Programação de Sistemas de Informação

Projeto Final Módulo 3

CURSO TÉCNICO DE GESTÃO E PROGRAMAÇÃO DE SISTEMAS INFORTMATICOS

Professor:

Nome dos Alunos: Henrique França, Tainara Santos

Nº Aluno:. L2462, L2478

2510/2025

O Relatório encontra-se em condições para ser apresentado

_____

Ciclo de Formação 2023/2026

Ano Letivo 2025 / 2026

Índice

Introdução

O presente projeto foi desenvolvido no âmbito da disciplina de Programação e Sistemas Informáticos, com o objetivo de criar um sistema de gestão hospitalar em Python. O sistema permite o cadastro e gerenciamento de pacientes, médicos, enfermeiros, administrativos e salas de atendimento, incluindo agendamento de consultas e controle de equipamentos. O desenvolvimento aplica conceitos fundamentais de programação orientada a objetos, como herança simples e múltipla, polimorfismo e classes abstratas, além de utilizar estruturas de dados como listas e dicionários para organizar as informações.

Seguimos uma estrutura modular, com cada classe implementada em um módulo separado, resultando em um total de 11 ficheiros, no nosso trabalho tem 6 etapas. Essa organização facilita a manutenção e a compreensão do código, além de permitir a reutilização e a extensão das funcionalidades de forma clara e eficiente.

Processo

1º Etapa

Primeiramente, começamos por criar as classes abstratas. A primeira delas é a Pessoa.py, que constitui a classe base de toda a estrutura do projeto. Esta classe é herdada por várias outras classes, como Paciente e Funcionário, servindo como superclasse.

O principal objetivo da classe Pessoa é evitar a repetição de código, garantindo que ao criar um paciente ou um médico, os atributos comuns, como nome e idade, estejam sempre presentes.

Por ser uma classe abstrata, Pessoa obriga as classes que dela herdam a implementar métodos específicos, como `exibir_informacoes()`. Isso garante consistência e padronização na forma como as informações das pessoas são apresentadas no sistema, além de permitir o uso de polimorfismo, onde diferentes tipos de pessoas podem ser manipulados de maneira uniforme.

No contexto do projeto, a classe Pessoa é essencial, pois qualquer operação que envolva dados de pessoas (como registrar pacientes, listar médicos ou funcionários) depende dos atributos e métodos definidos nesta classe. Por exemplo, ao listar todos os pacientes do hospital, o sistema acessa `nome` e `idade`, atributos que já vêm padronizados pela classe Pessoa, independentemente de serem pacientes, médicos ou enfermeiros.

Dessa forma, a classe Pessoa fornece estrutura, consistência e reutilização de código, facilitando a manutenção do sistema e a adição de novas funcionalidades futuras.

```python
from abc import ABC, abstractmethod

class Pessoa(ABC):
    def __init__(self, nome, idade):
        self._nome = nome
        self._idade = idade

    @property
    def nome(self):
        pass

    @nome.setter
    def nome(self):
        pass

    @property
    def idade(self):
        pass


    @idade.setter
    def idade(self):
        pass

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

A classe abstrata Sala foi criada para servir como base para todas as salas do hospital, como salas de consulta e salas de cirurgia. Ela define atributos comuns a qualquer sala, como o número da sala e a sua capacidade, permitindo que todas as salas compartilhem essas características sem precisar repetir código. Por ser abstrata, a classe Sala obriga que todas as classes que dela herdam implementam o método `detalhes_Sala()`, garantindo que cada tipo de sala possa exibir suas informações específicas de forma padronizada.

A utilização desta classe oferece várias vantagens, promove a reutilização de código, já que os atributos número e capacidade são definidos apenas uma vez, permite polimorfismo, possibilitando tratar diferentes tipos de salas de forma uniforme ao chamar o método `detalhes_Sala()` e organiza o sistema de forma escalável, de modo que novos tipos de salas podem ser criados apenas herdando a classe e implementando os métodos necessários, sem alterar o restante do código.

No contexto do projeto, a classe Sala é fundamental para o gerenciamento das instalações do hospital, pois possibilita registrar e exibir informações de todas as salas, controlar equipamentos nas salas de cirurgia e associar médicos responsáveis às salas de consulta. Dessa forma, garante padronização, flexibilidade e facilita a manutenção e expansão futura do sistema.

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


    @property
    def numero(self):
        pass

    @numero.setter
    def numero(self):
        pass


    @property
    def capacidade(self):
        pass

    @capacidade.setter
    def capacidade(self):
        pass

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

2º Etapa

A classe Funcionario é uma classe que herda da classe abstrata Pessoa. O seu principal objetivo é representar qualquer funcionário do hospital, servindo como base para subclasses mais específicas, como Médico, Enfermeiro e Administrativo. Ela implementa de forma completa os métodos e propriedades abstratas herdadas da classe Pessoa, nomeadamente os atributos `nome` e `idade`, garantindo que cada funcionário tenha esses dados devidamente validados e protegidos.

Além dos atributos herdados, a classe adiciona dois novos, `cargo`, que identifica o papel do funcionário dentro do hospital, e `salario`, que representa a sua remuneração. O atributo `salario` é protegido (_salario) e possui métodos getter e setter que aplicam validações, garantindo que o valor inserido seja sempre positivo. Já o atributo `cargo` é público, pois não necessita de validações complexas.

A classe Funcionario também possui métodos que permitem gerir e manipular os dados dos funcionários. O método `mostrar_informacoes()` exibe dados básicos, como nome, cargo e salário, enquanto o método `exibir_informacoes()` — que implementa o método abstrato definido em Pessoa — mostra todas as informações completas do funcionário. Além disso, existe o método `aplicar_aumento(porcentagem)`, que permite atualizar o salário com base em uma percentagem de aumento, assegurando que apenas valores positivos sejam aceites.

Do ponto de vista de programação orientada a objetos, esta classe demonstra de forma prática o encapsulamento, através do uso de atributos protegidos e propriedades (`@property`), e a herança, ao estender a classe abstrata Pessoa. Através dela, o projeto consegue garantir uma estrutura consistente entre todos os funcionários, promovendo a reutilização de código e evitando redundâncias.

Em resumo, a classe Funcionario é essencial para a organização do sistema, pois fornece a base para todos os profissionais do hospital, unificando a forma como são tratados atributos e comportamentos comuns, e servindo como ponto de partida para as classes especializadas que a estendem.

