RELATORIO

Programação de Sistemas de informação

Trabalho Avaliativo

CURSO DE GESTÃO E PROGRAMAÇÃO DE SISTEMAS INFORMATICOS

Professor: Breno Sousa

Nome do Aluno: Kauã de Andrade Simão/Kaique de Andrade Simão

NºAluno: L2466/L2465

23/10/2025

O relatório encontra-se em condições para se apresentado

_____

Ciclo de Formação 2023/2026

Ano letivo 2025/2026

Índice

Introdução

O presente projeto foi desenvolvido pelos alunos Kauã e Kaique em conjunto, como parte do 1º Trabalho Avaliativo da disciplina de PSI – Módulo 11, sob orientação do professor Breno Sousa.

O principal objetivo deste trabalho foi a criação de um sistema de gestão hospitalar utilizando a linguagem Python, aplicando conceitos fundamentais de Programação Orientada a Objetos (POO), tais como herança simples e múltipla, polimorfismo, classes abstratas e uso de módulos. Além disso, fez-se uso de estruturas de dados (listas, dicionários e tuplas) e laços de repetição, conforme solicitado na proposta.

Durante o desenvolvimento, trabalhamos de forma colaborativa, enfrentando e superando diversas dificuldades técnicas e conceituais. Por meio de pesquisa, troca de ideias e testes práticos, conseguimos implementar todas as funcionalidades exigidas, incluindo o registo de pacientes e funcionários, a gestão de consultas e salas, e a aplicação dos princípios de POO de maneira estruturada e funcional.

Este relatório apresenta a descrição do processo de desenvolvimento, as classes e métodos implementados, bem como uma análise dos resultados obtidos ao final do projeto.

Conteúdo do relatório

Hy.py

```
1   from abc import ABC, abstractmethod # Faz a Importação do abstractmethod
```

Já no começo do codígo é feita a importação do abstractmethod

De seguida são criadas as classes

Classe Pessoa (Abstrata):

```
class Pessoa(ABC):
    def __init__(self, nome, idade):
        self._nome = None
        self._idade = None
        self.nome = nome
        self.idade = idade

    @property
    def nome(self):
        return self._nome

    @nome.setter
    def nome(self, valor):
        self._nome = valor

    @property
    def idade(self):
        return self._idade

    @idade.setter
    def idade(self, valor):
        self._idade = valor

    @abstractmethod
    def exibir_informacoes(self):
        pass
```

Init(self, nome, idade): inicializa nome e idade.

@property nome: retorna o nome da pessoa.

@nome.setter: define o nome, apenas se não for vazio.

@property idade: retorna a idade da pessoa.

@idade.setter: define a idade, apenas se for positiva.

exibir_informacoes(self): método abstrato que foi implementado nas subclasses.

Classe Sala(Abstrata):

```python
class Sala(ABC):
    def __init__(self, numero, capacidade):
        self.numero = numero
        self.capacidade = capacidade

    @abstractmethod
    def exibir_informacoes(self):
        pass
```

Já foi criada de início para melhor organização do código em questão de que apenas 2 métodos abstratos foram pedidos

init(self, número, capacidade): inicializa a sala com número e capacidade máxima.

```python
    @property
    def numero(self):
        return self._numero

    @numero.setter
    def numero(self, valor):
        if isinstance(valor, int) and valor > 0:
            self._numero = valor
        else:
            raise ValueError("O número da sala deve ser um inteiro positivo.")

    @property
    def capacidade(self):
        return self._capacidade

    @capacidade.setter
    def capacidade(self, valor):
        if isinstance(valor, int) and valor > 0:
            self._capacidade = valor
        else:
            raise ValueError("A capacidade da sala deve ser um inteiro maior que zero.")

    @abstractmethod
    def exibir_informacoes(self):
        pass
```

@property numero: retorna o número da sala.

@numero.setter: define o número apenas se for positivo.

@property capacidade: retorna a capacidade da sala.

@capacidade.setter: define capacidade apenas se for maior que zero.

detalhar_sala(self): método abstrato para descrição da sala.

Classe Paciente:

```python
class Sala(ABC):

    @abstractmethod
    def nome(self):
        pass

class Paciente(Pessoa):
    def __init__(self, nome, idade, numero_utente):
        super().__init__(nome, idade)
        self._numero_utente = numero_utente
        self._historico_medico = []

    @property
    def numero_utente(self):
        return self._numero_utente

    def exibir_informacoes(self):
        print(f"Nome: {self.nome}")
        print(f"Idade: {self.idade}")
        print(f"Nº Utente: {self.numero_utente}")
        print(f"Histórico médico: {self._historico_medico if self._historico_medico else 'Nenhum registro'}")
```

init(self, nome, idade, numero_utente): inicializa paciente com nome, idade e número de utente.

@property numero_utente: retorna o número de utente.

adicionar_registro(self, descricao): adiciona uma entrada ao histórico médico.

mostrar_historico(self): exibe o histórico médico do paciente.

exibir_informacoes(self): mostra nome, idade e número de utente.

Classe Funcionario:

```python
class Funcionario(Pessoa):
    def __init__(self, nome, idade, cargo, salario):
        super().__init__(nome, idade)
        self.cargo = cargo
        self._salario = None
        self.salario = salario  # usa o setter para validar

    @property
    def salario(self):
        return self._salario

    @salario.setter
    def salario(self, valor):
        try:
            valor = float(valor)
            if valor >= 0:
                self._salario = valor
            else:
                raise ValueError("O salário deve ser um valor positivo.")
        except ValueError:
            print("Erro: salário inválido. Deve ser um número positivo.")
```

init(self, nome, idade, cargo, salario): inicializa dados do funcionário.

```python
def exibir_informacoes(self):
    print(f"Nome: {self.nome}")
    print(f"Idade: {self.idade}")
    print(f"Cargo: {self.cargo}")
    print(f"Salário: €{self.salario:.2f}")

def aplicar_aumento(self, percentual):
    if percentual > 0:
        aumento = self.salario * (percentual / 100)
        self.salario += aumento
        print(f"Aumento aplicado: +{aumento:.2f}€")
        print(f"Novo salário: €{self.salario:.2f}")
    else:
        raise ValueError("O percentual de aumento deve ser positivo.")
```

@property salario: retorna o salário atual.

