Escola Profissional
BENTO DE JESUS CARAÇA
DELEGAÇÃO DE LISBOA

RELATÓRIO

Programação de sistemas de informação

Projeto Final Módulo 11

Curso técnico de Gestão e programação de sistemas informáticos

Professor: Breno Sousa

Nome dos Alunos: Rafael Carvalho e Martim Sabido

Nº dos Alunos: L2472 e L2475

23/10/2025

O relatório encontra-se em condições para ser apresentado

_____

Ciclo de Formação 2023/2025

Ano Letivo 2025/2026

Escola Profissional
BENTO DE JESUS CARAÇA
DELEGAÇÃO DE LISBOA

Índice

Escola Profissional
BENTO DE JESUS CARAÇA
DELEGAÇÃO DE LISBOA

Introdução

O nosso trabalho consiste na implementação de um sistema hospitalar em Python, utilizando os princípios da Programação Orientada a Objectos (POO), como herança, polimorfismo e encapsulamento.

O sistema permite gerir pacientes, médicos, enfermeiros, administrativos e salas hospitalares, oferecendo funcionalidades como: Registo e listagem de pacientes e funcionários. Agendamento de consultas, com atualização automática do histórico do paciente. Registo de horários, turnos e salários dos funcionários. Gestão de salas de consulta e cirurgia, incluindo médicos responsáveis e equipamentos.

O objetivo é demonstrar a organização e manipulação de objetos inter-relacionados, garantindo a consistência dos dados e permitindo uma interatividade simples e intuitiva para o utilizador.

Conteúdo do código(Classe Pessoa)

```python
class Pessoa(ABC):
    def __init__(self, nome: str, idade: int):
        self.nome = nome
        self.idade = idade

    @property
    def nome(self) -> str:
        return self._nome

    @nome.setter
    def nome(self, v: str):
        if not v:
            raise ValueError("Nome vazio.")
        self._nome = v

    @property
    def idade(self) -> int:
        return self._idade

    @idade.setter
    def idade(self, v: int):
        if v <= 0:
            raise ValueError("Idade deve ser positiva.")
        self._idade = v

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

A classe Pessoa é definida como uma classe abstrata e serve como base para todos os tipos de pessoas do hospital.

O seu construtor recebe o nome e idade e usa os setters correspondentes para validar estes valores. O getter do nome permite aceder ao atributo de forma segura, e o setter garante que o nome não fique vazio, lançando um erro caso isso aconteça. De forma semelhante, o getter e o setter da idade permitem aceder e alterar o valor, garantindo que a idade seja sempre positiva. A classe também define um método abstrato para exibir informações, obrigando as subclasses a implementar este método de forma específica, o que demonstra polimorfismo.

Conteúdo do código(Classe Paciente)

```python
class Paciente(Pessoa):
    def __init__(self, nome: str, idade: int, numero_utente: str):
        super().__init__(nome, idade)
        self.numero_utente = numero_utente
        self.historico: List[str] = []

    @property
    def numero_utente(self) -> str:
        return self._numero_utente

    @numero_utente.setter
    def numero_utente(self, v: str):
        if not v:
            raise ValueError("Número inválido.")
        self._numero_utente = v

    def adicionar_registro(self, d: str):
        self.historico.append(d)

    def mostrar_historico(self):
        if not self.historico:
            print("Sem histórico.")
        else:
            for r in self.historico:
                print("-", r)

    def exibir_informacoes(self):
        print(f"Paciente: {self.nome}, Idade: {self.idade}, Nº Utente: {self.numero_utente}")
```

A classe Paciente herda de Pessoa e adiciona atributos próprios, nomeadamente o número de utente e um histórico, que é uma lista de registros médicos.

O construtor inicializa estes atributos, e o setter do número de utente garante que não seja vazio. O método para adicionar um registro ao histórico insere um novo evento na lista, enquanto o método para mostrar o histórico percorre a lista e imprime cada registro, ou informa se não houver registros.

O método para exibir informações sobrescreve o método abstrato da classe base, mostrando o nome, idade e número de utente do paciente.

Conteúdo do código(Classe Funcionário)

