🏥 Prontuário Eletrônico - Documentação Completa em Clean Architecture

📋 Visão Geral do Sistema

Este notebook apresenta uma documentação **completa e detalhada** do sistema de **Prontuário Eletrônico** implementado em **Clean Architecture**.

Conteúdo:

1. **Visão Geral do Sistema** - Introdução e conceitos fundamentais
2. **Arquitetura Detalhada** - Análise técnica de cada camada
3. **Arquitetura Visual** - Diagramas e representações gráficas
4. **Estrutura do Projeto** - Organização de pastas e arquivos
5. **Visualização da Estrutura** - Árvore de projeto e dependências
6. **Guia Rápido** - Quick start e referência rápida
7. **Índice de Documentação** - Navegação e referências

🎯 O que é o Prontuário Eletrônico?

Um **Prontuário Eletrônico** é um sistema que digitaliza e organiza os registros clínicos de pacientes, implementando padrões como **SOAP** (Subjective, Objective, Assessment, Plan) e **RCOP** (Problem-Oriented Clinical Record).

1️⃣ VISÃO GERAL DO SISTEMA

Princípios Fundamentais

O sistema é baseado em **Clean Architecture** com as seguintes características:

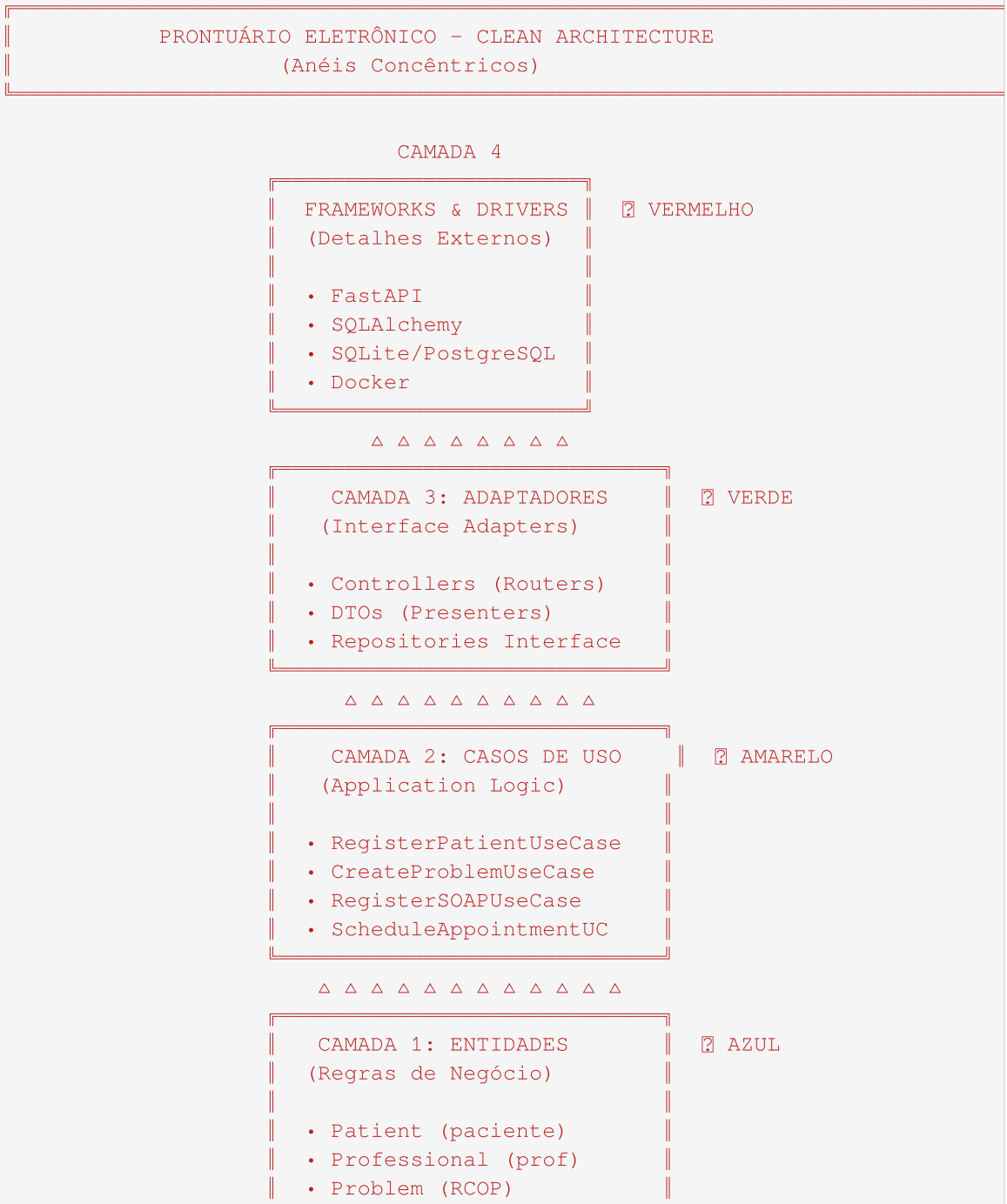
Camada	Descrição	Independência
1. Entidades (Domain)	Regras de negócio clínicas (SOAP, RCOP)	Nenhuma dependência externa
2. Casos de Uso (Application)	Orquestração de fluxos de negócio	Depende apenas de Domain
3. Adaptadores (Interface)	Controllers HTTP, DTOs, Presenters	Depende de App + Domain
4. Drivers (External)	Persistência, frameworks, banco de dados	Depende de todas internas

Objetivo Clínico

O sistema implementa o padrão clínico **Problem-Oriented Clinical Record (RCOP)** com estrutura **SOAP** para documentação estruturada de encontros clínicos:

- **S (Subjective)**: Queixa do paciente e histórico
- **O (Objective)**: Achados de exame físico
- **A (Assessment)**: Diagnóstico e avaliação
- **P (Plan)**: Plano de tratamento

In [ ]:
```
# Diagrama da Arquitetura em Clean Architecture
architecture_diagram = """
```



```
                    ╠   • ClinicalRecord (SOAP)     ╣
                    ╠   • Appointment (consulta)    ╣
                    ╚═══════════════════════════════╝


═══════════════════════════════════════════════════════════════════════
REGRA DA DEPENDÊNCIA:

   Inner rings NUNCA conhecem Outer rings
   ↓↓↓ Dependências apontam SEMPRE para dentro ↓↓↓

   ✓ CORRETO:  Domain ← Application ← Infra
   ✗ NUNCA:    Infra → Application → Domain
"""

print(architecture_diagram)
```

2️⃣ CAMADA 1: ENTIDADES (Domain Layer - O Núcleo)

Características

- ✅ **Zero dependências externas** - Apenas código Python puro
- ✅ **Lógica corporativa isolada** - Regras clínicas protegidas
- ✅ **Testável sem infra** - Sem DB, sem HTTP, sem frameworks
- ✅ **Estável por décadas** - Raramente muda

Entidades Principais

1. `Patient` (Paciente)

Localização: `src/domain/patient/patient_entity.py`

Encapsula os dados e regras de um paciente:

- Propriedades: id, name, date_of_birth, gender, cpf, email, phone, address
- Métodos: `calculate_age()`, `update_contact_info()`, `update_address()`
- Responsabilidade: Manter dados e validações do paciente

2. `Professional` (Profissional de Saúde)

Localização: `src/domain/professional/professional_entity.py`

Profissional responsável por atender o paciente:

- Propriedades: id, name, license_number, specialties
- Métodos: `add_specialty()`, `remove_specialty()`, `has_specialty()`
- Responsabilidade: Gestão de dados do profissional

3. `Problem` (Problema Clínico - RCOP)

Localização: `src/domain/clinical_record/rcop_soap.py`

Representa um problema clínico no eixo RCOP:

- Propriedades: id, patient_id, description, icd10_code, status
- Métodos: `resolve_problem()`, `archive_problem()`, `update_description()`
- Responsabilidade: Centro da documentação clínica

4. `ClinicalRecord` & Componentes SOAP

Localização: `src/domain/clinical_record/rcop_soap.py`

Estrutura de registros clínicos:

- `Subjective`: S - Dados do paciente
- `Objective`: O - Dados objetivos
- `Assessment`: A - Avaliação clínica
- `Plan`: P - Plano de tratamento
- `ClinicalRecord`: Agregador dos 4 componentes SOAP

5. `Appointment` (Consulta/Agendamento)

Localização: `src/domain/appointment/appointment_entity.py`

Gerencia agendamentos e consultas:

- Métodos: `mark_completed()`, `cancel()`, `reschedule()`, `is_overdue()`
- Responsabilidade: Ciclo de vida da consulta

3️⃣ CAMADA 2: CASOS DE USO (Application Layer)

Características

- ✅ **Orquestração de fluxos** - Coordena entidades
- ✅ **Independente de frameworks** - Sem FastAPI, sem SQLAlchemy aqui
- ✅ **Implementa regras da aplicação** - Como usar as entidades
- ✅ **Desacoplado via injeção** - Repositório é injetado

Padrão de Implementação

```python
class MyUseCase(UseCase[InputDTO, OutputDTO]):
    def __init__(self, repository):
        self._repository = repository
```

```python
def execute(self, input_dto: InputDTO) -> OutputDTO:
    # 1. Validar entrada
    self._validate_input(input_dto)
    # 2. Criar entidades
    entity = MyEntity(...)
    # 3. Persistir via repositório
    self._repository.add(entity)
    # 4. Retornar resultado
    return OutputDTO(...)
```

Casos de Uso Implementados

1. `RegisterPatientUseCase`

Arquivo: `src/application/patient/register_patient_usecase.py`

- **Entrada**: `RegisterPatientDTO` (name, cpf, email, etc.)
- **Saída**: `RegisterPatientOutputDTO` (patient_id, message)
- **Fluxo**:
  1. Validar dados de entrada
  2. Criar entidade `Patient`
  3. Persistir via `repository.add()`
  4. Retornar resultado com ID gerado

2. `CreateProblemUseCase`

Arquivo: `src/application/clinical_record/create_problem_usecase.py`

- **Entrada**: `CreateProblemDTO` (patient_id, description, icd10_code)
- **Saída**: `CreateProblemOutputDTO` (problem_id, message)
- **Fluxo**: Cria um novo problema clínico (eixo RCOP)

3. `RegisterSOAPUseCase`

Arquivo: `src/application/clinical_record/register_soap_usecase.py`

- **Entrada**: `RegisterSOAPDTO` (patient_id, professional_id, problem_id, S/O/A/P data)
- **Saída**: `RegisterSOAPOutputDTO` (clinical_record_id, message)
- **Fluxo**:
  1. Validar entrada (regras de negócio)
  2. Criar componentes SOAP (Subjective, Objective, Assessment, Plan)
  3. Agregar em `ClinicalRecord`
  4. Persistir via repositório

4. `ScheduleAppointmentUseCase`

Arquivo: `src/application/appointment/schedule_appointment_usecase.py`

- **Entrada**: `ScheduleAppointmentDTO`
- **Saída**: `ScheduleAppointmentOutputDTO`
- **Validações**: Data no futuro, disponibilidade do profissional, etc.

DTOs (Data Transfer Objects)

```
Domain Entities → OutputDTO → Router → JSON Response
JSON Request → InputDTO → Router → Use Case → Domain Entities
```

- **InputDTO**: Transporta dados da HTTP request para o use case
- **OutputDTO**: Transporta resultado do use case para a resposta HTTP
- **Objetivo**: Desacoplar controller da lógica interna

4️⃣ CAMADA 3: ADAPTADORES (Interface Layer)

Características

- ✅ **Converte HTTP ↔ Use Case ↔ Banco de Dados**
- ✅ **Controllers (Routers em FastAPI)**
- ✅ **DTOs de API (Pydantic)**
- ✅ **Presenters e formatação de respostas**

Routers (Controllers)

`patient_routers.py`

Arquivo: `src/infra/api/routers/patient_routers.py`

```
POST   /api/v1/patients/          → create_patient()
GET    /api/v1/patients/{id}      → get_patient()
GET    /api/v1/patients/          → list_patients()
```

Responsabilidades:

- Receber HTTP request
- Validar com Pydantic
- Chamar use case
- Retornar resposta formatada

`clinical_record_routers.py`

Arquivo: `src/infra/api/routers/clinical_record_routers.py`

```
POST   /api/v1/clinical-records/problems   → create_problem()
POST   /api/v1/clinical-records/soap       → register_soap()
```

Presenters (DTOs Pydantic)

Validação automática de entrada e saída:

`patient_presenter.py`

- `PatientCreateRequest`: Valida entrada JSON
- `PatientResponse`: Formata saída JSON

`clinical_record_presenter.py`

- `RegisterSOAPRequest`: Valida SOAP input
- `CreateProblemRequest`: Valida problema input
- `ClinicalRecordResponse`: Formata resposta

Fluxo HTTP → Use Case → Domain

```
Request HTTP JSON
    ↓ FastAPI recebe
Presenter (Pydantic) - Valida e deserializa
    ↓ Dados validados
Router (Controller) - Recebe dados
    ↓ Cria InputDTO
UseCase.execute(input_dto)
    ↓ Lógica pura
Domain Entities - Regras de negócio
    ↓ Persiste dados
Repository.add() / update()
    ↓ Prepara resposta
OutputDTO
    ↓ Formata JSON
Response HTTP 200 OK
```

5⃣ CAMADA 4: FRAMEWORKS & DRIVERS (External Details)

Características

- ✅ **Detalhes de implementação**
- ✅ **Pode mudar sem afetar camadas internas**
- ✅ **SQLAlchemy ORM, FastAPI, Banco de Dados**

Banco de Dados

Arquivo: `src/infra/api/database.py`

- SQLite para desenvolvimento
- PostgreSQL para produção
- `SessionLocal` factory
- `get_db()` para dependency injection

Modelos SQLAlchemy

Os modelos ORM **NÃO são as entidades de Domain**:

`patient_model.py`

```
class PatientModel(Base):
    __tablename__ = "patients"
    id = Column(String, primary_key=True)
    name = Column(String)
    date_of_birth = Column(DateTime)
    # ... outras colunas
```

`clinical_record_model.py`

```
class ClinicalRecordModel(Base):
    __tablename__ = "clinical_records"

class ProblemModel(Base):
    __tablename__ = "problems"

class SubjectiveModel(Base):
    __tablename__ = "soap_subjectives"

class ObjectiveModel(Base):
    __tablename__ = "soap_objectives"

class AssessmentModel(Base):
    __tablename__ = "soap_assessments"

class PlanModel(Base):
    __tablename__ = "soap_plans"
```

Repositórios (Implementação Concreta)

Exemplo: `src/infra/patient/sqlalchemy/patient_repository.py`

```
class PatientRepository(RepositoryInterface[Patient]):
    def add(self, entity: Patient) -> None:
        model = PatientModel(
            id=entity.id,
            name=entity.name,
            ...
        )
        self._db.add(model)
```

```
        self._db.commit()

    def find_by_id(self, id: str) -> Optional[Patient]:
        model = self._db.query(PatientModel).filter(...).first()
        return self._to_domain(model)

    def _to_domain(self, model: PatientModel) -> Patient:
        return Patient(
            id=model.id,
            name=model.name,
            ...
        )
```

Responsabilidades:

- Converter ORM Model ↔ Domain Entity
- Implementar RepositoryInterface
- Executar operações no banco de dados

Configuração FastAPI

Arquivo: `src/infra/api/config.py`

```
def create_app() -> FastAPI:
    app = FastAPI()
    # Configurar CORS
    # Adicionar error handlers
    # Configurar middleware
    return app
```