@salario.setter: atualiza o salário, impedindo valores negativos.

mostrar_informacoes(self): exibe nome, cargo e salário.

aplicar_aumento(self, percentual): aumenta o salário com base em um percentual.

Classe Medico:

```
class Medico(Pessoa):
    def __init__(self, nome, idade, salario_base, especialidade):
        super().__init__(nome, idade)
        self._salario_base = salario_base
        self._especialidade = None
        self.especialidade = especialidade  # usa o setter para validar
        self._pacientes = []
```

init(self, nome, idade, salario_base, especialidade): inicializa médico com especialidade e salário base.

```
    @property
    def salario_base(self):
        return self._salario_base

    @property
    def especialidade(self):
        return self._especialidade

    @especialidade.setter
    def especialidade(self, valor):
        if isinstance(valor, str) and valor.strip():
            self._especialidade = valor.strip()
        else:
            raise ValueError("Especialidade deve ser uma string não vazia.")
```

@property especialidade: retorna a especialidade do médico.

@especialidade.setter: define a especialidade, desde que não seja vazia.

```python
    def adicionar_paciente(self, paciente):
        self._pacientes.append(paciente)

    def listar_pacientes(self):
        if self._pacientes:
            print("Pacientes atendidos:")
            for i, paciente in enumerate(self._pacientes, 1):
                print(f"{i}. {paciente.nome} ({paciente.idade} anos)")
        else:
            print("Nenhum paciente atendido ainda.")

    def calcular_pagamento(self):
        valor_por_paciente = 50
        adicional = self._salario_base + len(self._pacientes) * valor_por_paciente
        pagamento_total = self._salario_base + adicional
        print("-------------------------------------")
        print("             PAGAMENTO             ")
        print(f"Salário base: €{self._salario_base:.2f}")
        print(f"Adicional por paciente atendido: €{adicional:.2f}")
        print(f"Pagamento total: €{pagamento_total:.2f}")
        print("-------------------------------------")
        return pagamento_total

    def exibir_informacoes(self):
        print(f"Nome: {self.nome}")
        print(f"Especialidade: {self.especialidade}")
        print(f"Número de pacientes atendidos: {len(self._pacientes)}")
```

adicionar_paciente(self, paciente): adiciona paciente à lista de atendidos.

listar_pacientes(self): mostra todos os pacientes do médico.

calcular_pagamento(self): retorna salário base + valor fixo por paciente atendido.

exibir_informacoes(self): exibe nome, especialidade e número de pacientes.

Classe Enfermeiro:

```python
class Enfermeiro(Pessoa):
    def __init__(self, nome, idade, salario_base, turno):
        super().__init__(nome, idade)
        self._salario_base = salario_base
        self._turno = None
        self.turno = turno  # usa o setter para validar
        self._pacientes = []
```

init(self, nome, idade, salario_base, turno): inicializa enfermeiro com turno (dia/noite).

```python
@property
def turno(self):
    return self._turno

@turno.setter
def turno(self, valor):
    if valor.lower() in ["dia", "noite"]:
        self._turno = valor.lower()
    else:
        raise ValueError("Turno inválido. Use 'dia' ou 'noite'.")
```

@property turno: retorna o turno atual.

@turno.setter: define o turno apenas se for "dia" ou "noite".

```python
    def adicionar_paciente(self, paciente):
        self._pacientes.append(paciente)

    def listar_pacientes(self):
        if self._pacientes:
            print("Pacientes sob responsabilidade:")
            for i, paciente in enumerate(self._pacientes, 1):
                print(f"{i}. {paciente.nome} ({paciente.idade} anos)")
        else:
            print("Nenhum paciente sob responsabilidade.")

    def calcular_pagamento(self):
        adicional = 100 if self.turno == "noite" else 50
        pagamento_total = self._salario_base + adicional
        print("-------------------------------------")
        print("             PAGAMENTO             ")
        print(f"Salário base: €{self._salario_base:.2f}")
        print(f"Adicional por turno ({self.turno}): €{adicional:.2f}")
        print(f"Pagamento total: €{pagamento_total:.2f}")
        print("-------------------------------------")
        return pagamento_total

    def exibir_informacoes(self):
        print(f"Nome: {self.nome}")
        print(f"Idade: {self.idade}")
        print(f"Turno: {self.turno}")
        print(f"Número de pacientes sob cuidado: {len(self._pacientes)}")
```

adicionar_paciente(self, paciente): adiciona paciente sob cuidado.

listar_pacientes(self): exibe pacientes sob responsabilidade.

calcular_pagamento(self): retorna salário base + adicional conforme o turno.

exibir_informacoes(self): exibe nome, turno e total de pacientes.

Classe Administrativo

```python
class Administrativo(Pessoa):
    def __init__(self, nome, idade, salario_base, setor):
        super().__init__(nome, idade)
        self._salario_base = salario_base
        self._setor = None
        self.setor = setor  # usa o setter para validar
        self._horas_trabalhadas = 0
```

```python
    @property
    def setor(self):
        return self._setor

    @setor.setter
    def setor(self, valor):
        setores_validos = ["financeiro", "recursos humanos", "logística", "atendimento"]
        if valor.lower() in setores_validos:
            self._setor = valor.lower()
        else:
            raise ValueError(f"Setor inválido. Escolha entre: {', '.join(setores_validos)}")
```

@property setor: retorna o setor de atuação.