```python
from Pessoa import Pessoa

class Funcionario(Pessoa):
    def __init__(self, nome, idade, cargo, salario):
        self._nome = nome
        self._idade = idade
        self.cargo = cargo
        self._salario = salario

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

    @nome.setter
    def nome(self, valor):
        if isinstance(valor, str) and len(valor.strip()) > 0:
            self._nome = valor
        else:
            print("Erro: Nome inválido")

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

    @idade.setter
    def idade(self, valor):
        if isinstance(valor, int) and valor > 0:
            self._idade = valor
        else:
            print("Erro: Idade deve ser um número inteiro positivo")
```

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

    @salario.setter
    def salario(self, new_salario):
        if new_salario >0:
            self._salario = new_salario
        else:
            print("Salário não pode ser nulo ou negativo.")


    def mostrar_informacoes(self):
        print("Nome:", self.nome)
        print("Cargo:", self.cargo)
        print("Salário:", self._salario)

    def aplicar_aumento(self, porcentagem):
        if porcentagem >0:
            aumento = self.salario * (porcentagem/100)
            self._salario = self._salario + aumento
            print(f"Novo salário: {self.salario}")

        else:
            print("Pocentagem te que ser maior que zero.")


    def exibir_informacoes(self):
        print("Nome:", self.nome)
        print("Idade:", self.idade)
        print("Cargo:", self.cargo)
        print("Salário:", self._salario)
```

A classe Paciente representa uma das entidades centrais do sistema, sendo responsável pela gestão das informações dos pacientes do hospital. Esta classe herda da classe abstrata `Pessoa`, o que lhe permite reutilizar atributos e comportamentos comuns, como o `nome` e a `idade`. A utilização da herança aqui é fundamental para evitar duplicação de código e garantir consistência na estrutura dos dados entre todas as entidades derivadas de `Pessoa`.

Na classe Paciente, foram definidos os seguintes atributos principais, nome e idade, herdados de `Pessoa`, armazenam os dados básicos do paciente e são protegidos para garantir o encapsulamento n_utente, que representa o número de utente identificador único de cada paciente no sistema, historico, uma lista que regista o histórico médico

do paciente, permitindo o armazenamento de consultas, observações e anotações relevantes.

A classe implementa propriedades (`@property`) e métodos *setters* para cada um dos atributos, de forma a validar os dados inseridos pelo utilizador. Por exemplo, impede que um nome vazio ou uma idade inválida sejam atribuídos, promovendo assim a integridade dos dados no sistema.

Além dos métodos de acesso, a classe define duas funções importantes para o controlo do histórico do paciente, `add_registro(descricao)`, que adiciona uma nova entrada ao histórico, garantindo que o texto não seja vazio; `mostrar_historico()`, que apresenta todos os registos gravados, numerados e formatados para facilitar a leitura.

Por fim, a classe implementa o método abstrato `exibir_informacoes()`, herdado de `Pessoa`, que exibe de forma clara os dados essenciais do paciente — nome, idade e número de utente.

Paciente é um exemplo prático de herança e polimorfismo, uma vez que concretiza métodos definidos de forma genérica na classe base. O seu desenvolvimento demonstra o uso adequado de encapsulamento, validação e abstração, pilares da programação orientada a objetos.

Assim, a classe Paciente é essencial para o sistema de gestão hospitalar, pois fornece a estrutura necessária para armazenar, validar e apresentar as informações dos pacientes, garantindo uma base sólida e coerente para as restantes funcionalidades do projeto.

```python
from Pessoa import Pessoa
class Paciente(Pessoa):


    def __init__(self, nome, idade, nutente):
        self._nome = nome
        self._idade = idade
        self._nutente = nutente
        self.historico = []

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

    @nome.setter
    def nome(self, valor):
        if isinstance(valor, str) and len(valor.strip()) > 0:
            self._nome = valor
        else:
            print("Erro: Nome inválido")

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

    @idade.setter
    def idade(self, valor):
        if isinstance(valor, int) and valor > 0:
            self._idade = valor
        else:
            print("Erro: Idade deve ser um número inteiro positivo")
```

```python
    @property
    def nutente(self):
        return self._nutente

    @nutente.setter
    def nutente(self, valor):
        if len(valor)>0:
            self._nutente = valor

        else:
            print("Erro: Valor inválido")


    def add_registro(self, descricao):
        if len(descricao)>0:
            self.historico.append(descricao)
            print("Descrição registrada!")

        else:
            print("Erro: A descrição não pode estar vazia")


    def mostrar_historico(self):
        if len(self.historico)>0:
            print("Histórico do Paciente: ")
            for i in range(len(self.historico)):
                print(i +1, "-", self.historico[i])

        else:
            print("O histórico está vazio")



    def exibir_informacoes(self):
        print("Nome:", self.nome)
        print("Idade:", self.idade)
        print("Número de Utente:", self.nutente)
```

