Universidade Federal do Piauí - UFPI

Sistemas de Informação

Programação Orientada a Objetos I - POO

Herança x Polimorfismo

Prof. Flávio Araújo - UFPI - Picos PI

Repetindo código

- Como toda empresa, nosso banco possui funcionários. Um funcionário possui nome, cpf e salário.
- Além de um funcionário comum, há também outros cargos, como os gerentes, que guardam as mesmas informações de um funcionário comum além de uma senha de acesso ao sistema e número de funcionários que ele gerencia.

```
class Funcionario:

    def __init__(self, nome, cpf, salario):
        self._nome = nome
        self._cpf = cpf
        self._salario = salario
```

```
class Gerente:

    def __init__(self, nome, cpf, salario, senha, qtd_gerenciados):
        self._nome = nome
        self._cpf = cpf
        self._salario = salario
        self._senha = senha
        self._qtd_gerenciados = qtd_gerenciados
```

Herança

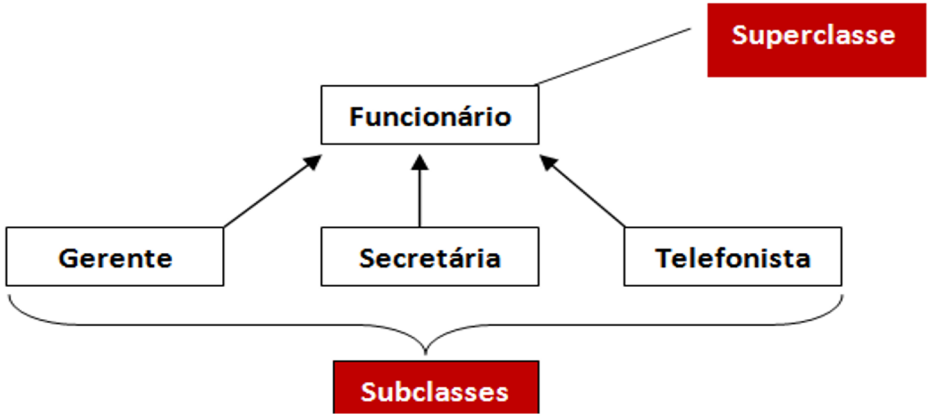
- Se tivéssemos um outro tipo de funcionário que tem características diferentes do funcionário comum, precisaríamos criar outra classe e copiar o código novamente. Além disso, se um dia aparecer uma nova funcionalidade para todos os funcionários, precisaríamos passar por todas as classes de funcionários e adicionar esse atributo.
- Existe um jeito de relacionarmos uma classe de tal maneira que uma delas herda tudo o que a outra tem. Isto é uma relação de herança entre classe “mãe” e classe “filha”.

```
class Gerente(Funcionario):

  def __init__(self, nome, cpf, salario, senha, qtd_funcionarios):
      super().__init__(nome, cpf, salario)
      self._senha = senha
      self._qtd_funcionarios = qtd_funcionarios
```

Herança

- A classe Gerente herda os atributos e métodos ‘privados’ de Funcionario. O super() é usado para referenciar a superclasse, a classe mãe, no nosso exemplo é Funcionario.
- A nomenclatura mais encontrada é que Funcionario é a **superclasse** de Gerente, e Gerente é a **subclasse** de Funcionario. Dizemos também que todo Gerente é um Funcionario. Outra forma é dizer que Funcionario é a classe **mãe** de Gerente e Gerente é a classe **filha**.



Herança: Definição

- Herança é um princípio de orientação a objetos, que permite que classes compartilhem atributos e métodos, através de "heranças". Ela é usada na intenção de reaproveitar código ou comportamento generalizado ou especializar operações ou atributos. O conceito de herança de várias classes é conhecido como herança múltipla.

Reescrita de métodos

- Todos os funcionários do banco recebem uma bonificação no final do ano. Os funcionários comuns recebem 10% do valor do salário e os gerentes recebem 15%.
- Se deixarmos a classe Gerente como está, ela vai herdar o método get_bonificação. Então temos que reescrever (sobrescrever, overrride) este método, assim como fizemos com __init__.

```
class Funcionario:

    def __init__(self, nome, cpf, salario):
        self._nome = nome
        self._cpf = cpf
        self._salario = salario

    # outros métodos e properties

    def get_bonificacao(self):
        return self._salario * 0.10
```

```
class Gerente(Funcionario):

    def __init__(self, nome, cpf, salario, senha, qtd_gerenciaveis):
        super().__init__(nome, cpf, salario)
        self._senha = senha
        self._qtd_gerenciaveis = qtd_gerenciaveis

    def get_bonificacao(self):
        return self._salario * 0.15
```