@setor.setter: define o setor apenas se for válido.

```python
    def registrar_horas(self, horas):
        if horas > 0:
            self._horas_trabalhadas += horas
        else:
            raise ValueError("Horas devem ser um valor positivo.")

    def calcular_pagamento(self):
        valor_por_hora = 10
        adicional = self._horas_trabalhadas * valor_por_hora
        pagamento_total = self._salario_base + adicional
        print("-------------------------------------")
        print("           PAGAMENTO ADMINISTRATIVO           ")
        print(f"Salário base: €{self._salario_base:.2f}")
        print(f"Horas registradas: {self._horas_trabalhadas}")
        print(f"Adicional por horas: €{adicional:.2f}")
        print(f"Pagamento total: €{pagamento_total:.2f}")
        print("-------------------------------------")
        return pagamento_total

    def exibir_informacoes(self):
        print(f"Nome: {self.nome}")
        print(f"Idade: {self.idade}")
        print(f"Setor: {self.setor}")
        print(f"Horas trabalhadas: {self._horas_trabalhadas}")
```

registrar_horas(self, horas): acumula horas trabalhadas.

calcular_pagamento(self): retorna salário base + valor por hora registrada.

exibir_informacoes(self): mostra nome, setor e total de horas trabalhadas.

Classe EnfermeiroChefe:

```python
class EnfermeiroChefe(Pessoa):
    def __init__(self, nome, idade, salario_base, turno, setor, bonus_chefia):
        super().__init__(nome, idade)
        self._salario_base = salario_base
        self._turno = None
        self.turno = turno  # usa o setter
        self._setor = None
        self.setor = setor  # usa o setter
        self._bonus_chefia = None
        self.bonus_chefia = bonus_chefia  # usa o setter
        self._pacientes = []
```

init(self, nome, idade, salario_base, turno, setor, bonus_chefia): inicializa o híbrido com dados de enfermeiro e administrativo.

```python
    @property
    def turno(self):
        return self._turno

    @turno.setter
    def turno(self, valor):
        if valor.lower() in ["dia", "noite"]:
            self._turno = valor.lower()
        else:
            raise ValueError("Turno inválido. Use 'dia' ou 'noite'.")

    @property
    def setor(self):
        return self._setor

    @setor.setter
    def setor(self, valor):
        setores_validos = ["emergência", "pediatria", "cirurgia", "clínica geral"]
        if valor.lower() in setores_validos:
            self._setor = valor.lower()
        else:
            raise ValueError(f"Setor inválido. Escolha entre: {', '.join(setores_validos)}")

    @property
    def bonus_chefia(self):
        return self._bonus_chefia

    @bonus_chefia.setter
    def bonus_chefia(self, valor):
        if isinstance(valor, (int, float)) and valor >= 0:
            self._bonus_chefia = valor
        else:
            raise ValueError("O bônus de chefia deve ser um número positivo.")
```

@property bonus_chefia: retorna o bônus adicional.

@bonus_chefia.setter: define o bônus apenas se for positivo.

```python
    def listar_pacientes(self):
        if self._pacientes:
            print("Pacientes sob responsabilidade:")
            for i, paciente in enumerate(self._pacientes, 1):
                print(f"{i}. {paciente.nome} ({paciente.idade} anos)")
        else:
            print("Nenhum paciente sob responsabilidade.")

    def calcular_pagamento(self):
        adicional_turno = 100 if self.turno == "noite" else 50
        pagamento_total = self._salario_base + adicional_turno + self.bonus_chefia
        print("--------------------------------------")
        print("         PAGAMENTO ENFERMEIRO CHEFE         ")
        print(f"Salário base: €{self._salario_base:.2f}")
        print(f"Turno: {self.turno}")
        print(f"Adicional por turno: €{adicional_turno:.2f}")
        print(f"Bônus de chefia: €{self.bonus_chefia:.2f}")
        print(f"Pagamento total: €{pagamento_total:.2f}")
        print("--------------------------------------")
        return pagamento_total

    def exibir_informacoes(self):
        print(f"Nome: {self.nome}")
        print(f"Idade: {self.idade}")
        print(f"Turno: {self.turno}")
        print(f"Setor: {self.setor}")
        print(f"Número de pacientes sob cuidado: {len(self._pacientes)}")
```

calcular_pagamento(self): combina o pagamento de enfermeiro e administrativo + bônus de chefia.

exibir_informacoes(self): mostra nome, turno, setor e pacientes sob cuidado.

Classe SalaConsulta (herda de Sala)

```python
class SalaConsulta(Sala):
    def __init__(self, numero, capacidade, medico_responsavel):
        super().__init__(numero, capacidade)
        self._medico_responsavel = None
        self.medico_responsavel = medico_responsavel
        self._pacientes_agendados = []
```

init(self, número, capacidade, medico_responsavel): inicializa sala de consulta com médico e pacientes.

```python
    @property
    def medico_responsavel(self):
        return self._medico_responsavel

    @medico_responsavel.setter
    def medico_responsavel(self, valor):
        if isinstance(valor, Medico):
            self._medico_responsavel = valor
        else:
            raise TypeError("O responsável deve ser uma instância da classe Medico.")

    @property
    def pacientes_agendados(self):
        return self._pacientes_agendados
```

@property medico_responsavel: retorna o médico responsável.

@medico_responsavel.setter: define o médico apenas se for uma instância válida de Médico.

```python
    def agendar_consulta(self, paciente):
        if not isinstance(paciente, Paciente):
            raise TypeError("O paciente deve ser uma instância da classe Paciente.")

        if paciente in self._pacientes_agendados:
            print(f"{paciente.nome} já está agendado para esta sala.")
            return

        if len(self._pacientes_agendados) < self.capacidade:
            self._pacientes_agendados.append(paciente)
            self.medico_responsavel.adicionar_paciente(paciente)
            print(f"Consulta agendada para {paciente.nome}.")
        else:
            print("Capacidade máxima da sala atingida. Não é possível agendar mais consultas.")

    def detalhar_sala(self):
        print("=== Sala de Consulta ===")
        print(f"Número da sala: {self.numero}")
        print(f"Capacidade: {self.capacidade}")
        print(f"Médico responsável: {self.medico_responsavel.nome}")
        print("Pacientes agendados:")
        if self._pacientes_agendados:
            for i, paciente in enumerate(self._pacientes_agendados, 1):
                print(f"{i}. {paciente.nome} ({paciente.idade} anos)")
        else:
            print("Nenhum paciente agendado.")

    def exibir_informacoes(self):
        self.detalhar_sala()
```

agendar_consulta(self, paciente): adiciona paciente à lista de consultas.

detalhar_sala(self): exibe número, capacidade e nome do médico responsável.