3º Etapa

A classe Medico herda da classe Funcionario, representando os médicos que trabalham no hospital. Através da herança, esta classe reutiliza todas as propriedades e validações definidas em `Funcionario`, acrescentando atributos e comportamentos específicos relacionados à função médica.

O construtor (`__init__`) inicializa os atributos herdados — nome, idade, cargo e salário — e acrescenta o atributo protegido especialidade, que indica a área de atuação do médico (por exemplo, cardiologia, pediatria, ortopedia, etc.). Além disso, é criada uma lista chamada pacientes, responsável por armazenar todos os pacientes atendidos por aquele médico.

A classe implementa propriedades para gerir a especialidade, com métodos *getter* e *setter* que garantem a integridade do dado, impedindo, por exemplo, que uma especialidade vazia seja atribuída.Entre os métodos adicionais definidos pela classe,
`add_paciente(paciente)` – adiciona um paciente à lista de pacientes do médico, registrando a associação entre ambos;
`listar_pacientes()` – exibe todos os pacientes atendidos pelo médico, enumerando-os de forma organizada;
`calcular_pagamento()` – calcula o pagamento total do médico com base no seu salário e em um valor fixo adicional para cada paciente atendido.

O método `exibir_informacoes()` foi sobrescrito para apresentar de forma detalhada os dados completos do médico, incluindo o nome, idade, especialidade, número de pacientes atendidos e o salário total (já com os bônus calculados).

A classe Medico exemplifica o uso de herança, encapsulamento e polimorfismo no projeto. Ela mostra como é possível reutilizar a estrutura e as validações de uma superclasse (`Funcionario`), estendendo-a com novas funcionalidades específicas. Além disso, demonstra a aplicação de lógica de negócio dentro dos métodos, como o cálculo automático de pagamento, tornando o sistema mais dinâmico e realista.

Em resumo, a classe Medico desempenha um papel central no sistema de gestão hospitalar, pois faz a ligação direta entre os funcionários e os pacientes. Ela garante a consistência dos dados, facilita o acompanhamento das atividades médicas e exemplifica boas práticas de programação orientada a objetos.

```python
from Funcionario import Funcionario

class Medico(Funcionario):

    def __init__(self, nome, idade, cargo, salario, especialidade):
        super().__init__(nome, idade, cargo, salario)
        self._especialidade = especialidade
        self.pacientes = []


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


    @especialidade.setter
    def especialidade(self, valor):
        if len(valor)>0:
            self._especialidade = valor
        else:
            print("Valor inválido.")

    def add_paciente(self, paciente):
        self.pacientes.append(paciente)
        print("Paciente Adcionado!")


    def listar_pacientes(self):
        if len(self.pacientes)>0:
            print("Pacientes atendidos: ")
            for i in range(len(self.pacientes)):
                print(i +1, "-", self.pacientes[i])

    def calcular_pagamento(self):
        valor_paciente = 75
        pg_total = self.salario + (len(self.pacientes) * valor_paciente)
        return pg_total

    def exibir_informacoes(self):
        print(f"Nome: {self.nome}")
        print(f"Idade:  {self.idade}")
        print(f"Especialidade: {self._especialidade}")
        print(f"Pacientes atendidos: {len(self.pacientes)}")
        print(f"Salário total: {self.calcular_pagamento()}")
```

A classe Enfermeiro herda a classe Funcionario, criada para representar os profissionais de enfermagem que trabalham no hospital. Tal como as outras classes derivadas, ela aproveita a estrutura e as validações fornecidas por `Funcionário`, adicionando atributos e comportamentos específicos da profissão de enfermagem.

O construtor da classe (`__init__`) utiliza o método da superclasse para inicializar os atributos comuns — nome, idade, cargo e salário — e acrescenta um novo atributo protegido, _turno, que identifica o período de trabalho do enfermeiro (por exemplo, “dia” ou “noite”). Além disso, a classe mantém uma lista chamada pacientes, que armazena todos os pacientes atualmente sob os cuidados desse profissional.

O atributo turno é controlado através de *getters* e *setters*, garantindo a integridade dos dados. O *setter* realiza uma validação simples, aceitando apenas os valores “dia” e “noite”. Caso seja informado um valor inválido, o programa exibe uma mensagem de erro. Entre os métodos adicionais

`add_paciente(paciente)` – adiciona um paciente à lista de cuidados do enfermeiro, registrando o acompanhamento individual;
`listar_pacientes()` – exibe todos os pacientes sob responsabilidade do enfermeiro, apresentando-os de forma organizada e numerada;
`calcular_pg()` – calcula o pagamento total do enfermeiro, somando um bônus de acordo com o turno de trabalho (bônus maior para o turno da noite, devido ao esforço adicional);
`exibir_informacoes()` – apresenta um resumo das informações principais do enfermeiro, incluindo nome, idade, turno, número de pacientes sob cuidado e o salário total com o adicional.

