PROGRAMAÇÃO AVANÇADA

Padrões de Projeto

Alunos:

Luan Vasconcelos A. Figueiredo
Manoel Santana Neto

Padrões de Projeto

1

PROJETO ESCOLHIDO

Sistema de Gerenciamento de Clínica Médica

2

TECNOLOGIA ESCOLHIDA

Python

Padrões Escolhidos

CRIAÇÃO

1

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

ESTRUTURA

2

- Composite
- Decorator
- Flyweight

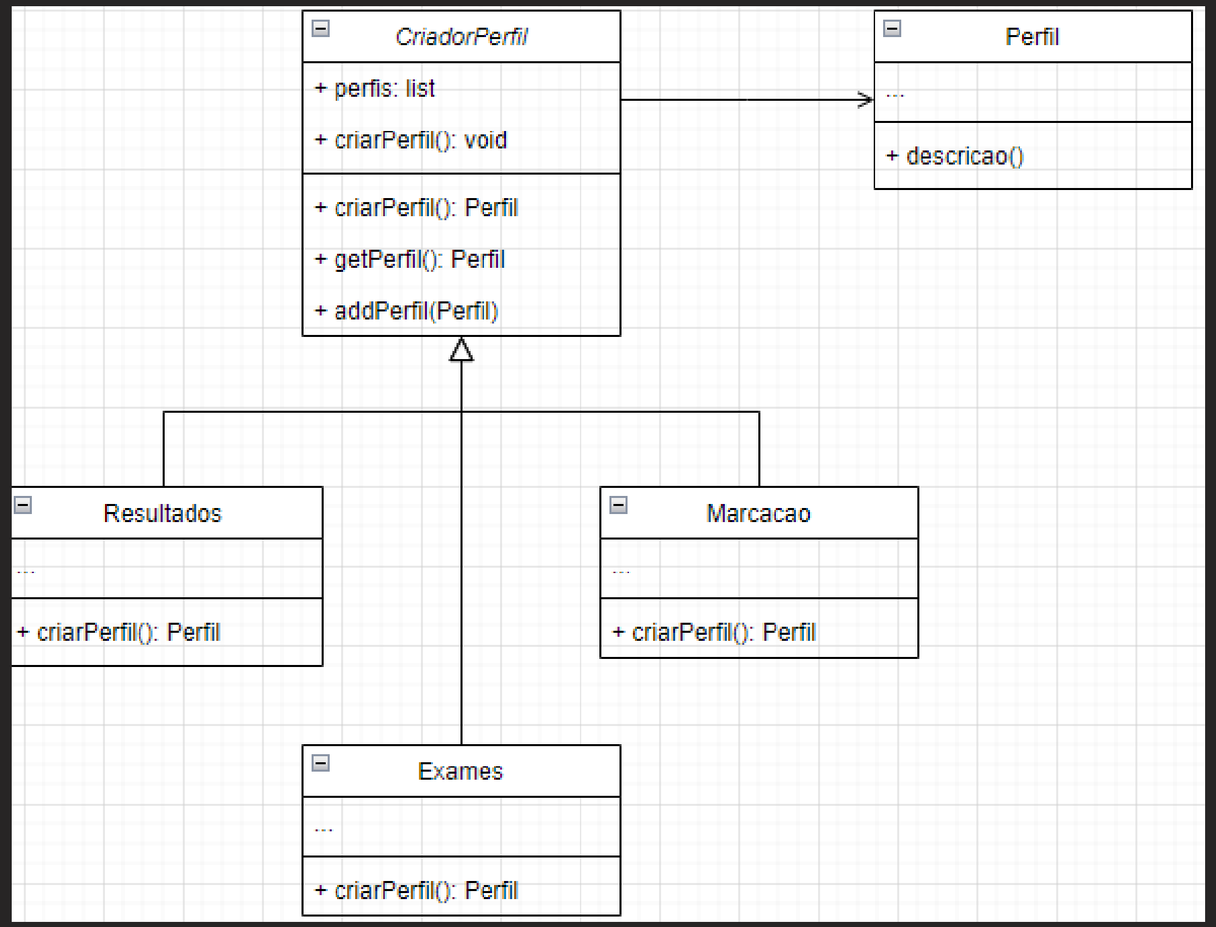
COMPORTAMENTO

3

- Iterator
- Memento
- State

Padrões de Criação
Factory Method

- Expõe um método ao cliente para criar os objetos
- Usa herança e subclasses para definir o objeto a ser criado



Padrões de Criação
Factory Method

```python
class Perfil(metaclass=ABCMeta):

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

```python
class PerfilPaciente(Perfil):
    def descricao(self):
        print("Perfil Paciente")
```

```python
class PerfilMedico(Perfil):
    def descricao(self):
        print("Perfil Médico")
```

```python
class PerfilAdmin(Perfil):
    def descricao(self):
        print("Perfil ADMIN")
```

Padrões de Criação
Factory Method

```python
class CriadorPerfil(metaclass=ABCMeta):
    def __init__(self):
        self.perfis = []
        self.criarPerfil()

    @abstractmethod
    def criarPerfil(self):
        pass

    def getPerfil(self):
        return self.perfis

    def addPerfil(self, perfil):
        self.perfis.append(perfil)
```

```python
class Exames(CriadorPerfil):
    def criarPerfil(self):
        self.addPerfil(PerfilMedico())
        self.addPerfil(PerfilAdmin())
```

```python
class Resultados(CriadorPerfil):
    def criarPerfil(self):
        self.addPerfil(PerfilMedico())
        self.addPerfil(PerfilPaciente())
        self.addPerfil(PerfilAdmin())
```