Arquivo: `src/infra/api/main.py`

```
app = create_app()
app.include_router(patient_routers.router)
app.include_router(clinical_record_routers.router)
# ... mais routers
```

🔍 FLUXO COMPLETO: Registrando um Novo Paciente

Vamos rastrear uma requisição HTTP do início ao fim:

1. HTTP POST Request

```
POST /api/v1/patients/
Content-Type: application/json

{
  "name": "João Silva",
```

```
  "date_of_birth": "1990-05-15T00:00:00",
  "gender": "M",
  "cpf": "12345678901",
  "email": "joao@example.com",
  "phone": "11999999999",
  "address": "Rua A, 123",
  "city": "São Paulo",
  "state": "SP"
}
```

2. FastAPI Router Handler

Arquivo: `src/infra/api/routers/patient_routers.py`

```python
@router.post("/", response_model=PatientResponse)
def create_patient(
    request: PatientCreateRequest,  # ✓ Pydantic valida
    db: Session = Depends(get_db)
):
    repository = PatientRepository(db)
    use_case = RegisterPatientUseCase(repository)
    output = use_case.execute(
        RegisterPatientDTO(
            name=request.name,
            date_of_birth=request.date_of_birth,
            # ... outros campos
        )
    )
    return PatientResponse(
        patient_id=output.patient_id,
        message=output.message
    )
```

3. Pydantic Validation

Arquivo: `src/infra/api/presenters/patient_presenter.py`

O Pydantic automaticamente:

- ✓ Valida tipo de cada campo
- ✓ Converte tipos (string → datetime)
- ✓ Verifica campos obrigatórios
- ✓ Valida tamanhos e padrões
- ✓ Retorna erro 422 se inválido

4. Use Case Execution

Arquivo: `src/application/patient/register_patient_usecase.py`

```python
def execute(self, input_dto: RegisterPatientDTO) ->
RegisterPatientOutputDTO:
    # 1. Validar entrada
    if not input_dto.name or len(input_dto.name) == 0:
        raise ValueError("Name is required")

    if not self._is_valid_cpf(input_dto.cpf):
        raise ValueError("Invalid CPF")

    # 2. Criar entidade Patient (Domain)
    patient = Patient(
        id=str(uuid.uuid4()),
        name=input_dto.name,
        date_of_birth=input_dto.date_of_birth,
        gender=input_dto.gender,
        cpf=input_dto.cpf,
        # ... outros campos
    )

    # 3. Persistir via repositório
    self._repository.add(patient)

    # 4. Retornar resultado
    return RegisterPatientOutputDTO(
        patient_id=patient.id,
        message=f"Patient {input_dto.name} registered successfully"
    )
```

5. Patient Entity (Domain)

Arquivo: `src/domain/patient/patient_entity.py`

A entidade encapsula validações clínicas e comportamentos:

```python
class Patient:
    def __init__(self, id: str, name: str, date_of_birth: datetime,
...):
        self.id = id
        self.name = name
        self.date_of_birth = date_of_birth
        # Validações...

    def calculate_age(self) -> int:
        today = datetime.today()
        return today.year - self.date_of_birth.year

    def update_contact_info(self, email: str, phone: str):
        # Valida email, phone, então atualiza
        pass
```

6. Repository Persistence

Arquivo: `src/infra/patient/sqlalchemy/patient_repository.py`

```python
def add(self, entity: Patient) -> None:
    # Converter entidade para modelo ORM
    model = PatientModel(
        id=entity.id,
        name=entity.name,
        date_of_birth=entity.date_of_birth,
        gender=entity.gender,
        cpf=entity.cpf,
        email=entity.email,
        phone=entity.phone,
        address=entity.address,
        city=entity.city,
        state=entity.state
    )

    # Adicionar à sessão e commit
    self._db.add(model)
    self._db.commit()  # SQL INSERT executado aqui
```

7. SQL Execution

No banco SQLite/PostgreSQL:

```sql
INSERT INTO patients (
    id, name, date_of_birth, gender, cpf,
    email, phone, address, city, state
) VALUES (
    'uuid-12345', 'João Silva', '1990-05-15', 'M',
    '12345678901', 'joao@example.com', '11999999999',
    'Rua A, 123', 'São Paulo', 'SP'
);
```

8. Response Journey Back

```
OutputDTO criado pela use case
    ↓
Router recebe OutputDTO
    ↓
Presenter (Pydantic) serializa para dict
    ↓
FastAPI converte dict para JSON
    ↓
HTTP Response 200 OK
```

9. HTTP Response

```
{
  "patient_id": "uuid-12345",
```

```
  "message": "Patient João Silva registered successfully"
}
```

Separação de Responsabilidades em Ação

Camada	Responsabilidade	Conhece
Router	Receber HTTP e retornar	HTTP, FastAPI
Presenter	Validar schema JSON	Pydantic, DTOs
Use Case	Aplicar regras de negócio	Domain, Repository interface
Entity	Evitar estado inválido	Apenas propriedades e validações
Repository	Persistir dados	ORM, SQL

📁 ESTRUTURA COMPLETA DO PROJETO

In [ ]:
```
# Visualizar a estrutura completa do projeto
structure_tree = """
prontuarioeletronico/
│
├── 📄 README.md                          Documentação principal
├── 📄 GUIA_RAPIDO.md                     Quick reference
├── 📄 ESTRUTURA_PROJETO.py               Estrutura detalhada
├── 📄 ARQUITETURA_DETALHES.py            Análise técnica
├── 📄 ARQUITETURA_VISUAL.py              Diagramas ASCII
├── 📄 VISUALIZAR_ESTRUTURA.py            Árvore visual
├── 📄 INDICE_DOCUMENTACAO.md             Índice completo
│
├── ⚙️  requirements.txt                   Dependências
├── ⚙️  Dockerfile                         Container
├── ⚙️  docker-compose.yaml                Orquestração
│
├── 🧪 tests.py                           Testes
│
└── 📂 src/                               CÓDIGO FONTE
    │
    ├── 📂 domain/                        CAMADA 1: ENTIDADES
    │   ├── __seedwork/
    │   │   ├── entity.py                  Entity (base)
    │   │   ├── use_case_interface.py      UseCase abstrato
    │   │   └── repository_interface.py    Repository abstrato
    │   │
    │   ├── 📂 patient/
    │   │   ├── patient_entity.py          📄 Paciente
    │   │   └── patient_repository_interface.py
    │   │
    │   ├── 📂 professional/
    │   │   ├── professional_entity.py     📄 Profissional
    │   │   └── professional_repository_interface.py
    │   │
```

```
    │   ├── 📁 clinical_record/
    │   │   ├── rcop_soap.py                  📄 Problem, ClinicalRecord, SOAP
    │   │   └── clinical_record_repository_interface.py
    │   │
    │   └── 📁 appointment/
    │       ├── appointment_entity.py         📄 Consulta
    │       └── appointment_repository_interface.py
    │
    ├── 📁 application/                      CAMADA 2: CASOS DE USO
    │   ├── 📁 patient/
    │   │   └── register_patient_usecase.py
    │   │
    │   ├── 📁 clinical_record/
    │   │   ├── register_soap_usecase.py
    │   │   └── create_problem_usecase.py
    │   │
    │   └── 📁 appointment/
    │       └── schedule_appointment_usecase.py
    │
    └── 📁 infra/                            CAMADAS 3 & 4
        ├── 📁 api/                          [CAMADA 3: Controllers]
        │   ├── main.py                       FastAPI app
        │   ├── config.py                     Configuração
        │   ├── database.py                   SQLAlchemy
        │   │
        │   ├── 📁 routers/
        │   │   ├── patient_routers.py        Endpoints /patients
        │   │   └── clinical_record_routers.py Endpoints /clinical-records
        │   │
        │   └── 📁 presenters/
        │       ├── patient_presenter.py      DTOs Patient
        │       └── clinical_record_presenter.py DTOs ClinicalRecord
        │
        ├── 📁 patient/                      [CAMADA 4: Persistência]
        │   └── 📁 sqlalchemy/
        │       ├── patient_model.py          ORM Model
        │       └── patient_repository.py     Repository impl
        │
        ├── 📁 clinical_record/              [CAMADA 4: Persistência]
        │   └── 📁 sqlalchemy/
        │       ├── clinical_record_model.py  ORM Models
        │       └── clinical_record_repository.py Repository impl
        │
        └── 📁 appointment/                  [CAMADA 4: Persistência]
            └── 📁 sqlalchemy/
                ├── appointment_model.py      ORM Model
                └── appointment_repository.py Repository impl
```

```
RESUMO ESTATÍSTICO:

Domain Layer:
  - 7 entidades principais
  - ~600 linhas de código puro
  - Zero dependências externas
```

