



LINGUAGEM DE PROGRAMAÇÃO APLICADA

AULA 2



Prof. Osmar Dias Junior
Prof. Renan Jorge



CONVERSA INICIAL

Nesta etapa, abordaremos as técnicas de aprofundamento do paradigma de programação orientado a objeto. Será utilizada a linguagem Python e o ambiente de desenvolvimento integrado Pycharm.

IDE significa Ambiente de Desenvolvimento Integrado (*Integrated Development Environment*, em inglês). É o software que fornece um conjunto abrangente de ferramentas e recursos para desenvolvedores de software escreverem, testarem e depurarem seus programas.

Vamos rever as bases dessa técnica colocando a classe como uma estrutura de atributos e métodos. Isso significa que a composição de uma classe inclui a lista de variáveis que conterão as características encerradas por ela e as funções adequadas para executar as ações por ocasião do uso dessa estrutura.

Vamos explorar o uso desse conjunto por meio do que chamamos de 'instanciar um objeto', que é a forma de utilização do modelo constituído pela classe como uma receita que o objeto passa a possuir nesse momento da programação. Os objetos podem ser instanciados em qualquer quantidade e passam a existir com as características formadas pela classe. Isso nos indica que podemos replicar uma mesma estrutura em um número virtualmente infundável de aplicações com as mesmas formas e ações da origem, que é a classe.

Na continuação veremos que uma classe pode herdar características de outra classe. Quando a classe tem essa relação de herança e recebe os atributos e métodos da primeira, ela recebe o título de subclasse ou classe filha, por estar abaixo no fluxo de informações.

A classe que transmite suas formas é chamada de superclasse ou classe pai. Dessa forma, o objeto instanciado por uma subclasse pode usufruir de tudo que a superclasse propõe, não sendo necessário instanciar um objeto para cada uma das classes acima e abaixo. Pode-se instanciar o objeto somente referenciado na subclasse, que pode utilizar tudo que está na superclasse automaticamente.

Seguindo pelos temas desta etapa, teremos uma introdução na diagramação UML, que significa Diagrama de Modelagem de Linguagem Unificada (*Unified Modeling Language*, em inglês). É uma linguagem



padronizada para visualizar, especificar, construir e documentar os artefatos de um programa de computador. O UML oferece uma variedade de diagramas que permite representar diferentes aspectos de um sistema, mas aqui nesta etapa teremos uma focalização nas estruturas de classes e suas relações de forma a auxiliar na composição dos sistemas mais complexos.

A técnica de planejar o programa utilizando UML permite que o programador tenha uma visão ampla da estrutura do sistema em construção, de forma a facilitar a programação e torná-la mais compacta e eficiente. Isso não é possível atingir sem um bom planejamento prévio.

Avançando nas aplicações de classes, objetos e suas relações de herança, juntamente com o planejamento em UML, teremos as aplicações práticas de como trabalhar com essas ferramentas. Dessa forma, é recomendável sempre começar um projeto pelo planejamento cuidadoso, seguindo para um mapeamento tão detalhado quanto possível em diagrama UML e, por fim, construir o programa baseado nas decisões previamente planejadas.

No tema que fecha essa sequência teremos uma introdução sobre como transformar o diagrama UML em código de programa de forma automática. Embora o processo não seja totalmente perfeito e não entregue um programa completo e funcional, ele pode realizar mais da metade da estrutura do programa, mas já em uma estrutura baseada no planejamento realizado na etapa da diagramação das classes. O software utilizado para essa tarefa será o StarUML em sua versão de experimentação.

TEMA 1 – COMEÇANDO COM CLASSES

1.1 Classe = Atributos + Métodos

Em programação usamos a palavra paradigma para definir uma forma de programar baseada em algum tipo de padrão. Por exemplo: a primeira forma de programar que as pessoas aprendem é colocar uma linha depois da outra formando uma sequência de instruções. Essa forma de programar é conhecida como “paradigma procedural”, ou seja, é um padrão de programação baseado em uma sequência de comandos que, ao serem executados, produzem um resultado global.



Com a evolução da computação, vários outros tipos de conduta ao programar, vários tipos de paradigmas, foram criados. Sempre foi uma ação comum o programador “inventar” uma forma de programar de acordo com suas necessidades particulares ao momento que estava. Ainda hoje isso acontece, mas como o desenvolvimento do pensamento de programação evoluiu grandemente e já se sabe uma forma de programar que é a melhor para a maioria dos casos, essa forma acaba se consagrando com um paradigma, um padrão de comportamento em programação.

No atual cenário de programação em que as máquinas oferecem um desempenho muito grande e as informações precisam trafegar de forma eficiente e veloz, um paradigma se sobressai como predominante, pois resolve a maioria esmagadora das necessidades dos códigos: o paradigma da programação orientada a objetos. A principal utilidade desse modo de pensar a programação consiste em poder criar um modelo e replicar tantas vezes quantas necessárias de acordo com o sistema em construção.