```python
class Marcacao(CriadorPerfil):
    def criarPerfil(self):
        self.addPerfil(PerfilMedico())
        self.addPerfil(PerfilPaciente())
        self.addPerfil(PerfilAdmin())
```

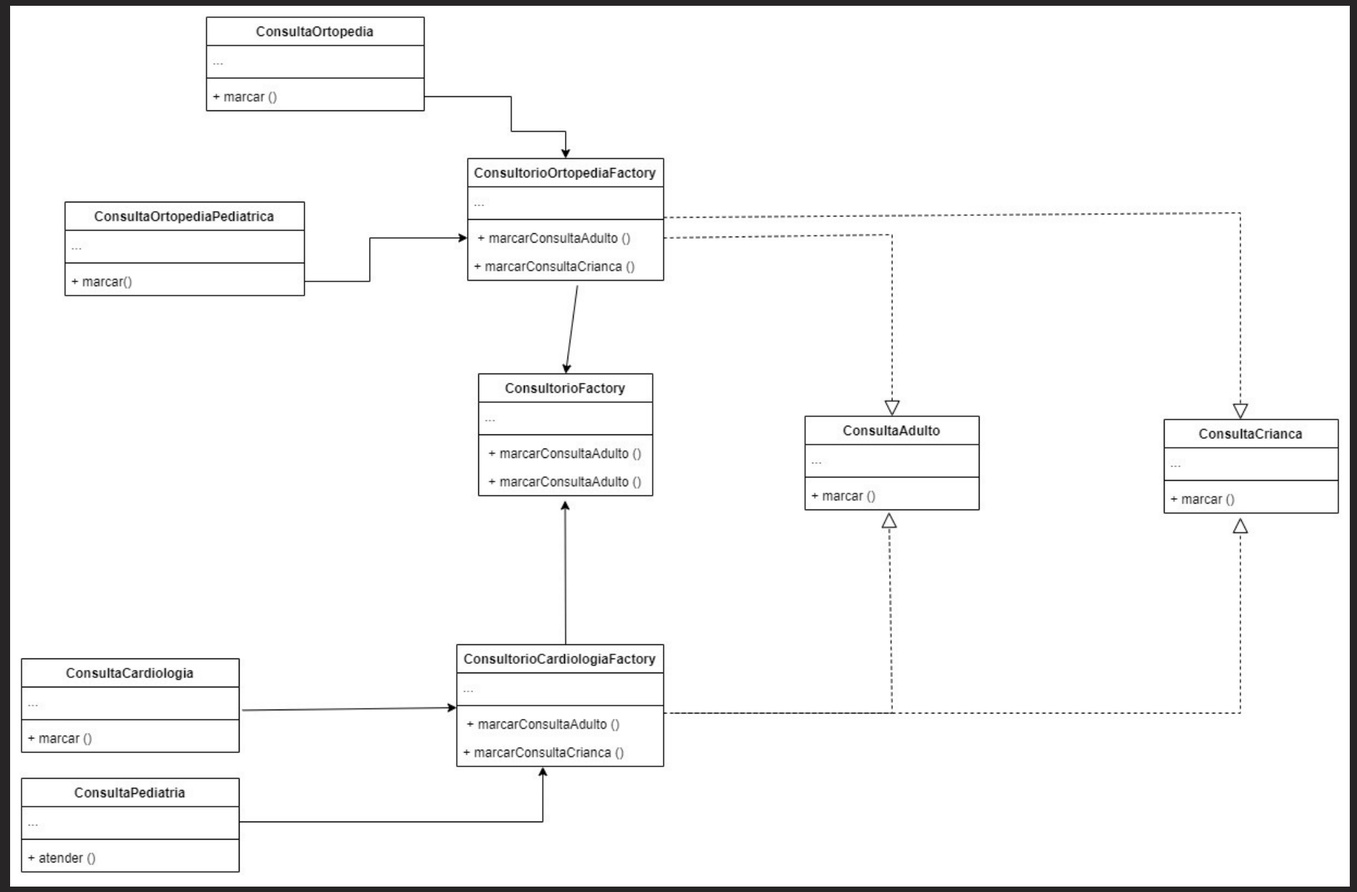
Padrões de Criação
Factory Method

```python
tipo_perfil = "Exames"
perfil = eval(tipo_perfil)()
print("Criando perfil", type(perfil).__name__)

tipo_perfil2 = "Resultados"
perfil2 = eval(tipo_perfil2)()
print("Criando perfil", type(perfil2).__name__)

tipo_perfil3 = "Marcacao"
perfil3 = eval(tipo_perfil3)()
print("Criando perfil", type(perfil3).__name__)
```

```
Criando perfil Exames
Criando perfil Resultados
Criando perfil Marcacao

Process finished with exit code 0
```

Padrões de Criação
Abstract Factory

- Contém um ou mais métodos de fábrica para criar uma família de objetos relacionados.
- Usa composição para delegar a responsabilidade de criar objetos de outra classe.



Padrões de Criação
Abstract Factory

```
class ConsultorioFactory(metaclass=ABCMeta):
    @abstractmethod
    def marcarConsultaAdulto(self):
        pass

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

Padrões de Criação
Abstract Factory

```
class ConsultorioCardiologiaFactory(ConsultorioFactory):
    def marcarConsultaAdulto(self):
        return ConsultaCardiologia()

    def marcarConsultaCrianca(self):
        return ConsultaPediatria()
```

```
class ConsultorioOrtopediaFactory(ConsultorioFactory):
    def marcarConsultaAdulto(self):
        return ConsultaOrtopedia()

    def marcarConsultaCrianca(self):
        return ConsultaPediatriaOrtopedica()
```

Padrões de Criação
Abstract Factory

```python
class ConsultaAdulto(metaclass=ABCMeta):
    @abstractmethod
    def marcar(self, consulta):
        pass


class ConsultaCrianca(metaclass=ABCMeta):
    @abstractmethod
    def marcar(self, consulta):
        pass
```

```python
class ConsultaCardiologia(ConsultaAdulto):
    def marcar(self):
        print('Marcando', type(self).__name__)


class ConsultaPediatria(ConsultaCrianca):
    def marcar(self):
        print('Marcando', type(self).__name__)


class ConsultaOrtopedia(ConsultaAdulto):
    def marcar(self):
        print('Marcando', type(self).__name__)


class ConsultaPediatriaOrtopedica(ConsultaCrianca):
    def marcar(self):
        print('Marcando', type(self).__name__)
```

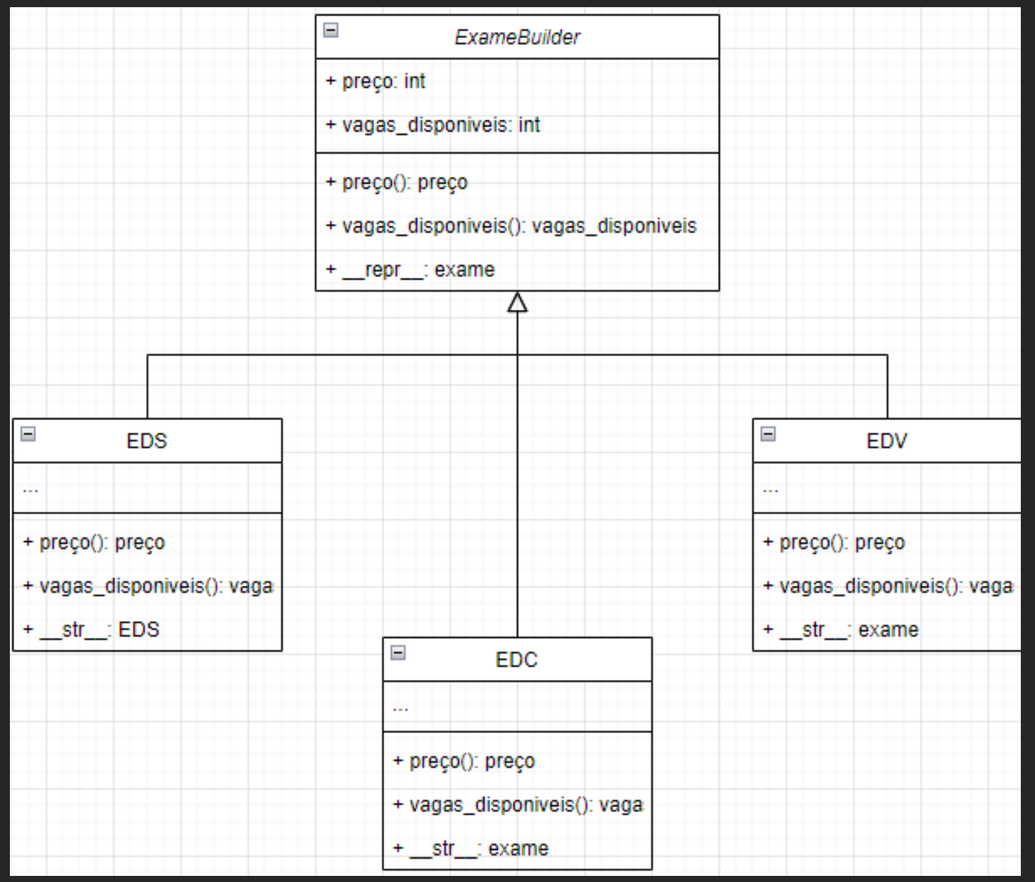
Padrões de Criação
Abstract Factory

```
class Consultorio:
    def consultorio(self):
        for factory in [ConsultorioCardiologiaFactory(), ConsultorioOrtopediaFactory()]:
            cons_crianca = factory.marcarConsultaCrianca()
            cons_adulto = factory.marcarConsultaAdulto()
            cons_adulto.marcar()
            cons_crianca.marcar()


if __name__ == '__main__':
    consultorio = Consultorio()
    consultorio.consultorio()
```

```
Marcando ConsultaCardiologia
Marcando ConsultaPediatria
Marcando ConsultaOrtopedia
Marcando ConsultaPediatriaOrtopedica

Process finished with exit code 0
```