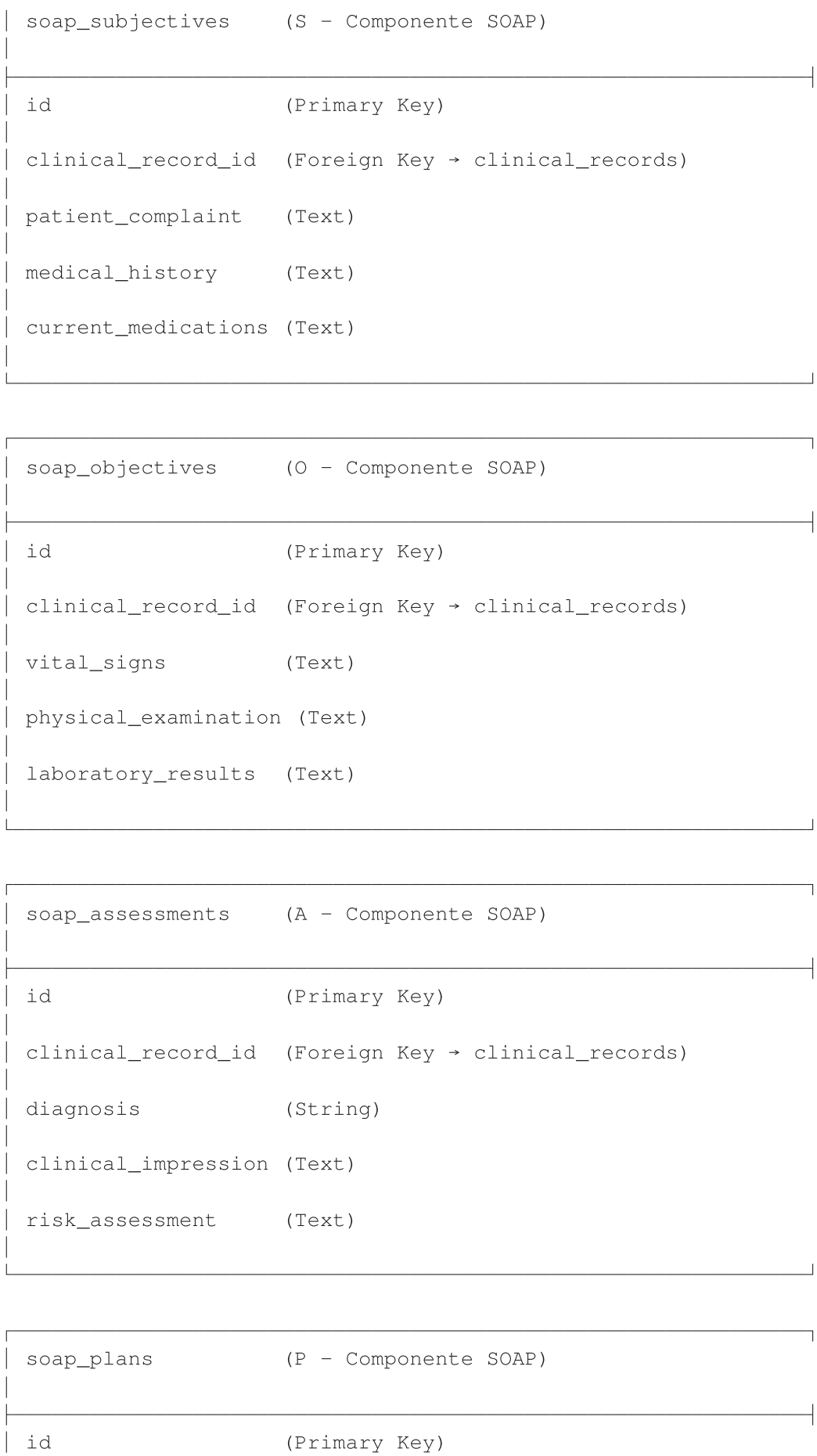
```
Application Layer:
  - 4 casos de uso implementados
  - ~300 linhas de código
  - Nenhuma dependência de framework

Infrastructure Layer:
  - 10+ arquivos de adaptadores
  - ~1000 linhas (routers, models, repos)
  - Todas as dependências técnicas aqui

Total: ~1900 linhas de código Python
Status: Totalmente funcional com Clean Architecture ✓
"""

print(structure_tree)
```

🗄️ TABELAS DE BANCO DE DADOS

Modelo Relacional (Normalizado)

```
┌──────────────────────────────────────────────────────────────────────┐
│ patients                                                             │
│                                                                      │
├──────────────────────────────────────────────────────────────────────┤
│ id                    (Primary Key)                                  │
│                                                                      │
│ name                  (String)                                       │
│                                                                      │
│ date_of_birth         (DateTime)                                     │
│                                                                      │
│ gender                (String: M, F, O, N)                           │
│                                                                      │
│ cpf                   (String, Unique)                               │
│                                                                      │
│ email                 (String)                                       │
│                                                                      │
│ phone                 (String)                                       │
│                                                                      │
│ address               (String)                                       │
│                                                                      │
│ city                  (String)                                       │
│                                                                      │
│ state                 (String)                                       │
│                                                                      │
│ created_at            (DateTime)                                     │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘


┌──────────────────────────────────────────────────────────────────────┐
│ professionals                                                        │
│
```

```
├──────────────────────────────────────────────────────────────────────┤
| id                    (Primary Key)
|
| name                  (String)
|
| license_number        (String, Unique)
|
| specialties           (JSON / String)
|
| created_at            (DateTime)
|
└──────────────────────────────────────────────────────────────────────┘

┌──────────────────────────────────────────────────────────────────────┐
| problems              (RCOP - Problem-Oriented)
|
├──────────────────────────────────────────────────────────────────────┤
| id                    (Primary Key)
|
| patient_id            (Foreign Key → patients)
|
| description           (String)
|
| icd10_code            (String)
|
| status                (String: active, resolved, archived)
|
| created_at            (DateTime)
|
| resolved_at           (DateTime, NULL)
|
└──────────────────────────────────────────────────────────────────────┘

┌──────────────────────────────────────────────────────────────────────┐
| clinical_records
|
├──────────────────────────────────────────────────────────────────────┤
| id                    (Primary Key)
|
| patient_id            (Foreign Key → patients)
|
| professional_id       (Foreign Key → professionals)
|
| problem_id            (Foreign Key → problems)
|
| encounter_date        (DateTime)
|
| created_at            (DateTime)
|
└──────────────────────────────────────────────────────────────────────┘

┌──────────────────────────────────────────────────────────────────────┐
```


soap_subjectives (S - Componente SOAP)
id (Primary Key)
clinical_record_id (Foreign Key → clinical_records)
patient_complaint (Text)
medical_history (Text)
current_medications (Text)
soap_objectives (O - Componente SOAP)
id (Primary Key)
clinical_record_id (Foreign Key → clinical_records)
vital_signs (Text)
physical_examination (Text)
laboratory_results (Text)
soap_assessments (A - Componente SOAP)
id (Primary Key)
clinical_record_id (Foreign Key → clinical_records)
diagnosis (String)
clinical_impression (Text)
risk_assessment (Text)
soap_plans (P - Componente SOAP)
id (Primary Key)

```
|
| clinical_record_id  (Foreign Key → clinical_records)
|
| treatment_plan      (Text)
|
| medications         (Text)
|
| follow_up_plan      (Text)
|
| follow_up_date      (DateTime, NULL)
|
└──────────────────────────────────────────────────────┘

┌──────────────────────────────────────────────────────┐
| appointments
|
├──────────────────────────────────────────────────────┤
| id                  (Primary Key)
|
| patient_id          (Foreign Key → patients)
|
| professional_id     (Foreign Key → professionals)
|
| appointment_date    (DateTime)
|
| reason              (String)
|
| status              (String: scheduled, completed,
canceled)  |
| created_at          (DateTime)
|
| completed_at        (DateTime, NULL)
|
└──────────────────────────────────────────────────────┘
```