Classe SalaCirurgia (herda de Sala)

```python
class SalaCirurgia(Sala):
    def __init__(self, numero, capacidade, medico_responsavel):
        super().__init__(numero, capacidade)
        self._medico_responsavel = None
        self.medico_responsavel = medico_responsavel  # usa o setter
        self._pacientes_agendados = []
```

init(self, numero, capacidade): inicializa sala de cirurgia com número e capacidade.

```python
    @property
    def medico_responsavel(self):
        return self._medico_responsavel

    @medico_responsavel.setter
    def medico_responsavel(self, valor):
        if isinstance(valor, Medico):
            self._medico_responsavel = valor
        else:
            raise TypeError("O responsável deve ser uma instância da classe Medico.")

    def agendar_consulta(self, paciente):
        if not isinstance(paciente, Paciente):
            raise TypeError("O paciente deve ser uma instância da classe Paciente.")

        if len(self._pacientes_agendados) < self.capacidade:
            self._pacientes_agendados.append(paciente)
            self.medico_responsavel.adicionar_paciente(paciente)
            print(f"Cirurgia agendada para {paciente.nome}.")
        else:
            print("Capacidade máxima da sala atingida. Não é possível agendar mais cirurgias.")
```

Permite agendar consultas para pacientes, verificando a capacidade da sala e associando-os ao médico responsável. Inclui validações para garantir que os objetos sejam instâncias corretas das classes Medico e Paciente. Também exibe detalhes da sala, como número, capacidade, médico e lista de pacientes agendados.

Classe Consulta:

```python
class Consulta:
    def __init__(self, medico, paciente, data, tipo, sala=None):
        self._sala = sala
        if not isinstance(medico, Medico):
            raise TypeError("O médico deve ser uma instância da classe Medico.")
        if not isinstance(paciente, Paciente):
            raise TypeError("O paciente deve ser uma instância da classe Paciente.")
        if not isinstance(data, str) or not data.strip():
            raise ValueError("A data deve ser uma string não vazia.")
        self._medico = medico
        self._paciente = paciente
        self._data = data.strip()
        self._tipo = None
        self.tipo = tipo  # usa o setter
```

init(self, medico, paciente, data, tipo): cria a ligação entre médico e paciente com data e tipo (rotina, emergência, etc.).

```python
    @property
    def tipo(self):
        return self._tipo

    @tipo.setter
    def tipo(self, valor):
        tipos_validos = ["rotina", "emergência", "especialidade", "urgência"]
        if isinstance(valor, str) and valor.strip().lower() in tipos_validos:
            self._tipo = valor.strip().lower()
        else:
            raise ValueError(f"Tipo de consulta inválido. Escolha entre: {', '.join(tipos_validos)}")

    @property
    def medico(self):
        return self._medico

    @property
    def paciente(self):
        return self._paciente

    @property
    def data(self):
        return self._data
```

@property tipo: retorna o tipo da consulta.

@tipo.setter: define o tipo apenas se for string válida.

```python
def exibir_detalhes(self):
    print("=== Detalhes da Consulta ===")
    print(f"Data: {self.data}")
    print(f"Tipo: {self.tipo}")
    print(f"Médico: {self.medico.nome} ({self.medico.especialidade})")
    print(f"Paciente: {self.paciente.nome}, {self.paciente.idade} anos")
```

exibir_detalhes(self): mostra informações do médico, paciente e tipo de consulta.

Hy_main.py

```
import os
from Hy import (
    Paciente, Medico, Enfermeiro, EnfermeiroChefe, Funcionario, Administrativo,
    SalaConsulta, SalaCirurgia, Consulta
)
```

Faz o import das Classes de Hy.py para Hy_main.py.

```
# Listas de armazenamento
pacientes_cadastrados = []
medicos_cadastrados = []
enfermeiros_cadastrados = []
chefes_cadastrados = []
funcionarios_cadastrados = []
administradores_cadastrados = []
consultas_realizadas = []
```

```python
def entrada_obrigatoria(mensagem):
    while True:
        valor = input(mensagem).strip()
        if valor:
            return valor
        else:
            print("Este campo não pode ficar vazio. Digite um valor válido.")

def adicionar_historico(self, descricao):
    if isinstance(descricao, str) and descricao.strip():
        self._historico_medico.append(descricao.strip())

def entrada_obrigatoria(mensagem):
    while True:
        valor = input(mensagem).strip()
        if valor:
            return valor
        else:
            print("Este campo não pode ficar vazio.")

def registrar_consulta():
    excluir()
    print("=== Registrar Nova Consulta ===")

    if not pacientes_cadastrados or not medicos_cadastrados:
        print("Cadastre pelo menos 1 paciente e 1 médico.")
        input("Pressione Enter para continuar...")
        return
```

Esse trecho garante que o usuário forneça entradas obrigatórias, adiciona histórico médico se válido, e inicia o registro de uma nova consulta, verificando se há pacientes e médicos cadastrados antes de prosseguir.

```python
# Escolher médico
print("\nMédicos disponíveis:")
for i, m in enumerate(medicos_cadastrados, 1):
    print(f"{i}. {m.nome} ({m.especialidade})")
try:
    idx_medico = int(input("Escolha o número do médico: "))
    medico = medicos_cadastrados[idx_medico - 1]
except (ValueError, IndexError):
    print("Médico inválido.")
    input("Pressione Enter para continuar...")
    return
```

Esse trecho exibe uma lista de médicos cadastrados, solicita ao usuário que escolha um pelo número correspondente e valida a entrada. Se o número for inválido (fora do intervalo ou não numérico), mostra uma mensagem de erro e interrompe o processo.

```python
# Escolher paciente
print("\nPacientes disponíveis:")
for i, p in enumerate(pacientes_cadastrados, 1):
    print(f"{i}. {p.nome} ({p.idade} anos)")
try:
    idx_paciente = int(input("Escolha o número do paciente: "))
    paciente = pacientes_cadastrados[idx_paciente - 1]
except (ValueError, IndexError):
    print("Paciente inválido.")
    input("Pressione Enter para continuar...")
    return
```