Padrões de Criação
Builder

- Cria uma classe "Builder" que permite construir objetos complexos passo a passo
- Permite que você produza diferentes tipos e representações de um objeto usando o mesmo código de construção



Padrões de Criação
Builder

```
class Exame:

    def __init__(self):
        self.Preco()
        self.vagas_disponiveis()

    def Preco(self):
        raise NotImplementedError

    def vagas_disponiveis(self):
        raise NotImplementedError

    def __repr__(self):
        return 'Preco : {0.preco} | Vagas disponiveis : {0.vagas}'.format(self)
```

Padrões de Criação
Builder

```python
# concrete exames
class EDS(Exame):
    """Classe para Exame de Sangue

    def Preco(self):
        self.preco = 8000

    def vagas_disponiveis(self):
        self.vagas = 5

    def __str__(self):
        return "Exame de Sangue"
```

```python
# concrete exame
class EDV(Exame):
    """Class para Exame de Vista'

    def Preco(self):
        self.preco = 10000

    def vagas_disponiveis(self):
        self.vagas = 4

    def __str__(self):
        return "Exame de Vista"
```

```python
# concrete exame
class EDC(Exame):
    """Class for Exame de Coração'

    def Preco(self):
        self.preco = 5000

    def vagas_disponiveis(self):
        self.vagas = 7

    def __str__(self):
        return "Exame de Coração"
```

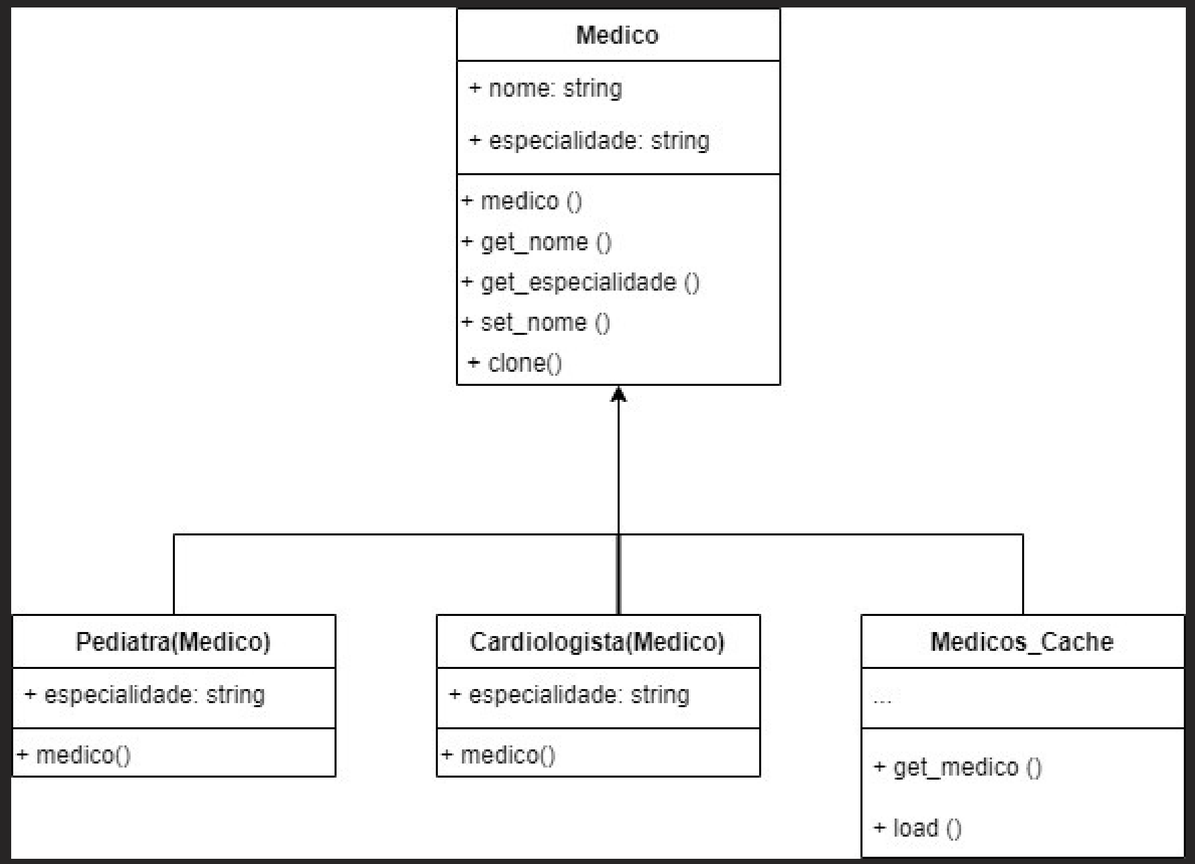
Padrões de Criação
Builder

```python
# main method
if __name__ == "__main__":
    eds = EDS()  # objeto para EDS Exame
    edv = EDV()  # objeto para EDV Exame
    edc = EDC()  # objeto para EDC Exame

    print(eds)
    print(eds.__repr__())
    print(f'Preço do Exame de sangue: {eds.preco}')
    print(f'Preço do Exame de Coração: {edc.preco}')
```

```
Exame de Sangue
Preco : 8000 | Vagas disponiveis :
Preço do Exame de sangue: 8000


Process finished with exit code 0
```

Padrões de Criação
Prototype

- Utilizado basicamente para clonar objetos
- Permite copiar objetos existentes sem fazer seu código ficar dependente de suas classes