```python
class Funcionario(Pessoa):
    def __init__(self, nome: str, idade: int, cargo: str, salario):
        super().__init__(nome, idade)
        self.cargo = cargo
        self.salario = salario

    @property
    def salario(self) -> float:
        return self._salario

    @salario.setter
    def salario(self, v):
        try:
            v = float(v)
        except (ValueError, TypeError):
            raise ValueError("Salário deve ser um número.")
        if v < 0:
            raise ValueError("Salário inválido.")
        self._salario = v

    def mostrar_informacoes(self):
        print(f"{self.cargo}: {self.nome}, Salário: €{self.salario:.2f}")

    def aplicar_aumento(self, p: float):
        self.salario += self.salario * p / 100
```

A classe Funcionário herda da classe Pessoa e adiciona os atributos cargo e salário. O setter do salário converte o valor para float e verifica se é positivo, mostrando um erro caso contrário. O método para mostrar informações imprime o cargo, o nome e o salário formatado, e o método para aplicar aumento ajusta o salário com base num percentual, usando o setter para garantir validação.

As subclasses podem sobrescrever estes métodos para alterar o cálculo do pagamento ou a forma de apresentar informações, demonstrando polimorfismo.

Conteúdo do código(Classe Médico)

```python
class Medico(Funcionario):
    def __init__(self, nome: str, idade: int, salario_base: float, especialidade: str):
        super().__init__(nome, idade, "Médico", salario_base)
        self.especialidade = especialidade
        self.pacientes: List[Paciente] = []

    @property
    def especialidade(self) -> str:
        return self._especialidade

    @especialidade.setter
    def especialidade(self, v: str):
        if not v:
            raise ValueError("Especialidade vazia.")
        self._especialidade = v

    def adicionar_paciente(self, p: Paciente):
        if not isinstance(p, Paciente):
            raise ValueError("Deve ser um Paciente.")
        self.pacientes.append(p)

    def listar_pacientes(self):
        if not self.pacientes:
            print("Sem pacientes.")
        else:
            for p in self.pacientes:
                print("-", p.nome)

    def calcular_pagamento(self) -> float:
        return self.salario + len(self.pacientes) * 50

    def exibir_informacoes(self):
        print(f"Médico: {self.nome}, Esp: {self.especialidade}, Pacientes: {len(self.pacientes)}")
```

A classe Médico é uma especialização da classe Funcionário e acrescenta o atributo especialidade e uma lista de pacientes.

O método para adicionar pacientes verifica se o objeto é realmente um paciente e, em caso afirmativo, adiciona-o à lista. O método para listar pacientes percorre a lista e imprime os nomes, ou indica que não há pacientes se a lista estiver vazia. O cálculo de pagamento adiciona um bônus por paciente ao salário base, e o método para exibir informações mostra a especialidade e o número de pacientes. Esta classe exemplifica herança, composição e polimorfismo.

Conteúdo do código(Classe Enfermeiro)

```python
class Enfermeiro(Funcionario):
    def __init__(self, nome: str, idade: int, salario_base: float, turno: str):
        super().__init__(nome, idade, "Enfermeiro", salario_base)
        self.turno = turno
        self.pacientes: List[Paciente] = []

    @property
    def turno(self) -> str:
        return self._turno

    @turno.setter
    def turno(self, v: str):
        if v not in ("dia", "noite"):
            raise ValueError("Turno inválido.")
        self._turno = v

    def adicionar_paciente(self, p: Paciente):
        if not isinstance(p, Paciente):
            raise ValueError("Deve ser um Paciente.")
        self.pacientes.append(p)

    def listar_pacientes(self):
        if not self.pacientes:
            print("Sem pacientes.")
        else:
            for p in self.pacientes:
                print("-", p.nome)

    def calcular_pagamento(self) -> float:
        return self.salario + (200 if self.turno == "noite" else 100)

    def exibir_informacoes(self):
        print(f"Enfermeiro: {self.nome}, Turno: {self.turno}, Pacientes: {len(self.pacientes)}")
```

A classe Enfermeiro herda também da classe Funcionário e adiciona o atributo turno (dia ou noite) e a lista de pacientes que o enfermeiro atende. O setter do turno garante que apenas os valores "dia" ou "noite" sejam válidos. Os métodos de adicionar e listar pacientes funcionam de forma semelhante aos do médico.

O cálculo de pagamento adiciona um valor extra dependendo do turno e o método de exibir informações mostra o nome, turno e número de pacientes.

Conteúdo do código(Classe Administrativo)