Por exemplo: em uma empresa é necessário haver um cadastro dos funcionários. Esse cadastro pode ser feito de várias maneiras diferentes na programação do sistema. Porém, a orientação a objeto transforma uma tarefa que seria árdua em uma tarefa extremamente simples e rápida. Basta criar uma lista de características, conhecidas como atributos, de cada funcionário. Ao fazer isso, poderíamos perceber quais dessas características são comuns a todos os funcionários. Todos são pessoas, têm seus nomes, moram em algum endereço, possuem certa idade, formação acadêmica, cargo que ocupa na empresa, entre outras. Esse cadastro precisaria ficar armazenado em um banco de dados, mas a forma como o programa lida com esses dados é o que realmente importa nesse momento.

Se tivermos todos os registros feitos dentro do nosso programa, isso traria a necessidade de grandes volumes de memória, uma vez que sempre teríamos as informações colocadas em variáveis, o que ocupa memória. Haveria um esforço computacional bastante intenso para gerenciar esse cadastro, principalmente se houvessem muitas pessoas. Poderia ser um cadastro de cidadãos em uma cidade, o que pode ser da ordem de alguns milhões de pessoas. Então, devido à clara necessidade de um banco de memória especializado na guarda e proteção dessas informações, um banco



de dados, precisamos pensar em como nosso sistema trabalhará em conjunto com o armazenamento.

Uma forma poderia ser pegar sempre os dados do banco e carregar em uma variável, fazer o que precisar com a informação e apagar ao final. Pode ser feito assim, mas existe uma forma mais eficiente e que diminui significativamente o esforço computacional e traz outras vantagens para a eficiência do sistema: a programação orientada a objetos.

No caso do cadastro, o que pode ser feito é criar uma estrutura de variáveis, que são os atributos, e alguns módulos de funções, conhecidos como métodos. Isso significa que haverá um conjunto de informações e procedimentos que somente estarão em funcionamento quando necessário, quando forem “chamados”. Se não houver necessidade, nada ocupará espaço em memória nem processamento, somente quando houver a provocação do processo.

Para o perfeito funcionamento dessa lógica é feito um modelo com todos os atributos e todos os métodos que podem ser utilizados em relação a esse modelo. Esse conjunto de atributos e métodos é montado dentro do que chamamos de CLASSE.

Sempre que houver necessidade é criado um objeto baseado nos atributos específicos do elemento a ser processado e os métodos ficam disponíveis para trabalhar com esse. No exemplo do cadastro de funcionários podemos modelar da seguinte forma: uma classe seria criada com atributos e métodos relacionados com o cadastro dos funcionários. Os atributos seriam os dados que todos têm, tais como nome, idade, endereço, telefone, cargo, salário, data de contratação. Para trabalhar com essas informações precisaríamos poder criar um registro, para incluir funcionários novos, consultar registros, alterar registro e apagar.

Como primeiro exercício vamos criar uma classe Funcionario (sem acento de propósito) com os atributos Nome, Cargo e Salário somente. E vamos colocar só um método para mostrar cada registro, por enquanto, que chamaremos MostrarReg(). O diagrama dessa primeira “CLASSE Funcionario” está apresentado na Figura 1, a seguir.



Figura 1 – Diagrama UML da CLASSE FUNCIONÁRIO



Fazer a classe é o primeiro grande passo para programarmos utilizando o paradigma de orientação a objetos. A classe sozinha, por si só, não faz nada, é somente o molde para podermos criar os registros dos funcionários que existem de verdade. É necessário que um OBJETO seja criado com base na classe. Esse processo é chamado de “INSTANCIAR UM OBJETO”. Isso significa que podemos criar um objeto com os dados do funcionário Antônio, outro objeto da Maria, um outro do Pedro e mais um da Zilda, por exemplo. Vamos fazer esse primeiro exercício na linguagem de programação Python, no ambiente PyCharm. Para isso escrevemos o seguinte código:

Figura 2 – Código para criar a classe Funcionario, instanciar 4 objetos e usar o método MostraReg() para cada um deles

```
# Criação da classe funcionário
# A classe sempre precisa ser iniciada pela instância de algum objeto
# Sozinha e sem invocada em um objeto ela não faz nada
class Funcionario:
    def __init__(self, in_nome: str, in_cargo: str, in_salario: int):
        self.nome: str = in_nome
        self.cargo: str = in_cargo
        self.salario: int = in_salario

    def MostraReg(self):
        print(f'O nome do funcionário é {self.nome}')
        print(f'O cargo do {self.nome} é {self.cargo}')
        print(f'O salário do {self.cargo} {self.nome} é ${self.salario}')
        print()

# Início do Main
# o corpo principal do programa onde começa a ser rodado

# Instanciando os objetos baseados na classe Funcionario() com os parâmetros
func1 = Funcionario(in_nome='Antônio', in_cargo='Gerente', in_salario=5000)
func2 = Funcionario(in_nome='Maria', in_cargo='Diretora', in_salario=25000)
func3 = Funcionario(in_nome='Pedro', in_cargo='Redator', in_salario=4000)
func4 = Funcionario(in_nome='Zilda', in_cargo='Supervisora', in_salario=7500)
```



```
# Chamando o método MostraReg() para cada funcionário 'func' instanciado
func1.MostraReg()
func2.MostraReg()
func3.MostraReg()
func4.MostraReg()
```