Padrões de Criação
Prototype

```python
class Medico(metaclass=ABCMeta):

    # constructor
    def __init__(self):
        self.nome = None
        self.especialidade = None

    @abstractmethod
    def medico(self):
        pass

    def get_especialidade(self):
        return self.especialidade

    def get_nome(self):
        return self.nome

    def set_nome(self, snome):
        self.nome = snome

    def clone(self):
        return copy.copy(self)
```

```python
class Pediatra(Medico):
    def __init__(self):
        super().__init__()
        self.especialidade = "Pediatra"

    def medico(self):
        print("Dentro do método de Pediatra::medico()")


class Cardiologista(Medico):
    def __init__(self):
        super().__init__()
        self.especialidade = "Cardiologista"

    def medico(self):
        print("Dentro do método de Cardiologista::medico()")
```

Padrões de Criação
Prototype

```
class Medicos_Cache:

    cache = {}

    @staticmethod
    def get_medico(snome):
        Med = Medicos_Cache.cache.get(snome, None)
        return Med.clone()

    @staticmethod
    def load():
        pediatra = Pediatra()
        pediatra.set_nome("João")
        Medicos_Cache.cache[pediatra.get_nome()] = pediatra

        cardio = Cardiologista()
        cardio.set_nome("Fernando")
        Medicos_Cache.cache[cardio.get_nome()] = cardio
```

Padrões de Criação
Prototype

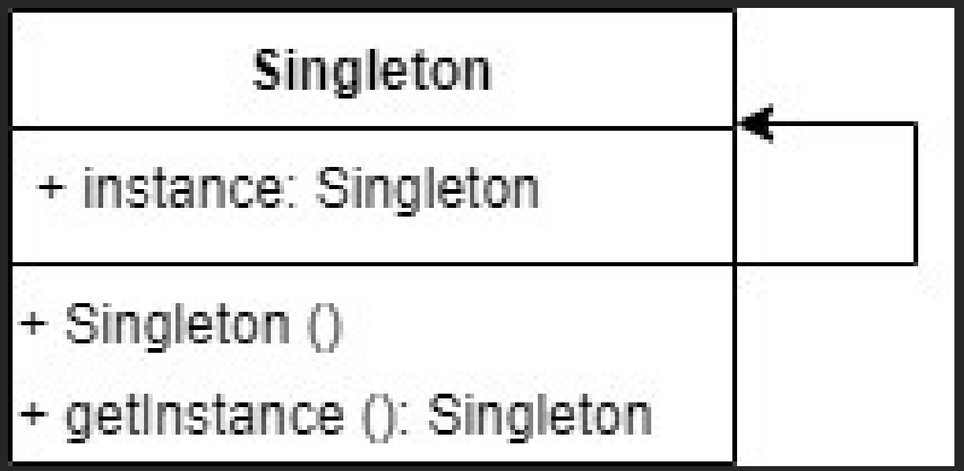
```python
class Medicos_Cache:

    cache = {}

    @staticmethod
    def get_medico(snome):
        Med = Medicos_Cache.cache.get(snome, None)
        return Med.clone()

    @staticmethod
    def load():
        pediatra = Pediatra()
        pediatra.set_nome("João")
        Medicos_Cache.cache[pediatra.get_nome()] = pediatra

        cardio = Cardiologista()
        cardio.set_nome("Fernando")
        Medicos_Cache.cache[cardio.get_nome()] = cardio
```

```python
# main function
if __name__ == '__main__':
    Medicos_Cache.load()

    pediatra = Medicos_Cache.get_medico("João")
    print(pediatra.get_especialidade())

    cardiologista = Medicos_Cache.get_medico("Fernando")
    print(cardiologista.get_especialidade())
```

```
Pediatra
Cardiologista

Process finished with exit code 0
```

Padrões de Criação
Singleton

- Possui um objetivo bem específico, garantir que apenas um objeto de uma determinada classe seja criado.



Padrões de Criação
Singleton

```python
class Singleton:
    __shared_instance = 'databasePath'

    @staticmethod
    def getInstance():

        """Metodo de acesso stático"""
        if Singleton.__shared_instance == 'databasePath':
            Singleton()
        return Singleton.__shared_instance

    def __init__(self):

        """virtual private constructor"""
        if Singleton.__shared_instance != 'databasePath':
            raise Exception("Essa é uma classe singleton !")
        else:
            Singleton.__shared_instance = self
```

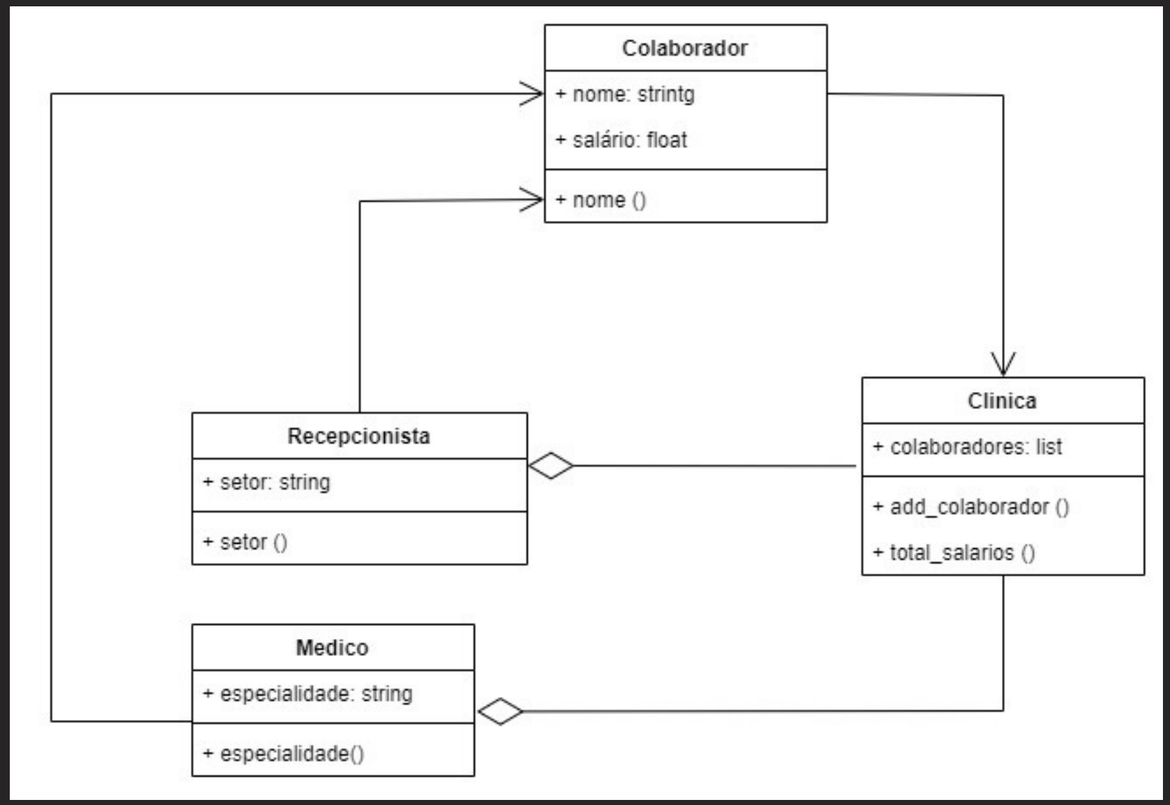
Padrões de Criação
Singleton

```
# main method
if __name__ == "__main__":
    # create object of Singleton Class
    obj = Singleton()
    print(obj)

    # pick the instance of the class
    obj = Singleton.getInstance()
    print(obj)
```

```
<__main__.Singleton object at 0x00000259CEF48BE0>
<__main__.Singleton object at 0x00000259CEF48BE0>

Process finished with exit code 0
```

Padrões de Estrutura
Composite

- Permite tratar objetos individuais de forma uniforme.
- Propósito é compor objetos em estruturas de árvores que representam hierarquias.