Esse trecho exibe uma lista de pacientes cadastrados, solicita ao usuário que escolha um pelo número, e valida a entrada. Se o número for inválido (não existe ou não é um número), mostra uma mensagem de erro e interrompe o processo.

```python
# Data e tipo
data = entrada_obrigatoria("Data da consulta (ex: 24/10/2025): ")
tipos_validos = ["rotina", "emergência", "especialidade", "urgência"]
while True:
    tipo = entrada_obrigatoria("Tipo de consulta (rotina/emergência/especialidade/urgência): ").lower()
    if tipo in tipos_validos:
        break
    else:
        print("Tipo inválido. Escolha entre:", ", ".join(tipos_validos))
```

Esse trecho coleta a data da consulta e o tipo, garantindo que o tipo informado seja um dos válidos: "rotina", "emergência", "especialidade" ou "urgência". Se o tipo for inválido, solicita novamente até que seja correto.

```python
# Criar e registrar consulta
consulta = Consulta(medico, paciente, data, tipo)
consultas_realizadas.append(consulta)
print("\nConsulta registrada com sucesso!")
consulta.exibir_detalhes()
input("\nPressione Enter para continuar...")
```

Esse trecho cria uma nova instância da classe `Consulta` com os dados fornecidos (médico, paciente, data e tipo), adiciona essa consulta à lista de consultas realizadas, exibe uma mensagem de sucesso e mostra os detalhes da consulta registrada.

```python
def calcular_pagamento(lista, titulo):
    excluir()
    print(f"=== Pagamentos de {titulo} ===")
    if lista:
        for i, pessoa in enumerate(lista, 1):
            print(f"\n{i}. {pessoa.nome}")
            try:
                valor = pessoa.calcular_pagamento()
                print(f"Pagamento total: €{valor:.2f}")
            except Exception as e:
                print(f"Erro ao calcular pagamento: {e}")
    else:
        print("Nenhum cadastro encontrado.")
    input("\nPressione Enter para continuar...")

def pagamento(lista, titulo):
    excluir()
    print(f"=== Pagamentos de {titulo} ===")
    if lista:
        for i, pessoa in enumerate(lista, 1):
            print(f"\n{i}. {pessoa.nome}")
            try:
                valor = pessoa.pagamento()
                print(f"Pagamento total: €{valor:.2f}")
            except Exception as e:
                print(f"Erro ao calcular pagamento: {e}")
    else:
        print("Nenhum cadastro encontrado.")
    input("\nPressione Enter para continuar...")
```

Essas duas funções exibem os pagamentos de pessoas em uma lista, com título personalizado. Ambas percorrem a lista e mostram o nome e o valor do pagamento, tratando erros se houver. A diferença está no método chamado: uma usa `calcular_pagamento()` e a outra `pagamento()`. Se a lista estiver vazia, informam que não há cadastros e aguardam o usuário pressionar Enter.

```python
def alterar_pagamento(lista, titulo):
    excluir()
    print(f"=== Alterar Pagamento de {titulo} ===")
    if not lista:
        print("Nenhum cadastro encontrado.")
        input("Pressione Enter para continuar...")
        return

    for i, pessoa in enumerate(lista, 1):
        print(f"{i}. {pessoa.nome}")

    try:
        escolha = int(input("\nEscolha o número da pessoa para alterar o pagamento: "))
        pessoa = lista[escolha - 1]
    except (ValueError, IndexError):
        print("Escolha inválida.")
        input("Pressione Enter para continuar...")
        return
```

Esse trecho permite alterar o pagamento de uma pessoa em uma lista. Ele exibe os nomes disponíveis, solicita ao usuário que escolha um número correspondente e valida a entrada. Se a lista estiver vazia ou a escolha for inválida, exibe uma mensagem de erro e encerra a função. É uma etapa inicial antes de aplicar a alteração no valor de pagamento da pessoa selecionada.

```python
    # Detecta tipo de pessoa e altera o valor correspondente
    try:
        if hasattr(pessoa, "salario"):
            novo = float(input("Novo salário: "))
            pessoa.salario = novo
        elif hasattr(pessoa, "salario_base"):
            novo = float(input("Novo salário base: "))
            pessoa._salario_base = novo
        if hasattr(pessoa, "bonus_chefia"):
            alterar_bonus = input("Deseja alterar o bônus de chefia? (s/n): ").lower()
            if alterar_bonus == "s":
                novo_bonus = float(input("Novo bônus de chefia: "))
                pessoa.bonus_chefia = novo_bonus
        print("\nPagamento atualizado com sucesso.")
    except Exception as e:
        print(f"Erro ao alterar pagamento: {e}")

    input("Pressione Enter para continuar...")
```

Esse trecho ajusta o pagamento de uma pessoa conforme seus atributos. Se ela tiver `salario`, ou `salario_base`, esses valores são atualizados com novos valores

informados. Se houver bonus_chefia, o usuário pode optar por alterá-lo também. Erros são tratados com mensagens, e ao final, o sistema confirma a atualização e aguarda o usuário continuar.

```
def excluir():
    os.system('cls' if os.name == 'nt' else 'clear')

# Funções de cadastro
def cadastrar_paciente():
    excluir()
    print("=== Cadastro de Paciente ===")
    nome = entrada_obrigatoria("Nome: ")
    while True:
        idade = int(entrada_obrigatoria("Idade: "))
        if idade < 120 and idade >0:
            break
        else:
            print("Insira um valor positivo dentro dos padrões do século")
    while True:
        utente = entrada_obrigatoria("Número do Utente: ").strip()
        if len(utente) == 9 and utente.isdigit():
            break
        else:
            print("O número do utente deve ter pelo menos 9 dígitos numéricos.")
    paciente = Paciente(nome, idade, utente)
    pacientes_cadastrados.append(paciente)
    print("\nPaciente cadastrado com sucesso:")
    paciente.exibir_informacoes()
    input("Pressione Enter para continuar...")
```

Esse trecho define a função cadastrar_paciente, que:

Limpa a tela com excluir() para uma interface mais limpa.

Solicita nome, idade (validando que seja entre 1 e 119), e número do utente (exatamente 9 dígitos).

Cria um objeto Paciente com os dados fornecidos e o adiciona à lista pacientes_cadastrados.