```python
class Administrativo(Funcionario):
    def __init__(self, nome: str, idade: int, salario_base: float, setor: str):
        super().__init__(nome, idade, "Administrativo", salario_base)
        self.setor = setor
        self.horas = 0

    @property
    def setor(self) -> str:
        return self._setor

    @setor.setter
    def setor(self, v: str):
        if not v:
            raise ValueError("Setor inválido.")
        self._setor = v

    def registrar_horas(self, h: int):
        self.horas += h

    def calcular_pagamento(self) -> float:
        return self.salario + self.horas * 10

    def exibir_informacoes(self):
        print(f"Administrativo: {self.nome}, Setor: {self.setor}, Horas: {self.horas}")
```

A classe Administrativo herda da classe Funcionário e acrescenta os atributos setor e horas trabalhadas. O setter do setor garante que não fique vazio. O método para registrar horas incrementa o total de horas, e o cálculo de pagamento adiciona ao salário base um valor proporcional às horas trabalhadas. O método de exibir informações mostra o setor e as horas acumuladas.

Conteúdo do código(Classe EnfermeiroChefe)

```python
class EnfermeiroChefe(Funcionario):
    def __init__(self, nome: str, idade: int, salario_base: float, turno: str, setor: str, bonus_chefia: float):
        super().__init__(nome, idade, "Enfermeiro Chefe", salario_base)
        self.turno = turno
        self.setor = setor
        self.bonus_chefia = bonus_chefia
        self.pacientes: List[Paciente] = []
        self.horas = 0

    @property
    def turno(self) -> str:
        return self._turno

    @turno.setter
    def turno(self, v: str):
        if v not in ("dia", "noite"):
            raise ValueError("Turno inválido.")
        self._turno = v

    @property
    def setor(self) -> str:
        return self._setor

    @setor.setter
    def setor(self, v: str):
        if not v:
            raise ValueError("Setor inválido.")
        self._setor = v

    @property
    def bonus_chefia(self) -> float:
        return self._bonus_chefia
```

O EnfermeiroChefe é uma classe que combina funções de enfermeiro e administrativo através de herança múltipla. Ele herda atributos e métodos de ambos, como turno, lista de pacientes, setor, horas trabalhadas e salário.

Quando é criado, o construtor inicializa os atributos do Enfermeiro (turno, salário e pacientes) e também os do Administrativo (setor e horas), garantindo que todos os dados estejam corretos. Além disso, define o bonus_chefia, com validação para que nunca seja negativo.

Conteúdo do código(Classe EnfermeiroChefe pt2)

```python
@bonus_chefia.setter
def bonus_chefia(self, v: float):
    if v < 0:
        raise ValueError("Bônus inválido.")
    self._bonus_chefia = v

def adicionar_paciente(self, p: Paciente):
    if not isinstance(p, Paciente):
        raise ValueError("Deve ser um Paciente.")
    self.pacientes.append(p)

def registrar_horas(self, h: int):
    self.horas += h

def calcular_pagamento(self) -> float:
    return self.salario + (200 if self.turno == "noite" else 100) + self.horas * 10 + self.bonus_chefia

def exibir_informacoes(self):
    print(f"Enf. Chefe: {self.nome}, Turno: {self.turno}, Setor: {self.setor}, Pacientes: {len(self.pacientes)}")
```

O método de cálculo de pagamento combina o salário base, o adicional pelo turno, o pagamento por horas extras e o bônus de chefia, mostrando polimorfismo, já que altera o comportamento padrão das classes base. O método de exibir informações mostra o nome, turno, setor e número de pacientes, permitindo ver rapidamente o papel do enfermeiro chefe.

Conteúdo do código(Classe Sala)

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

    @property
    def numero(self) -> int:
        return self._numero

    @numero.setter
    def numero(self, v: int):
        if v <= 0:
            raise ValueError("Número inválido.")
        self._numero = v

    @property
    def capacidade(self) -> int:
        return self._capacidade

    @capacidade.setter
    def capacidade(self, v: int):
        if v <= 0:
            raise ValueError("Capacidade inválida.")
        self._capacidade = v

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

A classe Sala é abstrata e contém atributos comuns a todas as salas, como número e capacidade, com validações para impedir valores inválidos.

O método abstrato detalhar_sala obriga as subclasses a fornecerem uma descrição detalhada da sala.

Conteúdo do código(Classe SalaConsulta)