Padrões de Estrutura
Composite

```python
class Colaborador:
    def __init__(self, nome, salario):
        self.__nome = nome
        self.salario = salario

    @property
    def nome(self):
        return self.__nome
```

```python
class Recepcionista(Colaborador):
    def __init__(self, nome, salario, setor=None):
        super(Recepcionista, self).__init__(nome, salario)
        self.__setor = setor

    @property
    def setor(self):
        return self.__setor
```

```python
class Medico(Colaborador):
    def __init__(self, nome, salario, especialidade=None):
        super(Medico, self).__init__(nome, salario)
        self.__especialidade = especialidade

    @property
    def especialidade(self):
        return self.__especialidade
```

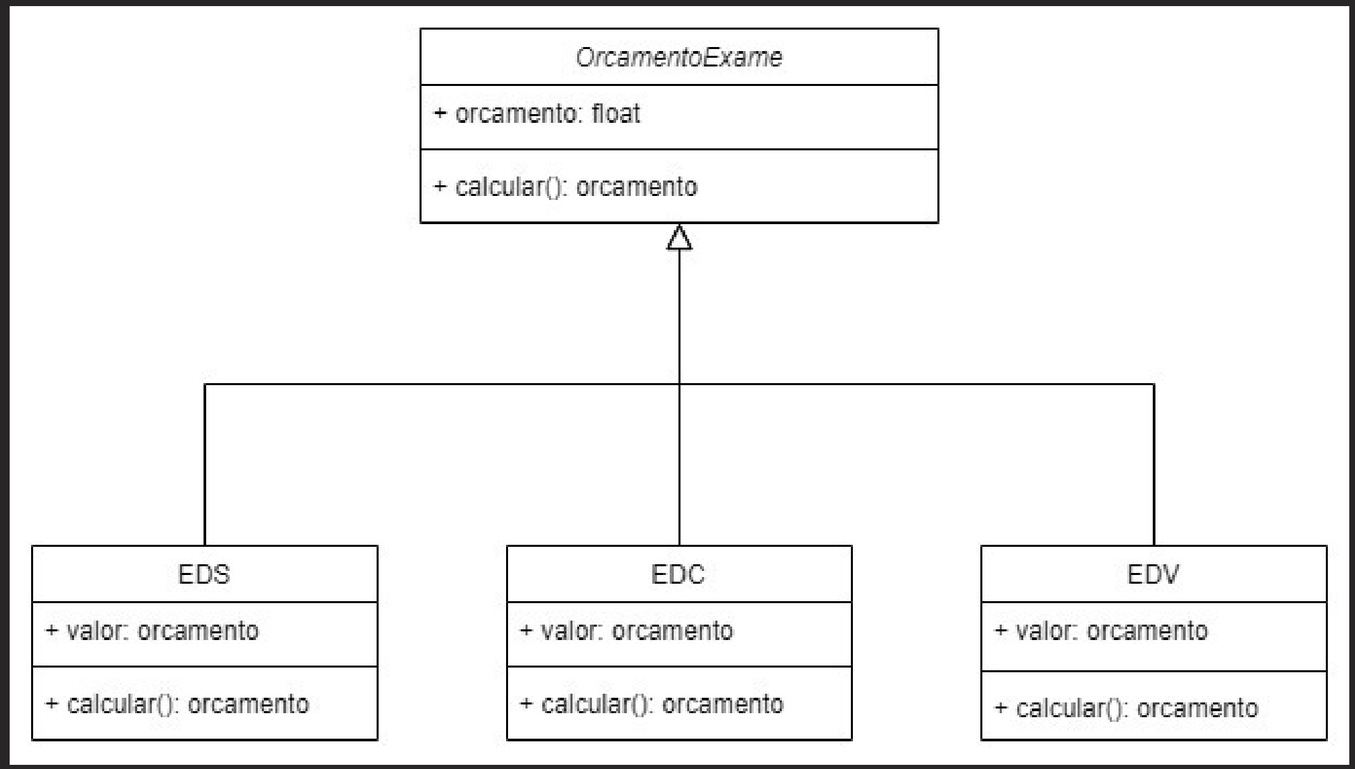
Padrões de Estrutura
Composite

```python
if __name__ == '__main__':
    joao = Recepcionista('Joao da Silva', 1800)
    carla = Medico('Carla Camila', 1900)

    clinica = Clinica()
    clinica.add_colaborador(joao)
    clinica.add_colaborador(carla)

    print(f'Total salários: {clinica.total_salarios()}')
```

```
Total salários: 3700


Process finished with exit code 0
```

Padrões de Estrutura
Decorator

- Permite compor/decorar os parâmetros de forma dinâmica.
- Propósito é compor objetos em estruturas de árvores que representam hierarquias.



Padrões de Estrutura
Decorator

```python
class OrcamentoExame:

    def __init__(self, orcamento):
        self._orcamento = orcamento

    def calcular(self):
        return self._orcamento
```

Padrões de Estrutura
Decorator

```python
class Particular(OrcamentoExame):

    def __init__(self, valor):
        self._valor = valor

    def calcular(self):
        return self._valor.calcular() * 0.90
```

```python
class Particular(OrcamentoExame):

    def __init__(self, valor):
        self._valor = valor

    def calcular(self):
        return self._valor.calcular() * 0.90
```

```python
class Cartao(OrcamentoExame):

    def __init__(self, valor):
        self._valor = valor

    def calcular(self):
        return self._valor.calcular() * 1.10
```