A classe Enfermeiro ilustra bem o uso dos princípios de herança e encapsulamento, herdando características de `Funcionário` e expandindo-as com novas regras e comportamentos. Também demonstra a aplicação prática da validação de dados e da reutilização de código dentro de um contexto realista de gestão hospitalar.

Em resumo, a classe Enfermeiro é essencial para o funcionamento do sistema, pois permite acompanhar os profissionais de enfermagem, o seu turno de trabalho e os pacientes sob seus cuidados, mantendo a coerência e a segurança das informações dentro do hospital.

```python
from Funcionario import Funcionario
class Enfermeiro(Funcionario):
    def __init__(self, nome, idade, cargo, salario, turno):
        Funcionario.__init__(self, nome, idade, cargo, salario)
        self._turno = turno
        self.pacientes = []


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

    @turno.setter
    def turno(self, valor):
        if valor == "dia" or valor == "noite":
            self._turno = valor
        else:
            print("Erro: o turno deve ser 'dia' ou 'noite'. ")


    def add_paciente(self, paciente):
        self.pacientes.append(paciente)
        print("Paciente adicionado aos cuidados do enfermeiro!")


    def listar_pacientes(self):
        if len(self.pacientes)>0:
            for i in range(len(self.pacientes)):
                print(i + 1, "-", self.pacientes[i])

        else:
            print("Nenhum paciente sob cuidado.")


    def calcular_pg(self):
        if self._turno == "noite":
            adicional = 100
        else :
            adicional = 20

        pg_total = self.salario + adicional
        return pg_total


    def exibir_informacoes(self):
        print(f"Nome: {self.nome}")
        print(f"Idade: {self.idade}")
        print(f"Turno: {self._turno}")
        print(f"Pacientes sob cuidado: {len(self.pacientes)}")
        print(f"Salário total: {self.calcular_pg()}")
```

A classe Administrativo representa os funcionários responsáveis pelas tarefas administrativas dentro do hospital, como secretariado, atendimento e gestão de documentos. Assim como as outras classes especializadas, Administrativo herda da classe base Funcionario, aproveitando toda a estrutura e validações já definidas para os atributos comuns — nome, idade, cargo e salário.

No construtor da classe (`__init__`), além de inicializar os dados herdados, é criado um novo atributo protegido chamado _setor, que indica o departamento ou área de atuação do funcionário (por exemplo, “receção” ou “contabilidade”). Outro atributo, hrs_t, é inicializado com o valor zero e serve para armazenar o total de horas trabalhadas pelo funcionário administrativo.

O atributo setor é gerido através de propriedades (*getters* e *setters*), garantindo que o valor informado não seja vazio. Essa abordagem assegura a integridade dos dados e previne erros comuns, como a ausência de um setor válido.

Além disso, a classe define métodos que representam as operações típicas desse tipo de funcionário:

`registrar_horas(horas)` – permite adicionar horas trabalhadas ao total acumulado. O método aceita apenas valores positivos e confirma o registo com uma mensagem apropriada.
`calcular_pagamento()` – calcula o salário total do funcionário, somando ao salário base um valor adicional proporcional às horas trabalhadas. O valor da hora é definido localmente (no código), mas poderia ser obtido futuramente de uma base de dados ou ficheiro de configuração.
`exibir_informacoes()` – apresenta de forma organizada todas as informações do funcionário administrativo, incluindo nome, idade, setor, total de horas e o valor final do salário com as horas contabilizadas.

A classe Administrativo demonstra de forma clara a reutilização de código através de herança, além de aplicar encapsulamento e validação de dados. Também ilustra a extensibilidade do sistema, uma vez que amplia o comportamento da classe base para atender a necessidades específicas de um tipo particular de funcionário.

Em suma, a classe Administrativo desempenha um papel fundamental no projeto, pois permite gerir de forma simples e eficaz os colaboradores responsáveis pelas funções

administrativas do hospital, assegurando consistência e precisão no controlo de horas e nos cálculos salariais.

```python
from Funcionario import Funcionario
class Administrativo(Funcionario):

    def __init__(self, nome, idade, cargo, salario, setor):
        super().__init__(nome, idade, cargo, salario)
        self._setor = setor
        self.hrs_t = 0



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

    @setor.setter
    def setor(self, valor):
        if len(valor) > 0:
            self._setor = valor
        else:
            print("Erro: o setor não pode estar vazio.")



    def registrar_horas(self, horas):
        if horas > 0:
            self.hrs_t = self.hrs_t + horas
            print("Horas registradas com sucesso!")

        else:

            print("Erro: as horas devem ser maiores que zero.")



    def calcular_pagamento(self):
        valor_por_hora = 8
        pagamento_total = self.salario + (self.hrs_t * valor_por_hora)
        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.hrs_t}")
        print(f"Salário total: {self.calcular_pagamento()}")
```

4º Etapa

A classe EnfermeiroChefe demonstra o uso de herança múltipla em Python. Essa classe combina as funcionalidades de duas outras classes já existentes, Enfermeiro e Administrativo, representando um profissional que acumula responsabilidades técnicas e de gestão dentro do hospital.