Exibe as informações do paciente cadastrado e aguarda o usuário pressionar Enter para continuar.

```python
def cadastrar_medico():
    excluir()
    print("=== Cadastro de Médico ===")
    nome = entrada_obrigatoria("Nome: ")
    while True:
        idade = int(entrada_obrigatoria("Idade: "))
        if idade < 120 and idade >0:
            break
        else:
            print("Insira um valor positivo dentro dos padrões do século")
    while True:
        salario = float(entrada_obrigatoria("Salário base: "))
        if salario < 10000000 and salario > 0:
            break
        else:
            print("Insira um valor positivo dentro do que o governo pode pagar!")
    especialidade = entrada_obrigatoria("Especialidade: ")
    medico = Medico(nome, idade, salario, especialidade)
    medicos_cadastrados.append(medico)
    print("\nMédico cadastrado com sucesso:")
    medico.exibir_informacoes()
    input("Pressione Enter para continuar...")
```

Esse trecho define a função cadastrar_medico, que:

Limpa a tela com excluir() e inicia o processo de cadastro.

Solicita nome, idade (entre 1 e 119), salário base (positivo e abaixo de 10 milhões), e especialidade.

Cria um objeto Medico com os dados fornecidos e o adiciona à lista medicos_cadastrados.

```python
def cadastrar_enfermeiro():
    excluir()
    print("=== Cadastro de Enfermeiro ===")
    nome = entrada_obrigatoria("Nome: ")
    while True:
        idade = int(entrada_obrigatoria("Idade: "))
        if idade < 120 and idade > 0:
            break
        else:
            print("Insira um valor positivo dentro dos padrões do século")
    while True:
        salario = float(entrada_obrigatoria("Salário base: "))
        if salario < 10000000 and salario > 0:
            break
        else:
            print("Insira um valor positivo dentro do que o governo pode pagar!")
    while True:
        turno = entrada_obrigatoria("Turno (dia/noite): ").lower()
        if turno in ["dia", "noite"]:
            break
        else:
            print("Turno inválido. Digite 'dia' ou 'noite'.")
    enfermeiro = Enfermeiro(nome, idade, salario, turno)
    enfermeiros_cadastrados.append(enfermeiro)
    print("\nEnfermeiro cadastrado com sucesso:")
    enfermeiro.exibir_informacoes()
    input("Pressione Enter para continuar...")
```

Essa função cadastrar_enfermeiro realiza o cadastro de um enfermeiro no sistema. Aqui está um resumo do que ela faz:

Limpa a tela com excluir() e exibe o título do cadastro.

Solicita e valida os dados: nome, idade (entre 1 e 119), salário (positivo e abaixo de 10 milhões) e turno (apenas "dia" ou "noite").

Cria um objeto Enfermeiro com os dados fornecidos e o adiciona à lista enfermeiros_cadastrados.

Exibe as informações do enfermeiro cadastrado e aguarda o usuário pressionar Enter para continuar.

```python
def cadastrar_enfermeiro_chefe():
    excluir()
    print("=== Cadastro de Enfermeiro Chefe ===")
    nome = entrada_obrigatoria("Nome: ")
    while True:
        idade = int(entrada_obrigatoria("Idade: "))
        if idade < 120 and idade >0:
            break
        else:
            print("Insira um valor positivo dentro dos padrões do século")
    while True:
        salario = float(entrada_obrigatoria("Salário base: "))
        if salario < 10000000 and salario > 0:
            break
        else:
            print("Insira um valor positivo dentro do que o governo pode pagar!")
    while True:
        turno = entrada_obrigatoria("Turno (dia/noite): ")
        if turno == "dia" and turno == "noite":
            break
        else:
        else:
            print("Os turnos estão entre o dia/noite")
    setores_validos = ["emergência", "pediatria", "cirurgia", "clínica geral"]
    while True:
        setor = entrada_obrigatoria(f"Setor: [emergência, pediatria, cirurgia, clinica geral] ").lower()
        if setor in setores_validos:
            break
        else:
            print("Setor inválido. Tente: emergência, pediatria, cirurgia, clínica geral.")
    while True:
        bonus = float(entrada_obrigatoria("Bónus de chefia: (Até 1.000€) "))
        if bonus < 1000 and bonus > 0:
            break
        else:
            print("Bonus invalido.")
    chefe = EnfermeiroChefe(nome, idade, salario, turno, setor, bonus)
    chefes_cadastrados.append(chefe)
    print("\nEnfermeiro Chefe cadastrado com sucesso:")
    chefe.exibir_informacoes()
    input("Pressione Enter para continuar...")
```

Essa função realiza o cadastro de um enfermeiro chefe. Ela solicita e valida os seguintes dados: nome, idade (entre 1 e 119), salário base (positivo e abaixo de 10 milhões), turno (deveria aceitar "dia" ou "noite", mas há um erro lógico na verificação), setor (entre opções válidas como emergência, pediatria, cirurgia ou clínica geral) e bônus de chefia (até 1.000€). Após reunir os dados, cria um objeto `EnfermeiroChefe`, adiciona à lista de chefes cadastrados, exibe suas informações e aguarda o usuário pressionar Enter para continuar.

```python
def cadastrar_funcionario():
    excluir()
    print("=== Cadastro de Funcionário ===")
    nome = entrada_obrigatoria("Nome: ")
    while True:
        idade = int(entrada_obrigatoria("Idade: "))
        if idade < 120 and idade >0:
            break
        else:
            print("Insira um valor positivo dentro dos padrões do século")
    cargo = input("Cargo: ")
    while True:
        salario = float(entrada_obrigatoria("Salário base: "))
        if salario < 10000000 and salario > 0:
            break
        else:
            print("Insira um valor positivo dentro do que o governo pode pagar!")
    funcionario = Funcionario(nome, idade, cargo, salario)
    funcionarios_cadastrados.append(funcionario)
    print("\nFuncionário cadastrado com sucesso:")
    funcionario.exibir_informacoes()
    input("Pressione Enter para continuar...")
```