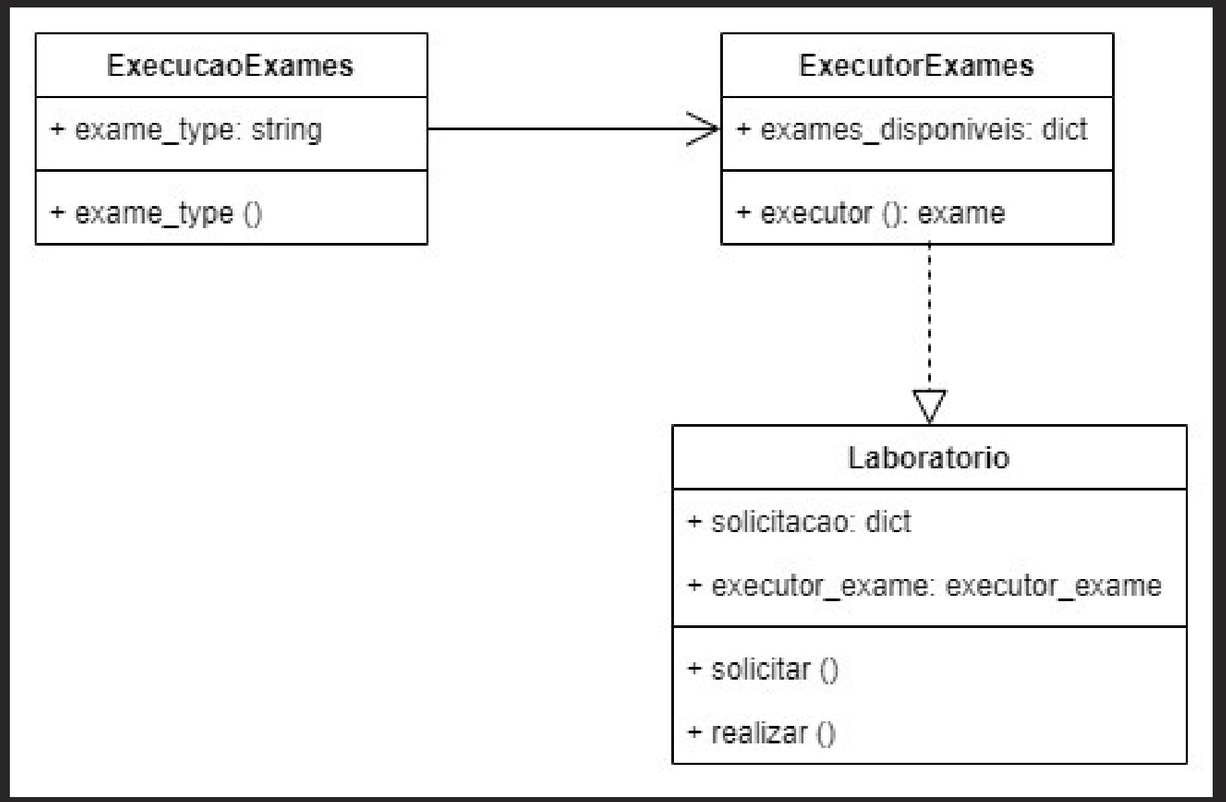
Padrões de Estrutura
Decorator

```
if __name__ == '__main__':

    orcamento = OrcamentoExame(1000)
    orcamento_particular = Particular(orcamento)
    orcamento_particular_a_vista = Particular(aVista(orcamento))
    orcamento_cartao = Cartao(orcamento)

    print(f'O valor original é: {orcamento.calcular()}')
    print(f'Desconto de 10% e o valor novo é: {orcamento_particular.calcular()}')
    print(f'Desconto de 10% no valor particular: {orcamento_particular_a_vista.calcular()}')
    print(f'O valor no cartão é: {orcamento_cartao.calcular()}')
```

```
O valor original é: 1000
Desconto de 10% e o valor novo é: 900.0
Desconto de 10% no valor particular: 810.0
O valor no cartão é: 1100.0

Process finished with exit code 0
```

Padrões de Estrutura
Flyweight

- Tem como objetivo minimizar o uso de memória ou custos computacionais
- Compartilhando a maior quantidade de dados entre objetos similares.



Padrões de Estrutura
Flyweight

```
class ExecucaoExames:
    def __init__(self, exame_type):
        self.__exame_type = exame_type

    @property
    def exame_type(self):
        return self.exame_type
```

Padrões de Estrutura
Flyweight

```
class ExecutorExames:
    def __init__(self):
        self.__exames_disponiveis = dict()

    def executor(self, escolha):
        if escolha not in self.__exames_disponiveis:
            self.__exames_disponiveis[escolha] = ExecucaoExames(escolha)
        return self.__exames_disponiveis[escolha]
```

Padrões de Estrutura
Flyweight

```python
class Laboratorio:
    def __init__(self, executor_exame):
        self.__solicitacao = dict()
        self.__executor_exame = executor_exame

    def solicitar(self, exame_type, sala):
        if sala not in self.__solicitacao:
            self.__solicitacao[sala] = list()
        self.__solicitacao[sala].append(self.__executor_exame.executor(exame_type))

    def realizar(self):
        for sala, solicitacao in self.__solicitacao.items():
            print(f'Exame realizado na sala {sala}')
```

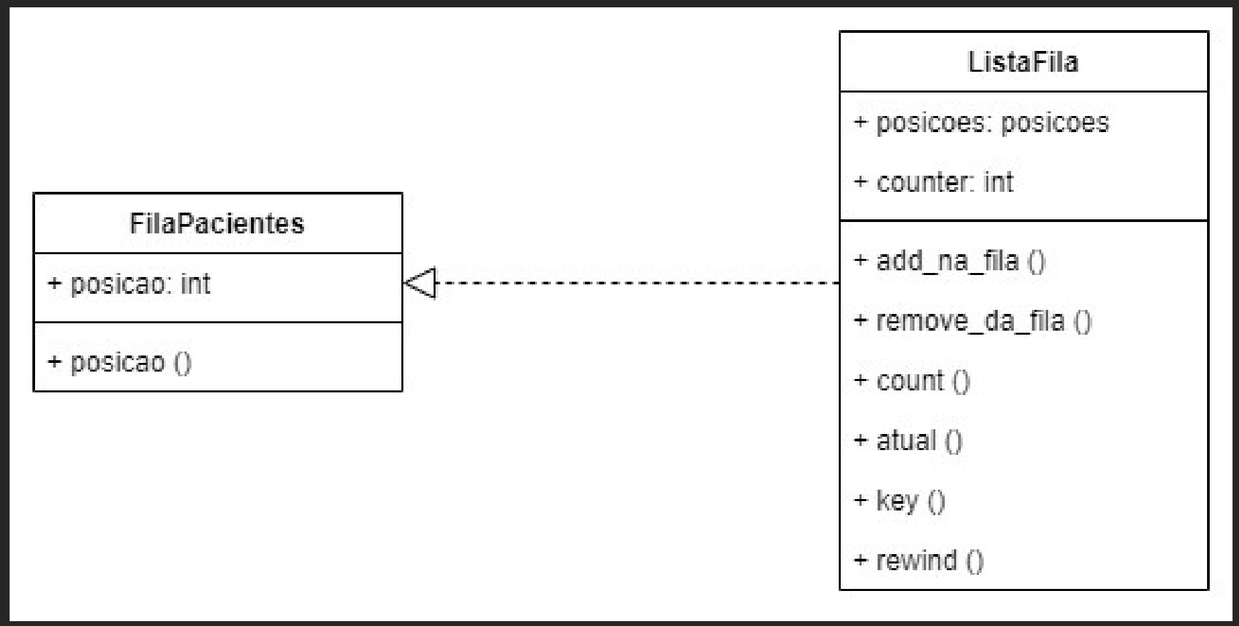
Padrões de Estrutura
Flyweight

```
if __name__ == '__main__':
    executor_exame = ExecutorExames()
    lab = Laboratorio(executor_exame)

    lab.solicitar('Exame de Sangue', 1)
    lab.solicitar('Exame de Vista', 2)
    lab.solicitar('Exame Cardíaco', 3)

    lab.realizar()
```

```
Exame realizado na sala 1
Exame realizado na sala 2
Exame realizado na sala 3


Process finished with exit code 0
```