```python
class SalaConsulta(Sala):
    def __init__(self, numero: int, capacidade: int, medico_responsavel: Medico):
        super().__init__(numero, capacidade)
        self.medico_responsavel = medico_responsavel
        self.pacientes: List[Paciente] = []

    @property
    def medico_responsavel(self) -> Medico:
        return self._medico_responsavel

    @medico_responsavel.setter
    def medico_responsavel(self, v: Medico):
        if not isinstance(v, Medico):
            raise ValueError("Responsável inválido.")
        self._medico_responsavel = v

    def agendar_consulta(self, p: Paciente):
        if not isinstance(p, Paciente):
            raise ValueError("Deve ser um Paciente.")
        if len(self.pacientes) >= self.capacidade:
            print("Sala cheia.")
        else:
            self.pacientes.append(p)
            self.medico_responsavel.adicionar_paciente(p)
            print(f"Consulta marcada para {p.nome} na sala {self.numero}.")

    def detalhar_sala(self):
        print(f"Sala {self.numero} (Consulta) - Médico: {self.medico_responsavel.nome}, Capacidade: {self.capacidade}")
```

A classe SalaConsulta herda de Sala e adiciona o médico responsável e uma lista de pacientes.

O método de agendar consulta verifica se a sala ainda tem capacidade, adiciona o paciente à lista da sala e também à lista do médico responsável, e imprime uma mensagem a confirmar o agendamento. O método de detalhar sala imprime o número da sala, capacidade e nome do médico responsável.

Conteúdo do código(Classe SalaCirugia)

```python
class SalaCirurgia(Sala):
    def __init__(self, numero: int, capacidade: int):
        super().__init__(numero, capacidade)
        self.equipamentos: List[str] = []

    def adicionar_equipamento(self, e: str):
        self.equipamentos.append(e)

    def detalhar_sala(self):
        print(f"Sala {self.numero} (Cirúrgica) - Capacidade: {self.capacidade}")
        if self.equipamentos:
            print("Equipamentos:", ", ".join(self.equipamentos))
        else:
            print("Nenhum equipamento registrado.")
```

A classe SalaCirurgia herda da classe Sala e adiciona uma lista de equipamentos.

O método para adicionar equipamento insere novos equipamentos na lista, e o método de detalhar sala mostra o número, capacidade e todos os equipamentos disponíveis, ou indica que não há equipamentos se a lista estiver vazia.

Conteudo do Código (main)

```
from trabalhofinalpsi import *
```

Esta linha serve para importar tudo do ficheiro principal, sendo que "*" que a parte que importa tudo, envés de ter de escrever manualmente as classes que devem ser importadas.

```
def main():
    pacientes, medicos, funcionarios, salas = [], [], [], []

    # Funcionários iniciais
    m1 = Medico("João Rocha", 45, 3000, "Cardiologia")
    e1 = Enfermeiro("Maria Silva", 32, 1200, "noite")
    a1 = Administrativo("Carla Sousa", 40, 1000, "Recursos Humanos")
    ec1 = EnfermeiroChefe("Ana Costa", 38, 1500, "dia", "Urgência", 300)
    medicos.append(m1)
    funcionarios.extend([m1, e1, a1, ec1])