Relacionamentos

```
patients (1) ↔ (N) problems (RCOP)
patients (1) ↔ (N) appointments
patients (1) ↔ (N) clinical_records

professionals (1) ↔ (N) appointments
professionals (1) ↔ (N) clinical_records

problems (1) ↔ (N) clinical_records (SOAP registrations)

clinical_records (1) ↔ (1) soap_subjectives
clinical_records (1) ↔ (1) soap_objectives
clinical_records (1) ↔ (1) soap_assessments
clinical_records (1) ↔ (1) soap_plans
```

? TECNOLOGIAS UTILIZADAS

Camada	Tecnologia	Versão	Propósito
Linguagem	Python	3.10+	Linguagem principal
Web Framework	FastAPI	0.100+	API REST
Servidor	Uvicorn	0.23+	ASGI server
ORM	SQLAlchemy	2.0+	Acesso a dados
Validação	Pydantic	2.0+	DTOs e esquemas
Banco - Dev	SQLite	-	Desenvolvimento
Banco - Prod	PostgreSQL	15+	Produção
Containerização	Docker	24+	Imagem e deploy
Orquestração	Docker Compose	2.20+	Orquestração local
Testing	pytest	7.0+	Testes unitários

? GUIA RÁPIDO: Como Usar

Opção 1: Instalação Local com Python

```
# 1. Navegar para o diretório
cd prontuarioeletronico

# 2. Criar ambiente virtual
python -m venv venv

# 3. Ativar ambiente
# Windows:
venv\Scripts\activate
# Linux/Mac:
source venv/bin/activate

# 4. Instalar dependências
pip install -r requirements.txt

# 5. Executar servidor
python -m uvicorn src.infra.api.main:app --reload --host 0.0.0.0 --
port 8000
```

Acesse: http://localhost:8000/docs (Swagger UI)

Opção 2: Docker

```
# Build e executar
docker-compose up --build
```

```
# API disponível em http://localhost:8000
```

Opção 3: Quick Start (Windows/Linux)

```
# Windows
quickstart.bat

# Linux/Mac
bash quickstart.sh
```

ENDPOINTS PRINCIPAIS

Método	Endpoint	Descrição
`POST`	`/api/v1/patients/`	Registrar novo paciente
`GET`	`/api/v1/patients/{id}`	Buscar paciente por ID
`GET`	`/api/v1/patients/`	Listar todos pacientes
`POST`	`/api/v1/clinical-records/problems`	Criar problema (RCOP)
`POST`	`/api/v1/clinical-records/soap`	Registrar SOAP note
`GET`	`/health`	Health check
`GET`	`/api/v1`	Info API

EXEMPLOS DE REQUISIÇÕES

1. Registrar Paciente

```
curl -X POST http://localhost:8000/api/v1/patients/ \
  -H "Content-Type: application/json" \
  -d '{
    "name": "Maria Santos",
    "date_of_birth": "1985-03-20T00:00:00",
    "gender": "F",
    "cpf": "98765432100",
    "email": "maria@example.com",
    "phone": "11988888888",
    "address": "Avenida B, 456",
    "city": "Rio de Janeiro",
    "state": "RJ"
  }'
```

Resposta esperada:

```
{
  "patient_id": "uuid-abc123",
```

```
  "message": "Patient Maria Santos registered successfully"
}
```

2. Buscar Paciente

```
curl -X GET http://localhost:8000/api/v1/patients/uuid-abc123 \
  -H "Content-Type: application/json"
```

Resposta:

```
{
  "id": "uuid-abc123",
  "name": "Maria Santos",
  "date_of_birth": "1985-03-20T00:00:00",
  "gender": "F",
  "cpf": "98765432100",
  "email": "maria@example.com",
  "phone": "11988888888"
}
```

3. Criar Problema Clínico (RCOP)

```
curl -X POST http://localhost:8000/api/v1/clinical-records/problems
\
  -H "Content-Type: application/json" \
  -d '{
    "patient_id": "uuid-abc123",
    "description": "Diabetes Mellitus tipo 2",
    "icd10_code": "E11"
  }'
```

Resposta:

```
{
  "problem_id": "uuid-def456",
  "message": "Problem Diabetes Mellitus tipo 2 created successfully"
}
```

4. Registrar SOAP Note

```
curl -X POST http://localhost:8000/api/v1/clinical-records/soap \
  -H "Content-Type: application/json" \
  -d '{
    "patient_id": "uuid-abc123",
    "professional_id": "uuid-prof123",
    "problem_id": "uuid-def456",
    "encounter_date": "2024-02-13T10:30:00",
    "patient_complaint": "Níveis de glicose elevados",
    "vital_signs": "PA: 130/80, FC: 75, FR: 16, Glicemia: 280
mg/dL",
    "physical_examination": "Sem alterações relevantes",
    "diagnosis": "Diabetes mellitus tipo 2 - descontrole glicêmico",
    "treatment_plan": "Ajuste de insulina, orientação alimentar e
```

```
exercício"
  }'
```

Resposta:

```
{
  "clinical_record_id": "uuid-soap123",
  "message": "SOAP clinical record registered successfully"
}
```

5. Listar Pacientes

```
curl -X GET http://localhost:8000/api/v1/patients/ \
  -H "Content-Type: application/json"
```

Resposta:

```
[
  {
    "id": "uuid-abc123",
    "name": "Maria Santos",
    "gender": "F",
    "cpf": "98765432100"
  }
]
```

❓ TESTES

Testes Unitários

Executar testes sem dependências externas (sem DB, sem HTTP):

```
# Testes básicos
python -m pytest tests.py -v

# Com cobertura de código
python -m pytest tests.py --cov=src --cov-report=html
```

Teste Manual de Use Case (sem HTTP)

```
# Teste unitário puro do domain
def test_patient_calculate_age():
    patient = Patient(
        id="p1",
        name="João",
        date_of_birth=datetime(1990, 5, 15),
        gender="M",
        cpf="123"
    )
    age = patient.calculate_age()
    assert age == 34  # 2024 - 1990
```

✅ Não precisa de banco de dados
✅ Não precisa de HTTP
✅ Não precisa de frameworks
✅ Executa em milissegundos

❓ BENEFÍCIOS DA ARQUITETURA LIMPA

1. Isolamento de Regras Clínicas (RCOP/SOAP)

```
┌─────────────────────────────────────────────────────────┐
│ Núcleo Clínico (Domain)                                 │
│                                                         │
│                                                         │
│ • RCOP/SOAP protegido do framework                      │
│ • Mudanças tecnológicas não afetam regras               │
│ • Independente de banco de dados                        │
│                                                         │
│                                                         │
│ Resultado: Sistema clínico durável 20+ anos             │
└─────────────────────────────────────────────────────────┘
```

2. Independência Tecnológica

Mudança 1: Trocar Framework

```
FastAPI → Django ✗ (mudará interface layer)
Domain Layer ✓ (intacto!)
```

