



Nome: Letícia Justino Maciel.

Disciplina: Arquitetura de Sistemas.

Professor(a): Renato William.

1. Requisitos Funcionais (baseados no código)

RF01 – Cadastrar pacientes

O código possui a classe Paciente com todos os dados básicos.

RF02 – Gerenciar prontuário

A classe Prontuario permite registrar histórico, observações e evoluções.

RF03 – Agendar consultas

A classe Consulta e a classe Agenda permitem criar, listar e organizar horários.

RF04 – Registrar atendimentos

A consulta guarda observações do atendimento.

RF05 – Gerenciar dentistas e usuários

A classe Dentista e Usuario controlam os perfis que acessam o sistema.

RF06 – Registrar pagamentos

A classe Financeiro guarda valor, forma de pagamento e status.

RF07 – Gerar relatórios simples

A classe Financeiro possui método para calcular receitas do mês.

RF08 – Autenticação básica

A classe Usuario possui método de login simples.

2. Requisitos Não Funcionais (baseados no que o código oferece)

RNF01 – Segurança básica:

O código implementa login por usuário e senha (simples).

RNF02 – Organização e clareza:

As classes são separadas por arquivo .py, facilitando manutenção.

RNF03 – Baixa complexidade:

Design simples para rodar facilmente em qualquer máquina.

RNF04 – Arquitetura monolítica:

Todo o sistema roda dentro do mesmo projeto, sem divisão por serviços.

RNF05 – Disponibilidade:

Por ser simples e local, o sistema abre imediatamente no Python.

3. Módulos Principais (baseados nos arquivos criados)

paciente.py – Cadastro de pacientes
prontuario.py – Histórico clínico do paciente
dentista.py – Cadastro dos dentistas
usuario.py – Login e autenticação
consulta.py – Consultas individuais
agenda.py – Organização das consultas
financeiro.py – Controle financeiro
main.py – Menu principal que junta tudo

4. Usuários do Sistema (com base no código)

Dentista:

Pode acessar pacientes, prontuários e consultas.

Secretária:

Pode cadastrar pacientes e marcar consultas.

Administrador:

Possui login que acessa tudo, incluindo financeiro.
Eles são diferenciados pela classe Usuario, no atributo tipo.

5. Design e Arquitetura de Software (refletindo o código criado)

Modelo usado: Monolítico

Tudo roda dentro de um único projeto Python.

Organização em camadas dentro do projeto

1. Camada de Dados (entidades)

Classes: Paciente, Dentista, Consulta, Prontuario, Financeiro, Usuario
→ Elas armazenam informações e regras básicas.

2. Camada de Lógica

Classes como Agenda e Financeiro aplicam regras práticas: marcar consulta, calcular total recebido, registrar evolução.

3. Camada de Apresentação

`main.py` exibe o menu e chama os métodos.

Atributos e Métodos (baseados diretamente no código)

Paciente

Atributos: nome, telefone, cpf, nascimento, endereço
Métodos: atualizar_dados()

Prontuario

Atributos: observações, evoluções
Métodos: adicionar_evolucao()

Consulta

Atributos: paciente, dentista, data, hora
Métodos: adicionar_observacao()

Agenda

Atributos: lista_consultas
Métodos: marcar_consulta(), listar()

Dentista

Atributos: nome, especialidade
Métodos: nenhum complexo (somente dados)

Financeiro

Atributos: pagamentos
Métodos: registrar_pagamento(), total_mes()

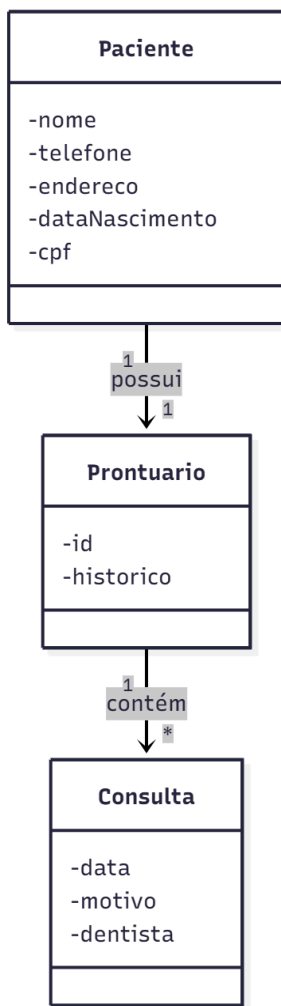
Usuario

Atributos: login, senha, tipo
Métodos: autenticar()

Conexões entre as classes

Paciente - possui um Prontuário
Consulta - liga Paciente + Dentista
Agenda - guarda várias Consultas
Financeiro - registra pagamentos das Consultas

DIAGRAMA UML:



PERGUNTAS:

Quais classes poderiam ser reutilizadas em outros sistemas?

Paciente - Pode ser usado em qualquer sistema que tenha cadastro de pessoas (ex.: clínica médica, academia, escola).

Dentista - Pode ser aproveitado em qualquer sistema que gerencie profissionais.

Consulta - Usável em sistemas de agendamento de qualquer tipo.

Financeiro - Pode ser usado em lojas, consultórios e sistemas que controlam pagamentos.

Agenda - Perfeita para qualquer aplicativo de marcação de horários.

Prontuário - Pode ser usado em outros sistemas de saúde.

Como esse design favorece a manutenção e expansão do software?

Cada classe tem uma responsabilidade, então alterações não bagunçam o resto do código.

Se quiser adicionar algo novo, como “cartão de vacinação” ou “estoque”, basta criar outra classe.

Os arquivos estão separados, então fica mais organizado e fácil de achar.

O `main.py` coordena tudo, então conseguimos testar as classes sem mexer no sistema inteiro.

O que mudaria se adotássemos uma arquitetura em camadas independentes (MVC ou microserviços)?

Se fosse **MVC**: As classes (`Paciente`, `Consulta`, `Prontuario...`) ficariam na parte Modelo. O `main.py` viraria `Controllers`, que chamam as funções. A parte visual (menus e prints) seria a `View`.

Isso deixaria o código mais dividido e organizado, mas exigiria criar mais arquivos e mais estrutura.

Se fossem **microserviços**: O sistema seria dividido em programas separados: um só para Pacientes, um só para Consultas, um só para Financeiro, etc.

Onde estariam as regras de negócio e a persistência?

Estão espalhadas dentro das classes e serviços. Por exemplo: Validação de dados - na classe `Paciente`.

Adicionar consulta - classe `Agenda`

Registrar pagamento - classe `Financeiro`

Adicionar atualização do paciente - classe `Prontuario`

Ou seja: as regras estão dentro das classes, onde elas fazem sentido.

Persistência (armazenamento)

No código simples que criamos, a persistência está sendo feita em listas dentro do próprio programa.

Ex.: lista de pacientes; lista de consultas; lista de prontuários.