O construtor (`__init__`) inicializa explicitamente ambas as partes herdadas. Primeiramente, é chamado o construtor de Enfermeiro, que define atributos como turno e a lista de pacientes sob cuidado. Em seguida, é invocado o construtor de Administrativo, responsável por atributos como setor e horas trabalhadas. Essa inicialização dupla é necessária porque a herança múltipla pode gerar ambiguidade na ordem de execução dos construtores, e, ao fazê-lo manualmente, o código mantém-se claro e controlado.

Além dos atributos herdados, a classe EnfermeiroChefe introduz um novo atributo protegido, _bonus_chefia, que representa um valor adicional pago ao profissional pelo cargo de liderança. Esse atributo é gerido através de propriedades (*getters* e *setters*), garantindo que o bônus seja sempre positivo.

O método `calcular_pagamento()` é um exemplo claro de polimorfismo e composição de comportamentos herdados. Ele combina os cálculos de pagamento das duas classes base — Enfermeiro (que adiciona um valor fixo de acordo com o turno) e Administrativo (que soma o salário base às horas registradas multiplicadas por um valor fixo por hora) — e ainda adiciona o bônus de chefia, resultando em um cálculo salarial completo e adaptado ao papel híbrido do enfermeiro-chefe.

Já o método `exibir_informacoes()` fornece uma visão consolidada dos dados do profissional, mostrando atributos herdados de ambas as classes: nome, idade, turno, setor, quantidade de pacientes, bônus de chefia e o salário total calculado.

A implementação da classe EnfermeiroChefe demonstra de forma prática a herança múltipla, um conceito poderoso da programação orientada a objetos, que permite que uma classe derive de mais de uma superclasse. Embora essa técnica exija cuidado para evitar conflitos de métodos e atributos, neste projeto ela foi usada de forma simples e eficiente, ilustrando como é possível combinar diferentes responsabilidades em um único objeto.

Em resumo, EnfermeiroChefe é uma classe essencial para mostrar a integração entre diferentes papéis no hospital, além de destacar o uso correto de herança múltipla, encapsulamento e polimorfismo — conceitos fundamentais no paradigma orientado a objetos e amplamente explorados neste projeto.

```python
from Enfermeiro import Enfermeiro
from Administrativo import Administrativo

class EnfermeiroChefe(Enfermeiro, Administrativo):
    def __init__(self, nome, idade, salario_base, turno, setor, bonus_chefia):
        Enfermeiro.__init__(self, nome, idade, "Enfermeiro Chefe", salario_base, turno)
        Administrativo.__init__(self, nome, idade, "Enfermeiro Chefe", salario_base, setor)
        self._bonus_chefia = bonus_chefia



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

    @bonus_chefia.setter
    def bonus_chefia(self, valor):
        if valor > 0:
            self._bonus_chefia = valor
        else:
            print("Erro: o bônus de chefia deve ser positivo.")



    def calcular_pagamento(self):
        pagamento_enfermeiro = Enfermeiro.calcular_pg(self)
        pagamento_admin = Administrativo.calcular_pagamento(self)
        pagamento_total = pagamento_enfermeiro + pagamento_admin + self._bonus_chefia
        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"Pacientes sob cuidado: {len(self.pacientes)}")
        print(f"Bônus de chefia: {self._bonus_chefia}")
        print(f"Salário total: {self.calcular_pagamento()}")
```

5º Etapa

A classe SalaCirurgia é uma implementação concreta da classe abstrata Sala, representando as salas de cirurgia do hospital. Essa classe especializa o comportamento genérico definido na superclasse, adicionando funcionalidades específicas relacionadas ao controle de equipamentos cirúrgicos e à configuração da sala.

O construtor (`__init__`) recebe dois parâmetros principais — número e capacidade — e os armazena em atributos protegidos, permitindo que sejam validados e controlados através de propriedades (*getters* e *setters*). Além disso, é inicializada uma lista chamada equipamentos, que será usada para registrar os materiais e aparelhos disponíveis em cada sala de cirurgia.

As propriedades `numero` e `capacidade` foram implementadas com validações de integridade, assegurando que apenas valores inteiros e positivos possam ser atribuídos. Essa abordagem previne erros de entrada e mantém a consistência dos dados dentro do sistema.

A classe também define o método `adicionar_equipamento()`, responsável por inserir novos equipamentos na lista da sala. O método inclui uma verificação simples para garantir que o nome do equipamento não seja vazio, e exibe mensagens de sucesso ou erro conforme o resultado da operação. Essa funcionalidade contribui para o gerenciamento dinâmico dos recursos hospitalares, permitindo registrar e acompanhar os instrumentos utilizados nas cirurgias.

O método `detalhes_Sala()` é a implementação concreta do método abstrato definido na classe Sala, e tem como função exibir as informações completas sobre a sala de cirurgia. Ele mostra o número da sala, sua capacidade e a lista de equipamentos disponíveis. Caso não haja nenhum equipamento cadastrado, o método apresenta uma mensagem informativa ao usuário.

Ela demonstra de forma prática a herança de uma classe abstrata, aplicando o conceito de polimorfismo para personalizar o comportamento do método

abstrato `detalhes_Sala`.

Serve como um componente essencial do sistema hospitalar, já que o gerenciamento de salas e equipamentos cirúrgicos é uma parte fundamental da administração hospitalar.