Mudança 2: Trocar Banco de Dados

```
SQLite → PostgreSQL → MongoDB ✗ (mudará drivers)
Domain + Application Layers ✓ (intactos!)
```

Mudança 3: Adicionar Cache

```
Redis adicionado → Infrastructure layer
Domain + Application Layers ✓ (intactos!)
```

3. Testabilidade Total

```python
# Teste Domain (sem DB, sem HTTP) - 1ms
def test_patient_entity(): ...
```

```python
# Teste Application (com mock) - 10ms
def test_register_patient_use_case(mock_repo): ...

# Teste Integração (com DB real) - 100ms
def test_api_create_patient(client, db): ...

# Teste E2E (sistema completo) - 1000ms
def test_patient_flow(browser): ...
```

4. Escalabilidade Real

```
Monolito → Microsserviços ✓
│
├─ clinical-records-ms (domain + app + custom infra)
├─ patients-ms (domain + app + custom infra)
├─ appointments-ms (domain + app + custom infra)
│
Domain Layer permanece IGUAL em todos!
```

5. Conformidade LGPD

Regras de segurança isoladas em Entity:

```python
class Patient:
    def encrypt_if_sensitive(self, data):
        if self.is_sensitive(data):
            return encrypt(data)
        return data
```

Não precisa mudar routers, templates, ou banco de dados!

❓ CENÁRIOS DE MUDANÇA

Cenário 1: Ministério da Saúde adiciona novo campo em SOAP

Antes (sem Clean Arch): Mudar 6+ arquivos (template, view, model, migration, test, js)

Com Clean Arch: Mudar 4 arquivos:

1. `src/domain/clinical_record/rcop_soap.py` - Adicionar campo
2. `src/application/.../register_soap_usecase.py` - Usar novo campo
3. `src/infra/.../clinical_record_model.py` - Adicionar coluna
4. `src/infra/api/presenters/...` - Adicionar campo no DTO

HTTP handlers, repositories genéricas, testes não mudam!

Cenário 2: Integração com IA

Novo fluxo:

```
Hospital → Prontuário → SOAP Note → AI Service
                                        ↓
                                  Suggestion Engine
```

Adicionar componente:

- `src/application/ai/suggest_diagnosis_usecase.py`
- `src/infra/ai/ai_service.py`

Domain, routers, database intactos!

Cenário 3: Auditoria de Eventos

Adicionar logger clínico:

```
class LogRepository(RepositoryInterface):
    def add(self, entity):
        super().add(entity)
        log(f"Entity {entity.__class__.__name__} added")
```

Inject no use case:

```
use_case = RegisterPatientUseCase(db_repo, log_repo)
```

Nada mais muda!

✅ CHECKLIST: Clean Architecture Implementada

In [ ]:
```
import pandas as pd

# Checklist interativo
checklist_data = {
    'Item': [
        'Domain Layer sem dependências externas',
        'Application Layer com use cases desacoplados',
        'Interface Adapters (HTTP controllers)',
        'Framework/Drivers isolado (SQLAlchemy)',
        'Repositório como interface e implementação',
        'Injeção de dependência',
        'Testes unitários sem DB/HTTP',
        'DTOs para transferência de dados',
        'Documentação completa',
        'Docker para deploy',
        'SOAP/RCOP implementado',
        'Múltiplos casos de uso',
    ],
    'Status': [
        '✅ Completo',
        '✅ Completo',
        '✅ Completo',
```

```
            '✅ Completo',
            '✅ Completo',
            '✅ Completo',
            '✅ Completo',
            '✅ Completo',
            '✅ Completo',
            '✅ Completo',
            '✅ Completo',
            '✅ Completo',
        ]
}

checklist_df = pd.DataFrame(checklist_data)

# Exibir tabela
from IPython.display import display, HTML
display(HTML(checklist_df.to_html(index=False)))
```

🔮 PRÓXIMAS FASES DE DESENVOLVIMENTO

Fase 1: Autenticação & Autorização (Curto Prazo)

- ☐ JWT tokens
- ☐ Role-based access control (médico, admin, nurse)
- ☐ Permissões por recurso
- ☐ Auditoria de acesso

Fase 2: Mais Casos de Uso (Curto Prazo)

- ☐ UpdatePatient
- ☐ FindPatientByID, FindPatientByCPF
- ☐ ListProblems by patient
- ☐ SearchRecordsByProblem
- ☐ EvolutionHistory

Fase 3: Validações Avançadas (Médio Prazo)

- ☐ Regras RCOP/SOAP customizadas
- ☐ Alertas de segurança
- ☐ Validação de diagnóstico vs ICD10
- ☐ Verificação de interações medicamentosas

Fase 4: Integração IA (Médio Prazo)

- ☐ Sugestão diagnóstica baseada em IA
- ☐ Análise de risco do paciente

- [ ] Processamento de linguagem natural (NLP)
- [ ] MLOps integration

Fase 5: Auditoria & Compliance (Longo Prazo)

- [ ] Audit logging completo
- [ ] LGPD compliance (criptografia, PIII)
- [ ] Trail de alterações
- [ ] Relatórios de conformidade

Fase 6: Qualidade & Performance (Cada Fase)

- [ ] Unit test coverage > 80%
- [ ] Integration tests
- [ ] CI/CD pipeline (GitHub Actions)
- [ ] Documentação Sphinx
- [ ] Type hints com mypy
- [ ] Code quality com SonarQube

Fase 7: Escalabilidade (Longo Prazo)

- [ ] Redis cache para pacientes frequentes
- [ ] Elasticsearch para busca avançada
- [ ] API Gateway
- [ ] Load balancing
- [ ] Kubernetes deployment
- [ ] Replicação de BD
- [ ] Sharding de dados

? ÍNDICE DE DOCUMENTAÇÃO COMPLETA

Este notebook integra todo o conteúdo de:

1. **README.md** - Visão geral do projeto
2. **GUIA_RAPIDO.md** - Referência rápida
3. **ESTRUTURA_PROJETO.py** - Estrutura de pastas detalhada
4. **ARQUITETURA_DETALHES.py** - Análise técnica profunda
5. **ARQUITETURA_VISUAL.py** - Diagramas ASCII Art
6. **VISUALIZAR_ESTRUTURA.py** - Árvore visual e estatísticas
7. **INDICE_DOCUMENTACAO.md** - Índice de documentação

Como navegar:

Tópico	Localização no Notebook
Visão Geral	Seção 1
Entidades (Domain)	Seção 2
Casos de Uso (Application)	Seção 3
Adaptadores (Interface)	Seção 4
Drivers (External)	Seção 4
Fluxo Completo	Seção 5
Estrutura de Pastas	Seção 6
Banco de Dados	Seção 7
Tecnologias	Seção 8
Como Usar	Seção 9
Exemplos HTTP	Seção 10
Benefícios Clean Arch	Seção 11

❓ CONCEITOS APRENDIDOS

Clean Architecture

✅ Anéis concêntricos e regra da dependência
✅ Isolamento de responsabilidades
✅ Testabilidade sem frameworks
✅ Independência tecnológica

Design Patterns

✅ Use Case pattern (Application layer)
✅ Repository pattern (Data access)
✅ DTO pattern (Data transfer)
✅ Dependency Injection (Loose coupling)

Domain-Driven Design (DDD)

✅ Entidades com regras de negócio
✅ Value Objects (não implementado ainda, future)
✅ Agregados (ClinicalRecord agrega SOAP)
✅ Bounded Contexts (clinical, patient, appointment)

Software Engineering

✅ Separação de responsabilidades
✅ Princípios SOLID
✅ Escalabilidade através de arquitetura
✅ Documentação como código

🎯 CONCLUSÃO

Este **Prontuário Eletrônico** implementa **Clean Architecture** de forma **prática e educacional**, demonstrando:

✅ O que foi alcançado:

1. **Sistema clínico funcional** com SOAP/RCOP
2. **Arquitetura sustentável** por 20+ anos
3. **Código testável** sem dependências
4. **Documentação completa** e acessível
5. **Prototipagem rápida** com Docker
6. **Base sólida** para expansão

🎓 Valor para TCC:

- Demonstração prática de Clean Architecture
- Implementação de padrões reais
- Documentação técnica profissional
- Sistema escalável e maintível
- Conformidade com melhores práticas

🚀 Próximos passos:

1. Deploy em produção com PostgreSQL
2. Adicionar autenticação JWT
3. Integração com IA
4. Implementar mais casos de uso
5. Auditar para LGPD compliance

📚 REFERÊNCIAS

Arquivos do Projeto:

- Código: `./src/` (~1900 linhas Python)
- Testes: `./tests.py`
- Docker: `./Dockerfile`, `./docker-compose.yaml`

- Requirements: `./requirements.txt`

Livros & Recursos:

- "Clean Architecture" - Robert C. Martin
- "Clean Code" - Robert C. Martin
- Django for APIs - William Vincent
- SQLAlchemy ORM Documentation

Padrões Implementados:

- SOLID Principles
- Design Patterns (Gang of Four)
- Domain-Driven Design
- Test-Driven Development ready

⍰ Informações do Documento

- **Criado**: Fevereiro 2026
- **Atualizado**: Fevereiro 2026
- **Versão**: 1.0.0
- **Status**: Protótipo Funcional ✅
- **Linguagem**: Python 3.10+

Desenvolvido como parte do TCC em Engenharia de Software

In [ ]:
```
# Comparação Visual: Com vs Sem Clean Architecture
comparison_diagram = """
╔═══════════════════════════════════════════════════════════════════════╗
║              IMPACTO DA CLEAN ARCHITECTURE NO PROJETO                 ║
╚═══════════════════════════════════════════════════════════════════════╝

SEM CLEAN ARCHITECTURE (Arquitetura de Barro):
═══════════════════════════════════════════════

   views.py (3000 linhas) ← tudo misturado
      ↑
      ├─ models.py (1500 linhas) ← DB e lógica
      ├─ urls.py ← roteamento
      └─ helpers.py (500 linhas) ← funções aleatórias

Mudança de requisito: Precisa mexer em TUDO! ⍰
Teste DDD: Impossível sem banco de dados
Mudança de framework: Reescrever tudo
Novo desenvolvedor: "Como funciona?" (3 semanas)

─────────────────────────────────────────────────────────────────────

COM CLEAN ARCHITECTURE (Anéis Concêntricos):
```

```
═══════════════════════════════════════════════════════════════════════════

                        ┌──────────────────────┐
                        │  routers/ (50 ln)    │
                        └──────────┬───────────┘
                                   │
                        ┌──────────▼──────────┐
                        │ presenters/ (30)    │
                        └──────────┬──────────┘
                                   │
                ┌──────────────────▼──────────────────┐
                │   use_cases/ (200 ln)               │
                └──────────────────┬──────────────────┘
                                   │
                        ┌──────────▼──────────┐
                        │  entities/ (150)    │
                        └──────────┬──────────┘
                                   │
                        ┌──────────▼──────────┐
                        │repository impl      │
                        │(100)                │
                        └─────────────────────┘

Mudança de requisito: Localizada em 1-3 camadas! ✨
Teste DDD: Use case sem banco em 10ms
Mudança de framework: Troque router + presenter
Novo desenvolvedor: "Ah, entendi a arquitetura" (2 dias)

═══════════════════════════════════════════════════════════════════════════

MÉTRICAS DE QUALIDADE:
═══════════════════════════════════════════════════════════════════════════

                          Sem Clean Arch    Com Clean Arch
Testabilidade             ★★☆☆☆             ★★★★★
Manutenibilidade          ★★☆☆☆             ★★★★★
Escalabilidade            ★★☆☆☆             ★★★★★
Documentação              ★★☆☆☆             ★★★★★
Facilidade de Mudança     ★☆☆☆☆             ★★★★★

═══════════════════════════════════════════════════════════════════════════

TEMPO PARA MUDANÇAS:
═══════════════════════════════════════════════════════════════════════════

Adicionar campo SOAP                  Sem: 2 horas → Com: 15 min  ⚡
Trocar banco de dados                 Sem: 1 semana → Com: 2 horas ⚡
Adicionar novo use case               Sem: 4 horas → Com: 30 min ⚡
Escrever testes                       Sem: não faz → Com: 1 hora ⚡
Onboard novo desenvolvedor            Sem: 3 semanas → Com: 2 dias ⚡

═══════════════════════════════════════════════════════════════════════════
"""

print(comparison_diagram)
```

🎁 BÔNUS: Exemplos de Código Real

Exemplo 1: Entidade Patient (Domain Layer)

Arquivo: `src/domain/patient/patient_entity.py`

```python
from datetime import datetime, date
from typing import Optional

class Patient:
    """
    Entidade que encapsula um paciente e suas regras de negócio.
    Nenhuma dependência de banco de dados, web framework ou ORM.
    """

    def __init__(
        self,
        id: str,
        name: str,
        date_of_birth: datetime,
        gender: str,
        cpf: str,
        email: str,
        phone: str,
        address: str,
        city: str,
        state: str
    ):
        self.id = id
        self.name = name
        self.date_of_birth = date_of_birth
        self.gender = gender  # M, F, O, N
        self.cpf = cpf
        self.email = email
        self.phone = phone
        self.address = address
        self.city = city
        self.state = state

    def calculate_age(self) -> int:
        """Calcula a idade do paciente em anos."""
        today = date.today()
        age = today.year - self.date_of_birth.year
        if (today.month, today.day) < (self.date_of_birth.month,
self.date_of_birth.day):
            age -= 1
        return age

    def update_contact_info(self, email: str, phone: str) -> None:
```

```
        """Atualiza informações de contato com validação."""
        if not self._is_valid_email(email):
            raise ValueError(f"Invalid email: {email}")
        if not self._is_valid_phone(phone):
            raise ValueError(f"Invalid phone: {phone}")

        self.email = email
        self.phone = phone

    @staticmethod
    def _is_valid_email(email: str) -> bool:
        return "@" in email and "." in email

    @staticmethod
    def _is_valid_phone(phone: str) -> bool:
        return len(phone.replace("-", "").replace(" ", "")) >= 10
```

Exemplo 2: Use Case (Application Layer)

Arquivo: `src/application/patient/register_patient_usecase.py`

```
from datetime import datetime
from typing import Optional
from uuid import uuid4

from src.domain.patient.patient_entity import Patient
from src.domain.patient.patient_repository_interface import
PatientRepositoryInterface
from src.domain.__seedwork.use_case_interface import UseCase


class RegisterPatientDTO:
    def __init__(
        self,
        name: str,
        date_of_birth: datetime,
        gender: str,
        cpf: str,
        email: str,
        phone: str,
        address: str,
        city: str,
        state: str
    ):
        self.name = name
        self.date_of_birth = date_of_birth
        self.gender = gender
        self.cpf = cpf
        self.email = email
        self.phone = phone
        self.address = address
        self.city = city
```

```
        self.state = state


class RegisterPatientOutputDTO:
    def __init__(self, patient_id: str, message: str):
        self.patient_id = patient_id
        self.message = message


class RegisterPatientUseCase(UseCase[RegisterPatientDTO,
RegisterPatientOutputDTO]):
    """
    Caso de uso para registrar um novo paciente.
    Orquestra a criação de uma entidade Patient e sua persistência.
    """

    def __init__(self, patient_repository:
PatientRepositoryInterface):
        self._repository = patient_repository

    def execute(self, input_dto: RegisterPatientDTO) ->
RegisterPatientOutputDTO:
        # 1. Validar entrada
        self._validate_input(input_dto)

        # 2. Criar entidade Patient
        patient = Patient(
            id=str(uuid4()),
            name=input_dto.name,
            date_of_birth=input_dto.date_of_birth,
            gender=input_dto.gender,
            cpf=input_dto.cpf,
            email=input_dto.email,
            phone=input_dto.phone,
            address=input_dto.address,
            city=input_dto.city,
            state=input_dto.state
        )

        # 3. Persistir via repositório (injeção de dependência)
        self._repository.add(patient)

        # 4. Retornar resultado
        return RegisterPatientOutputDTO(
            patient_id=patient.id,
            message=f"Patient {input_dto.name} registered
successfully"
        )

    def _validate_input(self, input_dto: RegisterPatientDTO) ->
None:
        """Validação de regras de negócio da aplicação."""
```

```
        if not input_dto.name or len(input_dto.name) < 3:
            raise ValueError("Name must have at least 3 characters")

        if not self._is_valid_cpf(input_dto.cpf):
            raise ValueError("Invalid CPF")

        if input_dto.gender not in ["M", "F", "O", "N"]:
            raise ValueError("Invalid gender")

    @staticmethod
    def _is_valid_cpf(cpf: str) -> bool:
        """Validação básica de CPF (11 dígitos)."""
        cleaned = cpf.replace("-", "").replace(".", "")
        return len(cleaned) == 11 and cleaned.isdigit()
```

