gerenciar Grupo



# 11 - MODELO DE CARACTERÍSTICAS