Em resumo, SalaCirurgia representa uma aplicação clara dos princípios de encapsulamento, herança e abstração, além de introduzir uma camada de controle sobre os recursos físicos do hospital. A sua implementação reforça a importância do uso de classes abstratas como modelo base para diferentes tipos de salas dentro da aplicação.

```python
from Sala import Sala
class SalaCirurgia(Sala):

    def __init__(self, numero, capacidade):
        self._numero = numero
        self._capacidade = capacidade
        self.equipamentos = []

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

    @numero.setter
    def numero(self, valor):
        if isinstance(valor, int) and valor > 0:
            self._numero = valor
        else:
            print("Erro: 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:
            print("Erro: Capacidade deve ser um número inteiro positivo")


    def adicionar_equipamento(self, equipamento):
        if len(equipamento) > 0:
            self.equipamentos.append(equipamento)
            print(f"Equipamento '{equipamento}' adicionado com sucesso!")
        else:
            print("Erro: nome de equipamento inválido.")


    def detalhes_Sala(self):
        print(f"Número da sala: {self.numero}")
        print(f"Capacidade: {self.capacidade}")
        print("Equipamentos disponíveis:")

        if len(self.equipamentos) == 0:
            print("Nenhum equipamento cadastrado.")
        else:
            for i in range(len(self.equipamentos)):
                print(i + 1, "-", self.equipamentos[i])
```

A classe SalaConsulta é uma implementação concreta da classe abstrata Sala, representando as salas utilizadas para consultas médicas dentro do hospital. Essa classe especializa o comportamento genérico definido na superclasse, adicionando funcionalidades relacionadas à gestão de pacientes e ao médico responsável pela sala.

O construtor (`__init__`) recebe três parâmetros principais: número, capacidade e médico responsável, sendo que este último deve ser um objeto da classe Medico. A sala também mantém uma lista chamada pacientes, que armazena os pacientes agendados para consultas naquela sala.

As propriedades `numero` e `capacidade` incluem validações de integridade, garantindo que apenas valores inteiros e positivos possam ser atribuídos. A propriedade `medico_responsavel` assegura que o responsável pela sala seja realmente um objeto da classe Medico, prevenindo inconsistências no gerenciamento do sistema.

A classe oferece o método `agendar_consulta()`, que adiciona pacientes à lista de agendamentos da sala. Esse método inclui validação básica para garantir que o paciente informado seja válido, e fornece mensagens de confirmação ou erro conforme a operação.

O método `detalhes_Sala()` implementa o método abstrato da superclasse, exibindo informações completas da sala de consulta: número, capacidade e nome do médico responsável. Ele fornece uma visão clara e rápida sobre os recursos e responsáveis da sala.

Ela demonstra a aplicação prática de herança e polimorfismo, implementando o método abstrato `detalhes_Sala` com comportamento específico.
Serve como componente essencial para o agendamento de consultas médicas, garantindo que cada sala tenha um médico responsável e uma lista organizada de pacientes.

Em resumo, SalaConsulta aplica princípios de encapsulamento, validação de dados e reutilização de código. Sua implementação reforça a confiabilidade do sistema ao manter o controle sobre médicos e pacientes de forma estruturada e segura.

```python
from Sala import Sala
from Medico import Medico

class SalaConsulta(Sala):

    def __init__(self, numero, capacidade, medico_responsavel):
        self._numero = numero
        self._capacidade = capacidade
        self._medico_responsavel = medico_responsavel
        self.pacientes = []

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

    @numero.setter
    def numero(self, valor):
        if isinstance(valor, int) and valor > 0:
            self._numero = valor
        else:
            print("Erro: 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:
            print("Erro: Capacidade deve ser um número inteiro positivo")


    @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:
            print("Erro: o responsável deve ser um objeto da classe Medico.")



    def agendar_consulta(self, paciente):

        if len(self.pacientes) >0:
            self.pacientes.append(paciente)
            print(f"Paciente {paciente} adicionado à lista de consultas.")
        else:
            print("Erro: Introduza um nome válido.")


    def detalhes_Sala(self):
        print(f"Número da sala: {self.numero}")
        print(f"Capacidade: {self.capacidade}")
        print(f"Médico responsável: {self._medico_responsavel.nome}")
```

6º Etapa

O ficheiro `menu.py` é a interface principal do sistema, responsável por interagir com o usuário e permitir a criação, gerenciamento e visualização de todas as entidades do hospital. Ele implementa um menu de linha de comando (CLI), que organiza as funcionalidades do sistema de forma intuitiva e segura.

O arquivo está estruturado em duas funções principais, `mostrar_menu()`
Esta função apresenta um cabeçalho do sistema, exibindo o nome do hospital e uma mensagem de boas-vindas. Ela pausa a execução até que o usuário pressione [Enter], garantindo que o usuário tenha tempo para visualizar as informações antes de prosseguir para o menu principal.

```python
from Paciente import Paciente
from Medico import Medico
from Enfermeiro import Enfermeiro
from Administrativo import Administrativo
from EnfermeiroChefe import EnfermeiroChefe
from SalaConsulta import SalaConsulta
from SalaCirurgia import SalaCirurgia

def mostrar_menu():
    print("======================")
    print("        HOSPITAL       ")
    print("======================")
    print("Olá, seja bem-vindo(a) ao menu do Hospital")
    input("Pressione [Enter] para continuar...")

def menu_principal():
    pacientes = []
    medicos = []
    enfermeiros = []
    administrativos = []
    enfermeiros_chefes = []
    salas_consulta = []
    salas_cirurgia = []
    consultas = []

    while True:
        print("\n===== MENU PRINCIPAL =====")
        print("1. Criar Paciente")
        print("2. Criar Médico")
        print("3. Criar Enfermeiro")
        print("4. Criar Administrativo")
        print("5. Criar Enfermeiro Chefe")
        print("6. Criar Sala de Consulta")
        print("7. Criar Sala de Cirurgia")
        print("8. Agendar Consulta")
        print("9. Adicionar Equipamento a Sala de Cirurgia")
        print("10. Listar Todos os Registros")
        print("0. Sair")

        op = input("Escolha uma opção: ")
```

`menu_principal()`

Esta função contém o loop central do sistema, que gerencia todas as operações de criação, modificação e listagem de objetos.