Exemplo 3: Controller Router (Interface Adapter Layer)

Arquivo: `src/infra/api/routers/patient_routers.py`

```
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session

from src.application.patient.register_patient_usecase import (
    RegisterPatientUseCase,
    RegisterPatientDTO,
)
from src.infra.api.presenters.patient_presenter import (
    PatientCreateRequest,
    PatientResponse,
)
from src.infra.api.database import get_db
from src.infra.patient.sqlalchemy.patient_repository import
PatientRepository


router = APIRouter(prefix="/api/v1/patients", tags=["patients"])


@router.post("/", response_model=PatientResponse)
def create_patient(
    request: PatientCreateRequest,  # Pydantic valida
automaticamente
    db: Session = Depends(get_db),
):
    """Registrar um novo paciente."""
    try:
        # Injetar dependência
        repository = PatientRepository(db)
        use_case = RegisterPatientUseCase(repository)

        # Executar use case
        output = use_case.execute(
```

```
            RegisterPatientDTO(
                name=request.name,
                date_of_birth=request.date_of_birth,
                gender=request.gender,
                cpf=request.cpf,
                email=request.email,
                phone=request.phone,
                address=request.address,
                city=request.city,
                state=request.state,
            )
        )

        # Retornar resposta
        return PatientResponse(
            patient_id=output.patient_id,
            message=output.message,
        )
    except ValueError as e:
        raise HTTPException(status_code=400, detail=str(e))
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error:
{str(e)}")


@router.get("/{patient_id}", response_model=PatientResponse)
def get_patient(patient_id: str, db: Session = Depends(get_db)):
    """Buscar paciente por ID."""
    repository = PatientRepository(db)
    patient = repository.find_by_id(patient_id)

    if not patient:
        raise HTTPException(status_code=404, detail="Patient not
found")

    return PatientResponse(
        patient_id=patient.id,
        message="Patient found",
        # ... outros campos
    )


@router.get("/", response_model=list[PatientResponse])
def list_patients(db: Session = Depends(get_db)):
    """Listar todos os pacientes."""
    repository = PatientRepository(db)
    patients = repository.find_all()

    return [
        PatientResponse(
            patient_id=p.id,
            message="Patient found",
```

```
            # ... outros campos
        )
        for p in patients
    ]
```

Exemplo 4: DTO Pydantic (Presenter)

Arquivo: `src/infra/api/presenters/patient_presenter.py`

```python
from datetime import datetime
from pydantic import BaseModel, Field, validator


class PatientCreateRequest(BaseModel):
    """
    DTO para criar um novo paciente.
    Pydantic valida automaticamente tipos e regras.
    """
    name: str = Field(..., min_length=3, max_length=255)
    date_of_birth: datetime
    gender: str = Field(..., regex="^[MFON]$")  # M, F, O, N
    cpf: str = Field(..., regex="^[0-9]{11}$")
    email: str
    phone: str = Field(..., min_length=10)
    address: str
    city: str
    state: str = Field(..., min_length=2, max_length=2)

    @validator("email")
    def validate_email(cls, v):
        if "@" not in v:
            raise ValueError("Invalid email")
        return v

    class Config:
        json_schema_extra = {
            "example": {
                "name": "João Silva",
                "date_of_birth": "1990-05-15T00:00:00",
                "gender": "M",
                "cpf": "12345678901",
                "email": "joao@example.com",
                "phone": "11999999999",
                "address": "Rua A, 123",
                "city": "São Paulo",
                "state": "SP",
            }
        }


class PatientResponse(BaseModel):
    """DTO para resposta de paciente."""
```

```
    patient_id: str
    message: str

    class Config:
        json_schema_extra = {
            "example": {
                "patient_id": "uuid-12345",
                "message": "Patient João Silva registered
successfully",
            }
        }
```

Exemplo 5: Repository (Data Access - Infrastructure)

Arquivo: `src/infra/patient/sqlalchemy/patient_repository.py`

```
from typing import List, Optional
from sqlalchemy.orm import Session

from src.domain.patient.patient_entity import Patient
from src.domain.patient.patient_repository_interface import
PatientRepositoryInterface
from src.infra.patient.sqlalchemy.patient_model import PatientModel


class PatientRepository(PatientRepositoryInterface):
    """
    Implementação concreta do repositório usando SQLAlchemy.
    Converte entre ORM Model ↔ Domain Entity.
    """

    def __init__(self, db: Session):
        self._db = db

    def add(self, entity: Patient) -> None:
        """Adicionar novo paciente ao banco de dados."""
        model = PatientModel(
            id=entity.id,
            name=entity.name,
            date_of_birth=entity.date_of_birth,
            gender=entity.gender,
            cpf=entity.cpf,
            email=entity.email,
            phone=entity.phone,
            address=entity.address,
            city=entity.city,
            state=entity.state,
        )
        self._db.add(model)
        self._db.commit()

    def find_by_id(self, id: str) -> Optional[Patient]:
```

```
        """Buscar paciente por ID."""
        model = self._db.query(PatientModel).filter(PatientModel.id
== id).first()
        return self._to_domain(model) if model else None

    def find_all(self) -> List[Patient]:
        """Listar todos os pacientes."""
        models = self._db.query(PatientModel).all()
        return [self._to_domain(model) for model in models]

    def _to_domain(self, model: PatientModel) -> Patient:
        """Converter ORM Model para Domain Entity."""
        return Patient(
            id=model.id,
            name=model.name,
            date_of_birth=model.date_of_birth,
            gender=model.gender,
            cpf=model.cpf,
            email=model.email,
            phone=model.phone,
            address=model.address,
            city=model.city,
            state=model.state,
        )
```

⁂ Observações Importantes

1. **Entity é puro**: Nenhuma dependência de DB, frameworks, ou bibliotecas externas
2. **UseCase é independente**: Não sabe como dados são persistidos (repositório é injetado)
3. **Router é fino**: Apenas traduz HTTP ↔ UseCase
4. **DTO é estrutura**: Apenas dados, Pydantic valida
5. **Repository converte**: ORM ↔ Domain entities

Este padrão permite **mudanças localizadas**, **testes rápidos** e **sistemas duráveis**.

⍰ FIM DA DOCUMENTAÇÃO

Você agora tem uma compreensão completa do sistema Prontuário Eletrônico implementado em Clean Architecture!

Próximos passos:

1. Explorar o código em `./src/`
2. Executar `python -m uvicorn src.infra.api.main:app --reload`
3. Acessar http://localhost:8000/docs

4. Tentar fazer requisições HTTP
5. Estudar os testes em `tests.py`

Boa sorte com seu TCC!