Estrutura Geral do Projeto — Sistema de Triagem Hospitalar

Documento com a organização de pastas, assinaturas de funções (o que recebem e retornam) e pontos marcados para completar código (TODO).

Organização de pastas

```
sistema_triagem/
├── core/
│   ├── __init__.py
│   ├── paciente.py          # modelos de domínio
│   ├── triagem.py           # fila (deque)
│   └── historico.py         # pilha
├── data/
│   ├── __init__.py
│   ├── persistencia.py     # leitura/gravação JSON
│   └── hash_table_service.py# CRUD via CPF
├── services/
│   ├── __init__.py
│   ├── triagem_service.py   # lógica de cadastro/triagem que usa data
│   └── atendimento_service.py # lógica de atendimento e histórico
├── gateway/
│   ├── __init__.py
│   ├── main.py              # CLI orquestrador
│   └── gui.py               # (opcional) Tkinter / Flask
├── tests/
│   ├── test_core.py
│   ├── test_data.py
│   └── test_services.py
├── data/pacientes.json      # DB local (opcional)
├── requirements.txt
├── README.md
└── LICENSE
```

Convenções

- Tipagem simples (hint com `->` e `: type`) em todas as funções públicas.
- Sempre validar inputs e lançar exceções claras (`ValueError`, `KeyError`, `RuntimeError`).
- Marcar locais que precisam ser completados com `# TODO:` e comentário explicando o que falta.

`core/paciente.py`

```
class Paciente:
    def __init__(self, cpf: str, nome: str, idade: int, prioridade: str =
"normal") -> None:
        """Inicializa um paciente.
        Args:
            cpf: CPF único (string com apenas dígitos ou formatado).
            nome: Nome completo.
            idade: Idade em anos (int >= 0).
            prioridade: 'emergência'|'urgente'|'normal' (case-insensitive).
        Returns: None
        """

    def to_dict(self) -> dict:
        """Retorna dicionário serializável para persistência."""

    @staticmethod
    def from_dict(data: dict) -> "Paciente":
        """Cria Paciente a partir do dicionário salvo.
        Args: data com chaves 'cpf','nome','idade','prioridade'.
        Returns: Paciente
        """
```

Local para completar: implementar validações (CPF, idade, prioridade) e mensagens de erro.

`core/triagem.py` (fila)

```
from collections import deque
from typing import Deque, List

class Triagem:
    def __init__(self) -> None:
        self.fila: Deque[str] = deque()  # guarda CPFs

    def adicionar_paciente(self, cpf: str) -> None:
        """Adiciona CPF ao final da fila. Valida formato do CPF."""

    def remover_paciente(self) -> str | None:
        """Remove e retorna o CPF do primeiro paciente. Retorna None se
vazia."""

    def listar(self) -> List[str]:
        """Retorna lista de CPFs na fila (ordem FIFO)."""

    def __len__(self) -> int:
        """Número de pacientes na fila."""
```

Local para completar: proteção contra duplicatas (decidir se aceita CPF duplicado na fila) e testes.

`core/historico.py` (pilha)

```python
from typing import List, Dict

class Historico:
    def __init__(self) -> None:
        self.pilha: List[Dict] = []  # lista de dicionários de paciente

    def adicionar(self, paciente_dict: Dict) -> None:
        """Empilha registro de atendimento (último atendimento no topo)."""

    def listar(self) -> List[Dict]:
        """Retorna histórico em ordem do último para o primeiro."""

    def top(self) -> Dict | None:
        """Retorna último atendimento sem remover; None se vazio."""
```

Local para completar: serialização ao salvar histórico (se necessário) e limites (ex.: manter só N últimos registros).

`data/persistencia.py`

```python
import json
import os
from typing import Any

class Persistencia:
    def __init__(self, arquivo_json: str) -> None:
        self.arquivo_json = arquivo_json
        os.makedirs(os.path.dirname(arquivo_json), exist_ok=True)

    def salvar(self, dados: Any) -> None:
        """Salva objeto serializável em JSON no caminho definido.
        Args: dados: objeto serializável (ex.: dict ou list).
        Returns: None
        """

    def carregar(self) -> Any:
        """Carrega e retorna conteúdo do JSON. Retorna estrutura vazia /
padrão se não existir.
        Returns: objeto (dict/list) carregado do arquivo.
        """
```

Local para completar: tratamento de concorrência (locks) se necessário; validação de schema do JSON.

data/hash_table_service.py

```python
from typing import Dict, Optional, List
from core.paciente import Paciente
from data.persistencia import Persistencia

class HashTableService:
    def __init__(self, arquivo_json: str = "data/pacientes.json") -> None:
        self.persistencia = Persistencia(arquivo_json)
        self.tabela: Dict[str, Dict] = self.persistencia.carregar() or {}

    def adicionar_paciente(self, paciente: Paciente) -> None:
        """Args: paciente: Paciente -> salva na tabela por cpf. Overwrite se
já existir.
        Returns: None
        """

    def buscar_paciente(self, cpf: str) -> Optional[Dict]:
        """Retorna dict do paciente ou None."""

    def listar_pacientes(self) -> List[Dict]:
        """Retorna lista de pacientes (valores da tabela)."""

    def remover_paciente(self, cpf: str) -> bool:
        """Remove paciente por CPF. Retorna True se removido, False se não
existia."""
```

Local para completar: validar formato do CPF; decidir comportamento de sobrescrita; adicionar métodos extras (atualizar campos, buscar por nome).