Inicialmente, são criadas listas para armazenar os objetos de cada classe (`pacientes`, `medicos`, `enfermeiros`, `administrativos`, `enfermeiros_chefes`, `salas_consulta`, `salas_cirurgia` e `consultas`). Essa abordagem garante que todos os dados fiquem em memória durante a execução, permitindo fácil acesso e manipulação das entidades.

O menu apresenta 10 opções principais e uma opção de saída, permitindo ao usuário:

Criar pessoas e funcionários, incluindo especializações como médicos, enfermeiros e administrativos.

Criar salas de consulta e cirurgia, associando médicos responsáveis quando necessário.

Agendar consultas, validando a existência de médicos e pacientes e registrando os eventos nas listas correspondentes.

Adicionar equipamentos às salas de cirurgia.

Listar todos os registros do sistema, utilizando os métodos `exibir_informacoes()` e `detalhes_Sala()` das respectivas classes, consolidando os dados para visualização completa.

Cada criação de objeto inclui validações de entrada (idade, salário, número de utente, turno, capacidade da sala, etc.), prevenindo a inserção de dados inválidos. Mensagens de erro ou confirmação são exibidas, fornecendo feedback claro ao usuário.

No caso do agendamento de consultas, os pacientes são associados diretamente aos médicos responsáveis, garantindo que a relação entre médicos e pacientes seja mantida corretamente, refletindo a realidade do hospital.

O menu utiliza estrutura `match/case` para organizar cada opção de forma legível e modular, permitindo fácil manutenção e expansão futura do sistema.

Em essência, `menu.py` serve como camada de interação entre o usuário e o sistema, integrando todas as classes implementadas e demonstrando na prática conceitos fundamentais de programação orientada a objetos, como:

Encapsulamento: através do uso de getters, setters e validações.

Herança e polimorfismo: as operações de exibição e manipulação de dados aproveitam métodos implementados nas classes base e especializadas.

Composição e relacionamentos entre objetos: médicos possuem pacientes, salas têm responsáveis e pacientes, salas de cirurgia mantêm equipamentos.