Padrões de Comportamento
Iterator

- Disponibiliza uma maneira de acessar elementos de um objeto sem expor o conteúdo todo.
- Desacopla algoritmos de contêineres



Padrões de Comportamento

Iterator

```python
class FilaPacientes:
    def __init__(self, posicao):
        self.__posicao = posicao

    @property
    def posicao(self):
        return self.__posicao
```

```python
class ListaFila:
    def __init__(self):
        self.__posicoes = list()
        self.__counter = 0

    def add_na_fila(self, fila):
        self.__posicoes.append(fila)

    def remove_da_fila(self, posicao):
        for index in range(0, len(self.__posicoes)):
            if self.__posicoes[index].posicao == posicao:
                self.__posicoes.pop(index)
                break
        else:
            print('Paciente não encontrado na fila')

    def count(self):
        return len(self.__posicoes)

    def atual(self):
        return self.__posicoes[self.__counter].posicao

    def key(self):
        return self.__counter

    def __next__(self):
        self.__counter += 1

    def rewind(self):
        self.__counter = 0
```

Padrões de Comportamento
Iterator

```
class ListaFila:
    def __init__(self):
        self.__posicoes = list()
        self.__counter = 0

    def add_na_fila(self, fila):
        self.__posicoes.append(fila)

    def remove_da_fila(self, posicao):
        for index in range(0, len(self.__posicoes)):
            if self.__posicoes[index].posicao == posicao:
                self.__posicoes.pop(index)
                break
        else:
            print('Paciente não encontrado na fila')
```

```
    def count(self):
        return len(self.__posicoes)

    def atual(self):
        return self.__posicoes[self.__counter].posicao

    def key(self):
        return self.__counter

    def __next__(self):
        self.__counter += 1

    def rewind(self):
        self.__counter = 0
```

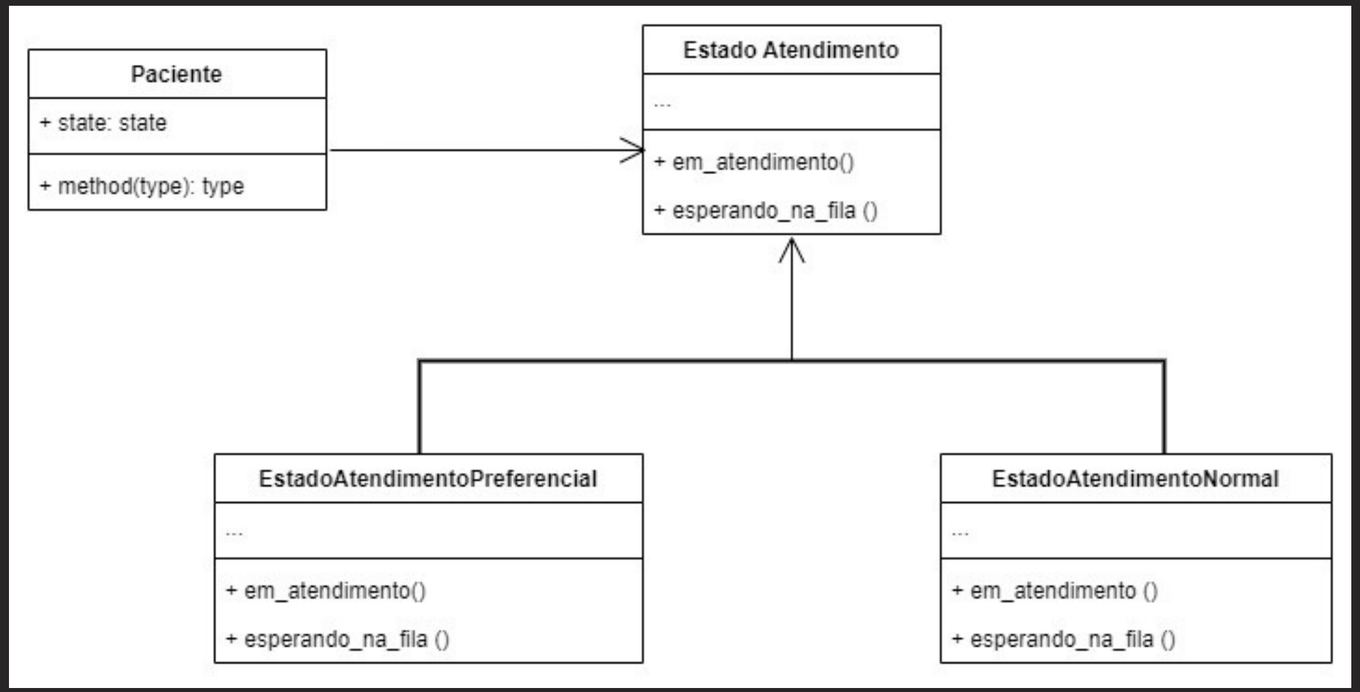
Padrões de Comportamento
Iterator

```python
if __name__ == '__main__':
    fila = ListaFila()

    fila.add_na_fila(FilaPacientes("João"))
    fila.add_na_fila(FilaPacientes("Fernando"))
    fila.add_na_fila(FilaPacientes("Márcia"))
    fila.add_na_fila(FilaPacientes("Júlia"))

    print(f'Quantidade de Pacientes na fila: {fila.count()}')
    fila.remove_da_fila("Márcia")
    print(f'Quantidade de Pacientes na fila: {fila.count()}')

    print(f'Ultimo paciente chamado: {fila.atual()}')
    next(fila)
    print(f'Ultimo paciente chamado: {fila.atual()}')
```

```
Quantidade de Pacientes na fila: 4
Quantidade de Pacientes na fila: 3
Ultimo paciente chamado: João
Ultimo paciente chamado: Fernando

Process finished with exit code 0
```

Padrões de Comportamento
State

- Pode ser utilizado quando precisamos definir um conjunto de estados e os mesmos possuem uma ordem bem definida



Padrões de Comportamento

State

```python
class EstadoAtendimento():
    @abstractmethod
    def em_atendimento(self):
        pass

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

```python
class EstadoAtendimentoPreferencial(EstadoAtendimento):
    def esperando_na_fila(self):
        return "O paciente com preferencial está aguardando na fila."

    def em_atendimento(self):
        return "O paciente com preferencial está sendo atendido."


class EstadoAtendimentoNormal(EstadoAtendimento):
    def esperando_na_fila(self):
        return "O paciente está aguardando na fila."