    # Salas
    sala1 = SalaConsulta(101, 2, m1)
    sala2 = SalaCirurgia(202, 1)
    sala2.adicionar_equipamento("Bisturi elétrico")
    sala2.adicionar_equipamento("Monitor cardíaco")
    salas.extend([sala1, sala2])
```

Nesta parte do código nós definimos o main, já atribuindo pessoas e toda a informação de cada pessoa para cada classe, exceto os pacientes que são adicionados manualmente durante a execução do código, por exemplo, com o médico é criado o médico com nome de João Rocha, 45 anos de idade, salário de 3000 euros por mês e da especialidade de cardiologia, e sequentemente com os outros trabalhadores, depois os médicos são adicionados a lista de médicos, que neste caso é apenas o Dr. Rocha, e todos os trabalhadores são adicionados a lista de trabalhadores. Essecialmente é a mesma coisa com as salas, atribuindo o código da sala, a capacidade e o médico da sala, que no caso da sala de consulta, é o Dr. Rocha. E na sala de cirurgia, é adicionado todo o equipamento necessário para realizar a cirurgia. No fim, ambas as salas são adicionadas a lista de salas que armazena todas as salas e objetos de cada sala.

Conteúdo do Código (main)

```
while True:
    print("\n--- SISTEMA HOSPITALAR ---")
    print("1. Adicionar paciente")
    print("2. Listar pacientes e histórico")
    print("3. Listar médicos")
    print("4. Agendar consulta")
    print("5. Listar salas")
    print("6. Atualizar funcionário")
    print("7. Adicionar registro ao histórico")
    print("0. Sair")
    opcao = input("Opção: ").strip()
```

Aqui é definido o menu interativo que será utilizado durante a execução do código, a linha "opcao = input("Opção: ").strip()" serve para o utilizador escrever a opção desejada para o código a executar, enquanto o strip remove espaços desnecessários durante o input.

```
if opcao == "1":
    nome = input("Nome do paciente: ").strip()
    try:
        idade = int(input("Idade: "))
    except ValueError:
        print("Idade inválida."); continue
    num = input("Nº Utente: ").strip()
    pacientes.append(Paciente(nome, idade, num))
    print(f"Paciente {nome} adicionado.")
```

Se a opção 1 for selecionada, é introduzido o nome do paciente, a idade e o nº de utente, se o numero introduzido para a idade não for inteiro dará erro e uma mensagem de erro será escrita, depois de tudo ser introduzido, uma instancia da classe paciente é criada e adicionada a lista de pacientes.

Conteúdo do código (main)

```
elif opcao == "2":
    if not pacientes: print("Nenhum paciente."); continue
    for p in pacientes:
        p.exibir_informacoes()
        p.mostrar_historico()
```

Se a segunda opção for selecionada, é verificado na lista de pacientes se existe algum paciente, se não houver, a mensagem de que não tem pacientes existente será mostrada, se existir pacientes, os pacientes e as suas informações e histórico.

```
elif opcao == "3":
    for m in medicos:
        m.exibir_informacoes()
        m.listar_pacientes()
```

Se a terceira opção for selecionada, a lista de médico é mostrada, neste caso não é necessário verificar a existência de médicos sendo que o médico já foi atribuído anteriormente, (Dr. Rocha).

Conteúdo do código (main)

```
elif opcao == "4":
    if not pacientes: print("Nenhum paciente."); continue
    salas_consulta = [s for s in salas if isinstance(s, SalaConsulta)]
    if not salas_consulta: print("Nenhuma sala de consulta."); continue
    for s in salas_consulta: s.detalhar_sala()
    try:
        num_sala = int(input("Número da sala: "))
    except ValueError: print("Número inválido."); continue
    sala = next((s for s in salas_consulta if s.numero == num_sala), None)
    if not sala: print("Sala não encontrada."); continue
    for i, p in enumerate(pacientes, 1): print(f"{i}. {p.nome}")
    try:
        escolha = int(input("Escolha paciente: "))
        paciente = pacientes[escolha-1]
    except: print("Escolha inválida."); continue
    sala.agendar_consulta(paciente)
    paciente.adicionar_registro(f"Consulta na sala {sala.numero} com {sala.medico_responsavel.nome}")
```

Se a quarta opção for selecionada, primeiro é verificado se algum paciente existe, se não existir nenhum paciente, a mensagem na segunda linha será apresentada, depois, é verificado a existência da sala, se não existir a sala, a mensagem adequada será apresentada, se estiver presente, os detalhes da sala serão mostrados, de seguinte será pedido para ser introduzido a informação da sala, como o número da sala (se o número não corresponder a nenhuma sala ou o valor não ser inteiro, a mensagem de erro será mostrado), depois será verificado a existência da sala de consulta, e verifica se corresponde ao número de sala introduzido, se nenhuma sala for encontrada com esse número, a mensagem adequada será mostrada, de seguinte uma lista de pacientes será apresentada e o utilizador escolherá o paciente à sua escolha através do número de utente, se o número de utente não corresponder a ninguém, a mensagem adequada será apresentada, depois o método de agendamento da sala será chamado para finalizar o agendamento da consulta, e de seguinte a mensagem final de confirmação será apresentada.

Conteúdo do código (main)