Essa organização torna o sistema robusto, intuitivo e extensível, permitindo que novas funcionalidades sejam adicionadas de forma estruturada sem comprometer a lógica existente.

```
match op:
    case "1":
        print("\n--- Criar Paciente ---")
        nome = input("Nome: ")

        while True:
            try:
                idade = int(input("Idade: "))
                break
            except ValueError:
                print("Erro: Idade inválida! Digite novamente.")

        numero_utente = input("Número de utente: ")
        paciente = Paciente(nome, idade, numero_utente)
        pacientes.append(paciente)
        print(f"Paciente {nome} criado com sucesso!")

    case "2":
        print("\n--- Criar Médico ---")
        nome = input("Nome: ")

        while True:
            try:
                idade = int(input("Idade: "))
                break
            except ValueError:
                print("Erro: Idade inválida! Digite novamente.")

        especialidade = input("Especialidade: ")

        while True:
            try:
                salario = float(input("Salário base: "))
                break
            except ValueError:
                print("Erro: Salário inválido! Digite novamente.")

        medico = Medico(nome, idade, "Médico", salario, especialidade)
        medicos.append(medico)
        print(f"Médico {nome} ({especialidade}) criado com sucesso!")

    case "3":
        print("\n--- Criar Enfermeiro ---")
        nome = input("Nome: ")

        while True:
            try:
                idade = int(input("Idade: "))
                break
            except ValueError:
                print("Erro: Idade inválida! Digite novamente.")

        turno = input("Turno (dia/noite): ")

        while True:
            try:
                salario = float(input("Salário base: "))
                break
            except ValueError:
                print("Erro: Salário inválido! Digite novamente.")

        enfermeiro = Enfermeiro(nome, idade, "Enfermeiro", salario, turno)
        enfermeiros.append(enfermeiro)
        print(f"Enfermeiro {nome} criado com sucesso!")
```

```python
    case "4":
        print("\n--- Criar Administrativo ---")
        nome = input("Nome: ")

        while True:
            try:
                idade = int(input("Idade: "))
                break
            except ValueError:
                print("Erro: Idade inválida! Digite novamente.")

        setor = input("Setor: ")

        while True:
            try:
                salario = float(input("Salário base: "))
                break
            except ValueError:
                print("Erro: Salário inválido! Digite novamente.")

        admin = Administrativo(nome, idade, "Administrativo", salario, setor)
        administrativos.append(admin)
        print(f"Funcionário administrativo {nome} criado com sucesso!")

    case "5":
        print("\n--- Criar Enfermeiro Chefe ---")
        nome = input("Nome: ")

        while True:
            try:
                idade = int(input("Idade: "))
                break
            except ValueError:
                print("Erro: Idade inválida! Digite novamente.")

        turno = input("Turno (dia/noite): ")
        setor = input("Setor: ")

        while True:
            try:
                salario = float(input("Salário base: "))
                bonus = float(input("Bônus de chefia: "))
                break
            except ValueError:
                print("Erro: Salário ou bônus inválido! Digite novamente.")

        chefe = EnfermeiroChefe(nome, idade, salario, turno, setor, bonus)
        enfermeiros_chefes.append(chefe)
        print(f"Enfermeiro Chefe {nome} criado com sucesso!")

    case "6":
        print("\n--- Criar Sala de Consulta ---")
        while True:
            try:
                numero = int(input("Número da sala: "))
                capacidade = int(input("Capacidade: "))
                break
            except ValueError:
                print("Erro: Número ou capacidade inválido! Digite novamente.")

        medico_nome = input("Nome do médico responsável: ")
        medico_resp = None
        for m in medicos:
            if m.nome == medico_nome:
                medico_resp = m
                break

        if medico_resp:
            sala = SalaConsulta(numero, capacidade, medico_resp)
            salas_consulta.append(sala)
            print(f"Sala de consulta {numero} criada com sucesso!")
        else:
            print("Erro: Médico não encontrado.")
```

```
case "7":
    print("\n--- Criar Sala de Cirurgia ---")
    while True:
        try:
            numero = int(input("Número da sala: "))
            capacidade = int(input("Capacidade: "))
            break
        except ValueError:
            print("Erro: Número ou capacidade inválido! Digite novamente.")

    sala = SalaCirurgia(numero, capacidade)
    salas_cirurgia.append(sala)
    print(f"Sala de cirurgia {numero} criada com sucesso!")

case "8":
    print("\n--- Agendar Consulta ---")
    nome_medico = input("Nome do Médico: ")
    nome_paciente = input("Nome do Paciente: ")


    medico_encontrado = None
    for m in medicos:
        if m.nome == nome_medico:
            medico_encontrado = m
            break

    if not medico_encontrado:
        print(f"Erro: Médico '{nome_medico}' não encontrado.")
        continue

    paciente_encontrado = None
    for p in pacientes:
        if p.nome == nome_paciente:
            paciente_encontrado = p
            break

    if not paciente_encontrado:
        print(f"Erro: Paciente '{nome_paciente}' não encontrado.")
        continue


    data = input("Data (DD/MM/AAAA): ")
    tipo = input("Tipo (rotina/emergência): ")

    consultas.append({
        "medico": medico_encontrado,
        "paciente": paciente_encontrado,
        "data": data,
        "tipo": tipo
    })


    medico_encontrado.add_paciente(paciente_encontrado)
    print("Consulta agendada com sucesso!")

case "9":
    print("\n--- Adicionar Equipamento ---")
    while True:
        try:
            numero = int(input("Número da sala de cirurgia: "))
            break
        except ValueError:
            print("Erro: Número inválido! Digite novamente.")

    equipamento = input("Nome do equipamento: ")
    for sala in salas_cirurgia:
        if sala.numero == numero:
            sala.adicionar_equipamento(equipamento)
            break
    else:
        print("Sala não encontrada.")
```

```
            case "10":
                print("\n--- Listagem Geral ---")
                print("\nMédicos:")
                for m in medicos:
                    m.exibir_informacoes()

                print("\nEnfermeiros:")
                for e in enfermeiros:
                    e.exibir_informacoes()

                print("\nAdministrativos:")
                for a in administrativos:
                    a.exibir_informacoes()

                print("\nEnfermeiros Chefes:")
                for c in enfermeiros_chefes:
                    c.exibir_informacoes()

                print("\nPacientes:")
                for p in pacientes:
                    print(f"Nome: {p.nome}, Número de utente: {p.nutente}, Idade: {p.idade}")

                print("\nSalas de Consulta:")
                for s in salas_consulta:
                    s.detalhes_Sala()

                print("\nSalas de Cirurgia:")
                for s in salas_cirurgia:
                    s.detalhes_Sala()

                print("\nConsultas agendadas:")
                for c in consultas:
                    print(f"Médico: {c['medico'].nome}, Paciente: {c['paciente'].nome}, Data: {c['data']}, Tipo: {c['tipo']}")

            case "0":
                print("Saindo do sistema... Até logo!")
                break

            case _:
                print("Opção inválida! Tente novamente.")

if __name__ == "__main__":
    mostrar_menu()
    menu_principal()
```

Conclusão

O desenvolvimento deste projeto permitiu consolidar e aplicar os conhecimentos adquiridos na disciplina de Programação e Sistemas Informáticos, especialmente em Python e Programação Orientada a Objetos. Através da implementação de um sistema de gestão hospitalar, foi possível explorar conceitos fundamentais como herança simples e múltipla, polimorfismo, classes abstratas e encapsulamento, garantindo a organização e reutilização do código.

O sistema desenvolvido demonstra a importância de uma estrutura modular e bem organizada, onde cada classe desempenha um papel específico e as entidades interagem de forma consistente. A implementação do menu interativo possibilitou testar todas as funcionalidades, desde a criação de pacientes, médicos, enfermeiros e administrativos até o agendamento de consultas e gerenciamento de salas, proporcionando uma visão prática de como as relações entre objetos podem ser modeladas em um contexto real.