Essa função realiza o cadastro de um funcionário. Ela solicita o nome, idade (entre 1 e 119), cargo e salário base (positivo e abaixo de 10 milhões). Após validar os dados, cria um objeto Funcionario, adiciona à lista de funcionários cadastrados, exibe suas informações e aguarda o usuário pressionar Enter para continuar.

```python
def cadastrar_administrador():
    excluir()
    print("=== Cadastro de Administrador ===")
    nome = entrada_obrigatoria("Nome: ")
    while True:
        idade = int(entrada_obrigatoria("Idade: "))
        if idade < 120 and idade >0:
            break
        else:
            print("Insira um valor positivo dentro dos padrões do século")
    salario = float(input("Salário: "))
    setores_validos = ["financeiro", "recursos humanos", "logística", "atendimento"]
    while True:
        setor = input(f"Setor: [financeiro, recursos humanos, logística, atendimento] ").lower()
        if setor in setores_validos:
            break
        else:
            print("Setor inválido. Tente: financeiro, recursos humanos, logística ou atendimento.")
    while True:
        horas = int(entrada_obrigatoria("Horas trabalhadas: (até 500hrs) "))
        if horas < 500 and horas > 0:
            break
        else:
            print("Suas horas não são validas.")
    administrador = Administrativo(nome, idade, salario, setor)
    administrador.registrar_horas(horas)
    administradores_cadastrados.append(administrador)
    print("\nAdministrador cadastrado com sucesso:")
    administrador.exibir_informacoes()
    administrador.calcular_pagamento()
    input("Pressione Enter para continuar...")
```

Essa função realiza o cadastro de um administrador. Ela solicita e valida os dados: nome, idade (entre 1 e 119), salário, setor (entre opções válidas como financeiro, recursos humanos, logística ou atendimento) e horas trabalhadas (até 500). Após reunir os dados, cria um objeto `Administrativo`, registra as horas, adiciona à lista de administradores cadastrados, exibe suas informações e calcula o pagamento. Por fim, aguarda o usuário pressionar Enter para continuar.

```python
def listar(lista, titulo):
    excluir()
    print(f"=== {titulo} Cadastrados ===")
    if lista:
        for i, pessoa in enumerate(lista, 1):
            print(f"\n{i}.")
            try:
                pessoa.exibir_informacoes()
            except AttributeError:
                print(f"{pessoa.nome} ({pessoa.idade} anos)")
                print("Obs: método exibir_informacoes() não implementado.")
    else:
        print("Nenhum cadastro encontrado.")
    input("\nPressione Enter para continuar...")
```