Polimorfismo

Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas. Isso não quer dizer que o objeto fica se transformando, muito pelo contrário, um objeto nasce de um tipo e morre daquele tipo, o que pode mudar é a maneira como nos referimos a ele.

```
class ControleDeBonificacoes:

    def __init__(self, total_bonificacoes=0):
        self._total_bonificacoes = total_bonificacoes

    def registra(self, funcionario):
        self._total_bonificacoes += funcionario.get_bonificacao()

    @property
    def total_bonificacoes(self):
        return self._total_bonificacoes
```

```
funcionario = Funcionario('João', '111111111-11', 2000.0)
print("bonificacao funcionario: {}".format(funcionario.get_bonificacao()))

gerente = Gerente("José", "222222222-22", 5000.0 '1234', 0)
print("bonificacao gerente: {}".format(gerente.get_bonificacao()))

controle = ControleDeBonificacoes()
controle.registra(funcionario)
controle.registra(gerente)

print("total: {}".format(controle.total_bonificacoes))
```

Polimorfismo

- No dia que criarmos uma classe Secretaria, que é filha de Funcionario, precisaremos mudar a classe ControleDeBonificacoes?
    - Não. Basta a classe Secretaria reescrever os métodos que lhes parecerem necessários. É exatamente esse o poder do polimorfismo juntamente com a reescrita do método: diminuir o acoplamento entre as classes para evitar que novos códigos resultem em modificações em inúmeros lugares.
- Perceba que quem criou ControleDeBonificaçoes pode nunca ter imaginado a criação da classe Secretaria. No entanto, não será necessário reimplementar esse controle para cada nova classe.
- Como Python não tem tipagem dinâmica, caso seja passado um objeto de uma classe que não possui o método get_bonificacao(), ocorrerá um erro.

Exercício

- Façam as questões 1, 2, 3, 4, 5 e 6 nas páginas 145, 146 e 147 da apostila da Caelum.

10.7 EXERCÍCIO: HERANÇA E POLIMORFISMO

Classes Abstratas

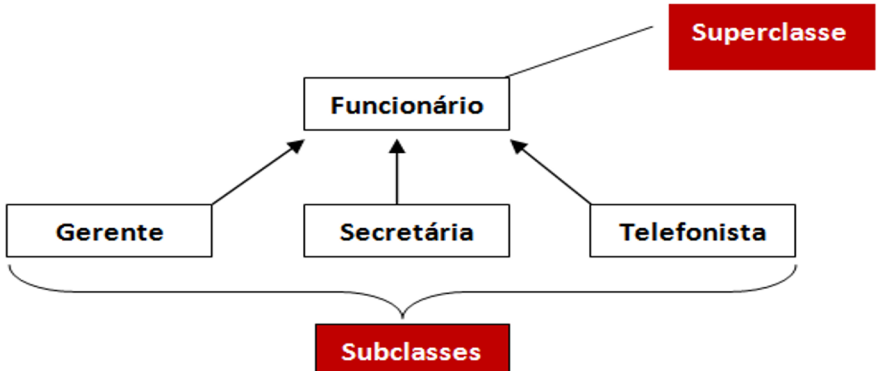
- Uma classe abstrata serve de modelo para outras classes, ou seja, novas classes herdam atributos e métodos dela. Uma classe abstrata não pode ser instanciada.
- No exemplo abaixo não faz sentido termos objetos da classe Funcionario. Por tanto, ela deve ser uma classe abstrata. Além disso, para que o método registra funcione, objetos derivados de funcionário devem implementar get_bonificacao(), portanto, esse método deve ser abstrato.

```
class ControleDeBonificacoes:

    def __init__(self, total_bonificacoes=0):
        self._total_bonificacoes = total_bonificacoes

    def registra(self, funcionario):
        self._total_bonificacoes += funcionario.get_bonificacao()

    @property
    def total_bonificacoes(self):
        return self._total_bonificacoes
```



Classes Abstratas

Classe Funcionario e método get_bonificacao() são abstratos

```
import abc

class Funcionario(abc.ABC):

    @abc.abstractmethod
    def get_bonificacao(self):



if __name__ == '__main__':
    f = Funcionario()
```

Classe Gerente é obrigada a implementar método abstrato

```
class Gerente(Funcionario):
    # outros métodos e propriedades

    def get_bonificacao(self):
        return self._salario * 0.15

if __name__ == '__main__':
    gerente = Gerente('jose', '222222222-22', 5000.0, '1234', 0)
    print(gerente.get_bonificacao())
```

Exercício

- Façam as questões 1, 2, 3, 4, 5, 6, 7, 8 e 9 nas páginas 150, 151 e 152 da apostila da Caelum.

10.9 EXERCÍCIOS - CLASSES ABSTRATAS