---

services/triagem_service.py

```python
from core.triagem import Triagem
from core.paciente import Paciente
from data.hash_table_service import HashTableService
from typing import Optional, List

class TriagemService:
    def __init__(self, data_service: HashTableService) -> None:
        self.data_service = data_service
        self.triagem = Triagem()

    @property
    def fila(self):
        return self.triagem.fila
```

```
    def cadastrar_paciente(self, cpf: str, nome: str, idade: int, prioridade:
str = "normal") -> None:
        """1) valida se paciente já existe; 2) cria Paciente; 3) adiciona em
data_service e fila."""

    def triar(self, cpf: str, prioridade: Optional[str] = None) -> str:
        """Define/atualiza prioridade do paciente cadastrado.
        Args: cpf: str, prioridade: opcional (se None, aplicar regra
automática).
        Returns: prioridade definida (string).
        Raises: KeyError se paciente não existir.
        """

    def listar_fila(self) -> List[dict]:
        """Retorna lista de dicts com informações dos CPFs na fila (para
exibição)."""
```

Local para completar: lógica automática de triagem (baseada em sintomas/idade) se desejado, validações, mensagens de erro amigáveis.

---

services/atendimento_service.py

```
from core.historico import Historico
from data.hash_table_service import HashTableService
from typing import Optional
from collections import deque

class AtendimentoService:
    def __init__(self, data_service: HashTableService) -> None:
        self.data_service = data_service
        self.historico = Historico()
        self._fila_externa: Optional[deque] = None

    def bind_fila(self, fila_obj: deque) -> None:
        """Vincula (injeção de dependência) a fila mantida pela triagem."""

    def chamar_proximo(self) -> Optional[dict]:
        """Seleciona próximo paciente por ordem de prioridade e registra no
histórico.
        Returns: paciente dict atendido ou None se fila vazia.
        """

    def mostrar_historico(self) -> list:
        """Retorna histórico (lista de dicts) do último para o primeiro."""
```

Local para completar: política de desempate dentro da mesma prioridade (FIFO), logs e testes de integração fila↔atendimento.

`gateway/main.py` (CLI)

Principais responsabilidades: - Criar instâncias (HashTableService, TriagemService, AtendimentoService) - Fazer injeção de dependência: `atendimento.bind_fila(triagem_service.triagem.fila)` - Loop de CLI com menu

Funções/assinaturas esperadas:

```
def build_system(persist_path: str = "data/pacientes.json") -> tuple:
    """Cria e retorna (data_service, triagem_svc, atendimento_svc)."""

def menu() -> None:
    """Loop principal do CLI (input/print)."""
```

Local para completar: tratar Ctrl+C (KeyboardInterrupt), salvar estado ao encerrar, mensagens de ajuda.

`gateway/gui.py` (sugestão Tkinter)

Função pública:

```
def iniciar_interface(triagem_svc, atendimento_svc) -> None:
    """Cria janela e vincula callbacks dos botões às operações dos services.
    Não deve conter lógica de negócio, apenas apresentação e validação
básica.
    """
```

Local para completar: telas de validação (formato CPF), atualização visual da fila/histórico em tempo real.

Testes sugeridos

- `tests/test_core.py`: testar validações e conversões (`Paciente.to_dict()/from_dict()`), comportamento da fila/pilha.
- `tests/test_data.py`: testar salvar/carregar, adicionar/buscar/remover na HashTableService (usar tmpdir/fixtures).
- `tests/test_services.py`: testes de integração simples: cadastrar -> triar -> chamar_proximo -> histórico.

Cada teste deve documentar pré-condição, ação e resultado esperado.

Locais marcados para completar (resumo)

- `core/paciente.py`: validações de CPF/idade/prioridade.
- `core/triagem.py`: políticas sobre duplicatas na fila.
- `core/historico.py`: limite de registros / serialização.
- `data/persistencia.py`: tratamento de arquivo inexistente, concorrência (locks) se necessário.
- `data/hash_table_service.py`: atualizações/consulta por nome.
- `services/triagem_service.py`: lógica de triagem automática (opcional) e tratamento de erros.
- `services/atendimento_service.py`: desempate FIFO e logs.
- `gateway/gui.py`: implementação das telas e conexões com callbacks.
- `tests/`: escrever testes unitários e de integração.

Recomendações finais

1. Comecem implementando e testando `core` + `data` antes de `services` e `gateway`.
2. Documentem as funções com docstrings (args/returns/raises).
3. Façam commits pequenos e frequentes por feature (ex.: `feature/core-paciente`).
4. Se desejarem GUI, priorizem Tkinter para entrega rápida; Flask se quiserem demo via navegador.

Se quiser, eu gero um *template* de arquivos (`.py`) com stubs e `# TODO` nos pontos indicados para o time começar — quer que eu gere esses arquivos agora?