Essa função `listar` exibe os cadastros de uma lista com um título personalizado. Ela limpa a tela, mostra os itens numerados e tenta chamar o método `exibir_informacoes()` de cada objeto. Se o método não existir, mostra apenas o nome e idade, com uma observação sobre a ausência do método. Se a lista estiver vazia, informa que não há cadastros e aguarda o usuário pressionar Enter para continuar.

```python
while True:
    excluir()
    print("\n=== Hyspytol ===")
    print("1 - Cadastrar")
    print("2 - Listar")
    print("3 - Pagamentos")
    print("4 - Criar Sala (Tipos)")
    print("5 - Consultas")
    print("6 - Encerrar programa")
    try:
        op = int(input("Escolha uma opção (1-6): "))
    except ValueError:
        print("Opção inválida. Tente novamente.")
        input("Pressione Enter para continuar...")
        continue
```

```python
match op:
    case 1 :
        excluir()
        while True:
            excluir()
            print("\n=== Cadastrar ===")
            print("1 - Cadastrar Pacientes")
            print("2 - Cadastrar Médicos")
            print("3 -Cadastrar Enfermeiros")
            print("4 - Cadastrar Enfermeiro Chefe")
            print("5 - Cadastrar Funcionários")
            print("6 - Cadastrar Administradores")
            print("7 - voltar")
            try:
                op = int(input("Escolha uma opção (1-7): "))
            except ValueError:
                print("Opção inválida. Tente novamente.")
                input("Pressione Enter para continuar...")
                continue
```

```
            match op:
                case 1: cadastrar_paciente()
                case 2: cadastrar_medico()
                case 3: cadastrar_enfermeiro()
                case 4: cadastrar_enfermeiro_chefe()
                case 5: cadastrar_funcionario()
                case 6: cadastrar_administrador()
                case 7: break
                case _:
                    print("Opção inválida.")
                    input("Pressione Enter para continuar...")

        case 2 :
            excluir()
            while True:
                excluir()
                print("\n=== Listagem ===")
                print("1 - Listagem Pacientes")
                print("2 - Listagem Médicos")
                print("3 - Listagem Enfermeiros")
                print("4 - Listagem Enfermeiro Chefe")
                print("5 - Listagem Funcionários")
                print("6 - Listagem Administradores")
                print("7 - voltar")
                try:
                    op = int(input("Escolha uma opção (1-7): "))
                except ValueError:
                    print("Opção inválida. Tente novamente.")
                    input("Pressione Enter para continuar...")
                    continue
```

```
                match op:
                    case 1: listar(pacientes_cadastrados, "Pacientes")
                    case 2: listar(medicos_cadastrados, "Médicos")
                    case 3: listar(enfermeiros_cadastrados, "Enfermeiros")
                    case 4: listar(chefes_cadastrados, "Enfermeiros Chefes")
                    case 5: listar(funcionarios_cadastrados, "Funcionários")
                    case 6: listar(administradores_cadastrados, "Administradores")
                    case 7: break
                    case _:
                        print("Opção inválida.")
                        input("Pressione Enter para continuar...")
```

```
case 3 :
    excluir()
    while True:
        excluir()
        print("\n=== Pagamentos ===")
        print("1 - Pagamentos Funcionários")
        print("2 - Pagamentos Médicos")
        print("3 - Pagamentos Enfermeiros")
        print("4 - Pagamento Enfermeiro Chefe")
        print("5 - Pagamento Administradores")
        print("6 - Alterar Pagamentos")
        print("7 - voltar")
        try:
            op = int(input("Escolha uma opção (1-7): "))
        except ValueError:
            print("Opção inválida. Tente novamente.")
            input("Pressione Enter para continuar...")
            continue
```

```
        match op:
            case 1: pagamento(funcionarios_cadastrados, "Funcionários")
            case 2: calcular_pagamento(medicos_cadastrados, "Médicos")
            case 3: calcular_pagamento(enfermeiros_cadastrados, "Enfermeiros")
            case 4: calcular_pagamento(chefes_cadastrados, "Enfermeiros Chefes")
            case 5: calcular_pagamento(administradores_cadastrados, "Administradores")
            case 6:
                excluir()
                print("\n=== Alterar Pagamentos ===")
                print("1 - Funcionários")
                print("2 - Médicos")
                print("3 - Enfermeiros")
                print("4 - Enfermeiros Chefes")
                print("5 - Administradores")
                print("6 - Voltar")
                try:
                    sub_op = int(input("Escolha uma opção (1-6): "))
                except ValueError:
                    print("Opção inválida.")
                    input("Pressione Enter para continuar...")
                    continue
```

```
                case 7: break
                case _:
                    print("Opção inválida.")
                    input("Pressione Enter para continuar...")
        case 4 :
            excluir()
            while True:
                excluir()
                print("\n=== Salas ===")
                print("1 - Ocupar Sala de Consulta")
                print("2 - Ocupar Sala de Cirurgia")
                print("3 - Desocupar Sala de Consulta")
                print("4 - Desocupar Sala de Cirugia")
                print("5 - voltar")
                try:
                    op = int(input("Escolha uma opção (1-5): "))
                except ValueError:
                    print("Opção inválida. Tente novamente.")
                    input("Pressione Enter para continuar...")
                    continue

                match op:
                    case 1:
                        sala_consulta_ocupada = None
                        sala_cirurgia_ocupada = None
                        excluir()
                        if not medicos_cadastrados or len(pacientes_cadastrados) < 1:
                            print("Cadastre pelo menos 1 médico e 1 paciente.")
                            input("Pressione Enter para continuar...")
                            continue
```

```
                case 4:  # Desocupar Sala de Cirurgia
                    excluir()
                    if not medicos_cadastrados or not pacientes_cadastrados:
                        print("Cadastre pelo menos 1 médico e 1 paciente.")
                        input("Pressione Enter para continuar...")
                        continue

                    if sala_cirurgia_ocupada:
                        sala_cirurgia_ocupada._equipamentos.clear()
                        print("Sala de cirurgia desocupada com sucesso.")
                        sala_cirurgia_ocupada.exibir_informacoes()
                    else:
                        print("Nenhuma sala de cirurgia está ocupada.")
                    input("Pressione Enter para continuar...")
                case 5: break
                case _:
                    print("Opção inválida.")
                    input("Pressione Enter para continuar...")
        case 5:
            excluir()
            registrar_consulta()
            if sala_consulta_ocupada:
                paciente.adicionar_historico(f"Consulta com Dr. {medico.nome} em sala {sala_consulta_ocupada.numero}")
            else:
                print("Nenhuma sala de consulta está ocupada.")
        case 6:
            excluir()
            print("Sistema encerrado.")
            break
        case _:
            print("Opção inválida.")
            input("Pressione Enter para continuar...")
```

```
            medico = medicos_cadastrados[0]
            sala_consulta_ocupada = SalaConsulta(101, 1, medico)
            for paciente in pacientes_cadastrados[:1]:
                sala_consulta_ocupada.agendar_consulta(paciente)
            sala_consulta_ocupada.exibir_informacoes()
            input("Pressione Enter para continuar...")
        case 2:
            excluir()
            if not medicos_cadastrados or not pacientes_cadastrados:
                print("Cadastre pelo menos 1 médico e 1 paciente.")
                input("Pressione Enter para continuar...")
                continue
            medico = medicos_cadastrados[0]
            sala_cirurgia_ocupada = SalaCirurgia(101, 1, medico)
            for paciente in pacientes_cadastrados[:1]:
                sala_cirurgia_ocupada.agendar_consulta(paciente)
            sala_cirurgia_ocupada.exibir_informacoes()
            input("Pressione Enter para continuar...")
        case 3:  # Desocupar Sala de Consulta
            excluir()
            if not medicos_cadastrados or len(pacientes_cadastrados) < 1:
                print("Cadastre pelo menos 1 médico e 1 paciente.")
                input("Pressione Enter para continuar...")
                continue

            if sala_consulta_ocupada:
                sala_consulta_ocupada._pacientes_agendados.clear()
                print("Sala de consulta desocupada com sucesso.")
                sala_consulta_ocupada.exibir_informacoes()
            else:
                print("Nenhuma sala de consulta está ocupada.")
            input("Pressione Enter para continuar...")
```

Esse código é um **menu interativo do sistema hospitalar**, chamado *Hyspytol*. Ele permite cadastrar, listar e gerenciar diferentes tipos de pessoas (como pacientes, médicos e funcionários), controlar pagamentos, criar e liberar salas (de consulta ou cirurgia) e registrar consultas.

Em resumo, o programa organiza as operações principais de um hospital em um **menu baseado em opções numéricas**, onde o usuário escolhe o que quer fazer e o sistema executa a função correspondente.

Conclusão

A realização deste projeto representou uma experiência enriquecedora tanto no âmbito técnico quanto no trabalho em equipe. Durante o desenvolvimento, Kaique e Kauã atuaram de forma conjunta, demonstrando empenho, colaboração e comprometimento para alcançar todos os objetivos propostos. Apesar das dificuldades encontradas ao longo do processo, como a aplicação correta dos conceitos de herança, polimorfismo e classes abstratas, conseguimos superar cada desafio por meio de pesquisa, testes e aprimoramento contínuo do código. O resultado foi um sistema funcional e completo de gestão hospitalar, que não apenas cumpre todos os requisitos solicitados no enunciado, mas também incorpora elementos inspirados em sistemas reais utilizados em hospitais, tornando o projeto mais autêntico e aplicável a situações concretas. Assim, concluímos este trabalho com a sensação de dever cumprido e com a certeza de que os conhecimentos adquiridos serão fundamentais para o nosso desenvolvimento profissional e acadêmico nas próximas etapas do curso.