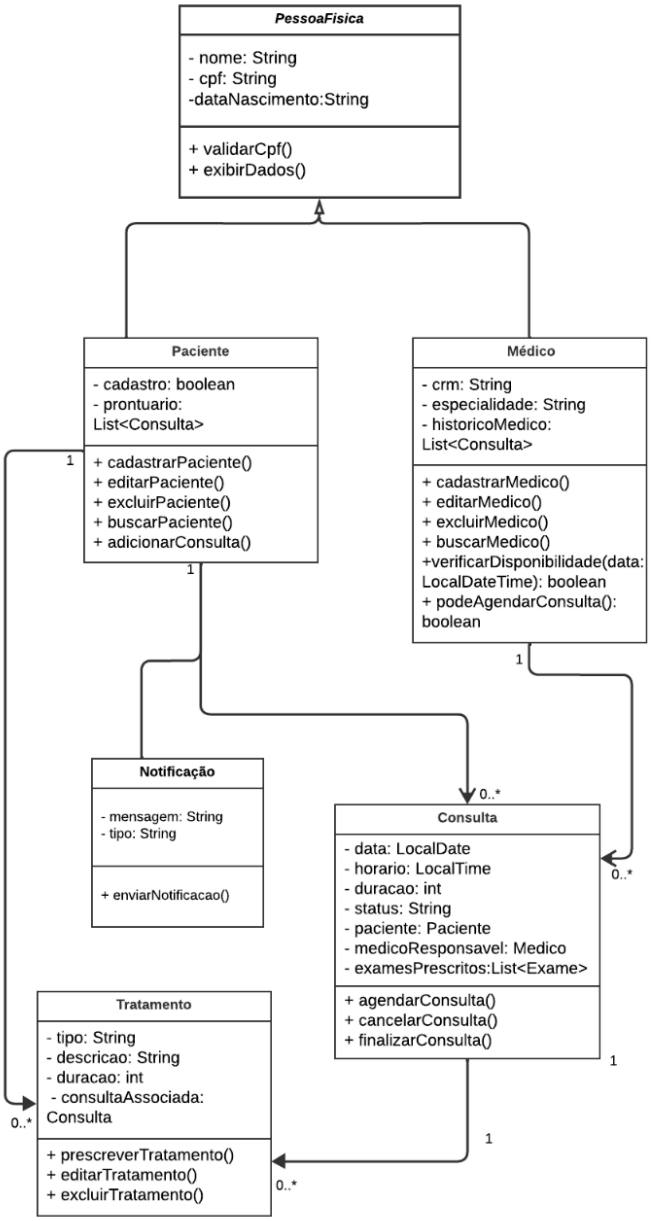
Maryane pereira da silva 221022687

Trabalho Prático



🔹 Associações entre as classes

As classes do sistema possuem **relacionamentos bem definidos** para garantir um fluxo consistente de informações.

1. **Paciente e Consulta**
   - Um paciente pode ter várias consultas em seu histórico.
   - Representado por um `List<Consulta>` na classe `Paciente`.
   - Associação **1:N (um para muitos)**.
2. **Médico e Consulta**
   - Um médico pode atender várias consultas.
   - Representado por um `List<Consulta>` na classe `Medico`.
   - Associação **1:N (um para muitos)**.
3. **Consulta e Exame/Medicamento**
   - Cada consulta pode ter vários exames e medicamentos prescritos.
   - Representado por `List<Exame>` e `List<Medicamento>` na classe `Consulta`.
   - Associação **1:N (um para muitos)**.
4. **Pagamento e Paciente**
   - Cada pagamento está associado a um paciente específico.
   - Representado por um atributo `Paciente` na classe `Pagamento`.
   - Associação **1:1 (um para um)**.

🔹 Herança

A herança foi utilizada para evitar **duplicação de código** e facilitar a manutenção.

1. **Pessoa (Superclasse)**
   - `Paciente` e `Medico` herdam de `Pessoa`.
   - Contém atributos comuns: `nome`, `cpf` e `dataNascimento`.
   - Método `toString()` sobrescrito para personalizar a exibição.
2. **Exame e Medicamento herdando de `Prescricao`**
   - Ambas as classes compartilham atributos como `consultaAssociada` e `dataValidade`.
   - Facilitando reuso e estrutura mais organizada.

🔹 Polimorfismo

🔸 Polimorfismo por Sobrescrita

O polimorfismo foi utilizado para **sobrescrever métodos em classes filhas**.

Exemplo:

O método `toString()` foi sobrescrito em várias classes (`Paciente`, `Medico`, `Consulta` etc.).
java

```
@Override
public String toString() {
    return "Paciente: " + nome + " | CPF: " + cpf;
}
```

-

🔸 Polimorfismo por Sobrecarga

Usamos sobrecarga para oferecer **diferentes formas de inicializar objetos**.

Exemplo:

Criando um pagamento com ou sem valor definido:
java
CopiarEditar

```
public Pagamento(Paciente paciente) {
    this.paciente = paciente;
    this.valor = 0.0;
    this.pago = false;
}

public Pagamento(Paciente paciente, double valor) {
    this.paciente = paciente;
    this.valor = valor;
    this.pago = false;
}
```

Justificativa para as Abordagens Customizadas

1. **Uso de Exceções Personalizadas**
   - Criamos exceções específicas (`HorarioIndisponivel`, `PagamentoPendenteException`, `EspecialidadeInvalida`) para tornar os erros **mais claros e tratáveis**.
   - Em vez de lançar `Exception` genérica, essas exceções melhoram a leitura e manutenção do código.
2. **Modularidade Separada em Pacotes**
   - `entidades` → contém as classes principais (`Paciente`, `Medico`, `Consulta`, etc.).
   - `servicos` → contém regras de negócio (`PagamentoService`, `AgendamentoService`).
   - `excecoes` → contém as exceções personalizadas.
   - Essa separação melhora a organização e **facilita futuras manutenções**.
3. **Uso de Collections para Gerenciar Listas**
   - `List<Consulta>` para histórico de consultas.
   - `Map<String, Pagamento>` para armazenar pagamentos de pacientes, garantindo busca rápida pelo CPF.