```
elif opcao == "5":
    for s in salas: s.detalhar_sala()
```

Esta opção apenas mostra todas as salas.

```
elif opcao == "6":
    nome = input("Nome do funcionário: ").strip()
    f = next((x for x in funcionarios if x.nome.lower() == nome.lower()), None)
    if not f: print("Funcionário não encontrado."); continue
    if isinstance(f, (Enfermeiro, EnfermeiroChefe)):
        t = input("Novo turno (dia/noite, enter para manter): ").strip().lower()
        if t in ("dia","noite"): f.turno = t
    try:
        a = input("Percentual aumento (enter para manter): ").strip()
        if a: f.aplicar_aumento(float(a))
    except: print("Aumento inválido.")
    if isinstance(f, (Administrativo, EnfermeiroChefe)):
        try:
            h = input("Horas a registrar (enter para manter): ").strip()
            if h: f.registrar_horas(int(h))
        except: print("Horas inválidas.")
    print(f"Pagamento total: €{f.calcular_pagamento():.2f}")
```

Nesta opção é pedido para introduzir o número do funcionário, de seguinte, é procurado através da lista de funcionários por um funcionário que tenha o mesmo nome que foi introduzido, se não corresponder a mensagem de erro é mostrada, depois é verificado se o funcionário é um enfermeiro ou enfermeiro chefe, se for, é permitido mudar o turno desse funcionário, depois é permitido adicionar o aumento a esse funcionário através de uma percentagem, depois é verificado se o funcionário é administrativo ou enfermeiro chefe, se sim, é possível registrar as horas trabalhadas, se o dado escrito não for um número inteiro, dará erro, depois o método para calcular o aumento é chamado.

Conteúdo do código (main)

```
elif opcao == "7":
    if not pacientes: print("Nenhum paciente."); continue
    for i, p in enumerate(pacientes, 1): print(f"{i}. {p.nome}")
    try:
        paciente = pacientes[int(input("Escolha paciente: "))-1]
    except: print("Escolha inválida."); continue
    r = input("Digite o registro: ").strip()
    if r: paciente.adicionar_registro(r)
    print("Registro adicionado.")
```

Nesta opção primeiro é verificado se existe algum paciente, se houver é mostrado todos os pacientes de forma númerada, depois é pedido para escolher um paciente pelo número mas é removido um número porque o python começa com valor 0, depois é adicionado o registro ao paciente escolhido.

```
        elif opcao == "0":
            print("Saindo..."); break

        else:
            print("Opção inválida.")


if __name__ == "__main__":
    main()
```

Opção 0 encerra o código e se a opção não corresponder às possiveis, diz que é opção inválida.

Conclusão

Com este projeto, conseguimos desenvolver um sistema simples, mas bem completo, para ajudar no gerenciamento de um hospital. Ele permite cadastrar pacientes, marcar consultas, controlar salas e funcionários, além de registrar informações importantes de cada atendimento.

Durante a criação, aprendemos bastante sobre programação orientada a objetos, tratamento de erros, listas e como deixar o código mais organizado e fácil de entender. Também vimos na prática como a programação pode ser usada para resolver situações reais do dia a dia, como o controle de pacientes e horários.

No fim, o projeto cumpriu o que a gente queria: criar um sistema funcional, aprender com o processo e mostrar como a tecnologia pode facilitar o trabalho dentro de um hospital.