    def em_atendimento(self):
        return "O paciente está sendo atendido."
```

Padrões de Comportamento
State

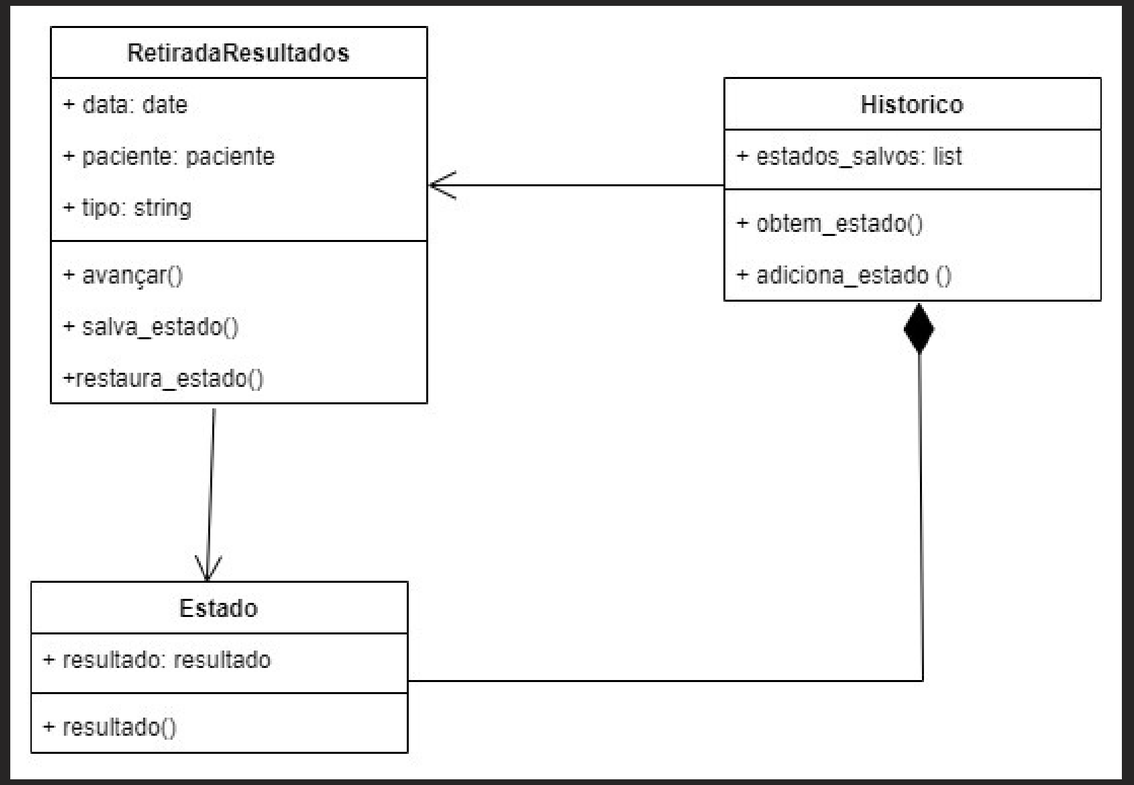
```
class Paciente(EstadoAtendimento):
    def __init__(self, state):
        self.state = state

    def set_state(self, state):
        self.state = state

    def esperando_na_fila(self):
        return self.state.esperando_na_fila()

    def em_atendimento(self):
        return self.state.em_atendimento()
```

```
if __name__ == '__main__':
    paciente = Paciente(EstadoAtendimentoNormal())
    print("Atendimento Normal: " + paciente.em_atendimento())
    print("Atendimento Normal: " + paciente.esperando_na_fila())

    paciente.set_state(EstadoAtendimentoPreferencial())
    print("Atendimento Preferencial: " + paciente.em_atendimento())
    print("Atendimento Preferencial: " + paciente.esperando_na_fila())
```

```
Atendimento Normal: O paciente está sendo atendido.
Atendimento Normal: O paciente está aguardando na fila.
Atendimento Preferencial: O paciente com preferencial está sendo atendido.
Atendimento Preferencial: O paciente com preferencial está aguardando na fila.

Process finished with exit code 0
```

Padrões de Comportamento
Memento

- Similar ao padrão **STATE**
- Podemos aplicar o memento sempre que desejamos guardar um estado que possa ser restaurado futuramente.



Padrões de Comportamento
Memento

```python
class RetiradaResultados:
    def __init__(self, data, paciente, tipo):
        self.data = data
        self.paciente = paciente
        self.tipo = tipo

    def avanca(self):
        if self.tipo == 'AGUARDANDO RESULTADO':
            self.tipo = 'DISPONÍVEL PARA RETIRADA'
        elif self.tipo == 'DISPONÍVEL PARA RETIRADA':
            self.tipo = 'RETIRADO'
        elif self.tipo == 'RETIRADO':
            self.tipo = 'CONCLUIDO'
```

```python
def salva_estado(self):
    # Não podemos passar o self para o RetiradaResultados pois se o resultado fosse
    # alterado o estado anterior dele também seria alterado
    return Estado(
        RetiradaResultados(data=self.data, paciente=self.paciente, tipo=self.tipo)
    )

def restaura_estado(self, estado):
    self.paciente = estado.resultado.paciente
    self.data = estado.resultado.data
    self.tipo = estado.resultado.tipo
```

Padrões de Comportamento
Memento

```python
class Estado:
    def __init__(self, resultado):
        self.__resultado = resultado

    @property
    def resultado(self):
        return self.__resultado
```

```python
class Historico:
    def __init__(self):
        self.__estados_salvos = list()

    def obtem_estado(self, indice):
        return self.__estados_salvos[indice]

    def adiciona_estado(self, estado):
        self.__estados_salvos.append(estado)
```

Padrões de Comportamento
Memento

```
if __name__ == '__main__':

    historico = Historico()

    resultado = RetiradaResultados(data=date.today(), paciente='Joao da Silva Santos', tipo='AGUARDANDO RESULTADO')
    print(resultado.paciente)
    print(resultado.tipo)

    resultado.avanca()

    print(resultado.paciente)
    print(resultado.tipo)
    historico.adiciona_estado(resultado.salva_estado())
```

```
Joao da Silva Santos
AGUARDANDO RESULTADO
Joao da Silva Santos
DISPONÍVEL PARA RETIRADA
```

Padrões de Comportamento
Memento

```
resultado.avanca()

print(resultado.paciente)
print(resultado.tipo)

# Caso onde o nome teria sido errado
resultado.paciente = 'Joao Ferreira Santos'

historico.adiciona_estado(resultado.salva_estado())

print(resultado.paciente)
print(resultado.tipo)
```

```
Joao da Silva Santos
RETIRADO
Joao Ferreira Santos
RETIRADO
```

Padrões de Comportamento
Memento

```
resultado.avanca()

historico.adiciona_estado(resultado.salva_estado())

print(resultado.paciente)
print(resultado.tipo)

resultado.restaura_estado(historico.obtem_estado(0))

print(resultado.paciente)
print(resultado.tipo)
```

```
Joao Ferreira Santos
CONCLUIDO
Joao da Silva Santos
DISPONÍVEL PARA RETIRADA

Process finished with exit code 0
```

OBRIGADO!