1º - Criação da CLASSE Funcionario, de acordo com o diagrama transformado para o código em Python:

Figura 3 – Criação da classe Funcionario

```
main.py x
1 # Criação da classe funcionário
2 # A classe sempre precisa ser iniciada pela instância de algum objeto
3 # Sazinha é sem invocada em um objeto ela não faz nada
4 usages
5 class Funcionario:
6     def __init__(self, in_nome: str, in_cargo: str, in_salario: int):
7         self.nome: str = in_nome
8         self.cargo: str = in_cargo
9         self.salario: int = in_salario
10
11     4 usages
12     def MostraReg(self):
13         print(f'O nome do funcionário é {self.nome}')
14         print(f'O cargo do {self.nome} é {self.cargo}')
15         print(f'O salário da {self.cargo} {self.nome} é ${self.salario}')
16         print() # pular uma linha
```

Na Figura 2, vemos a classe Funcionario sendo criada em código Python. Observar os detalhes:

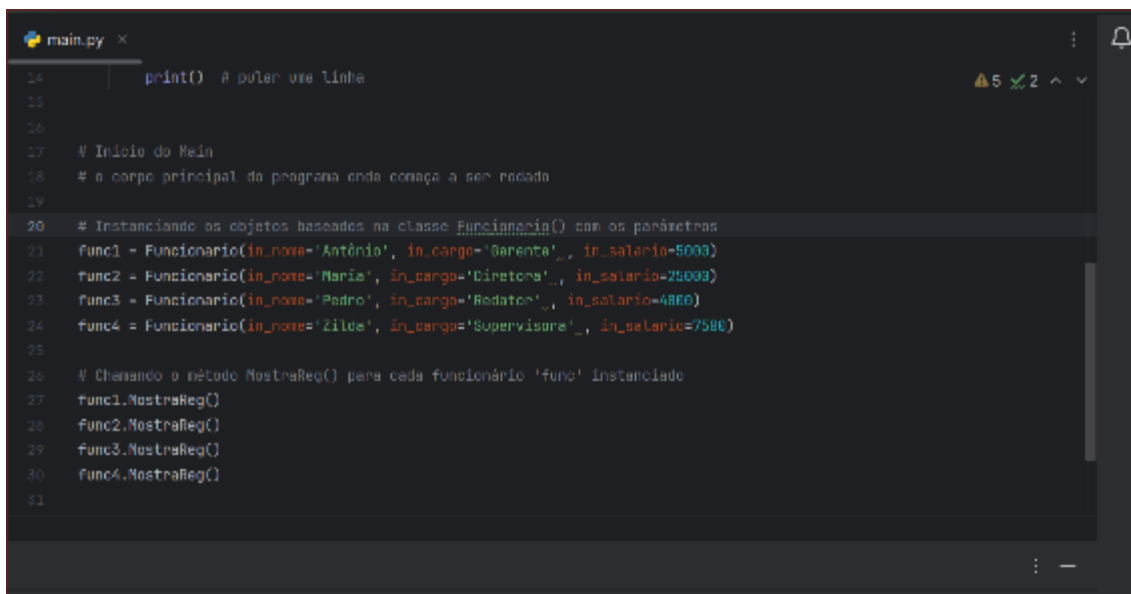
- Linha 1: o comentário para lembrarmos qual ação está sendo feita;
- Linha 2: o comando para definição de uma classe em Python: 'class Funcionario:'. 'Funcionario' foi escrito sem acento propositalmente;
- Linha 3: é feita a definição da função de construção da classe. O construtor declara as variáveis que serão usadas pela classe. Detalhes importantes: a palavra 'self' é utilizada sempre em primeiro lugar na lista de parâmetros a serem inseridos na inicialização. É obrigatório inserir esses parâmetros se eles estiverem declarados dentro dos parêntesis. Se o programador não quiser que seja obrigatório, basta não colocar esses parâmetros na inicialização. Os parâmetros ainda não são os atributos ou as variáveis da classe, isso será feito dentro da função __init__ logo em seguida;



- Linhas 4, 5 e 6: as variáveis que são os atributos da classe estão declaradas com a palavra 'self' na frente, separada por ponto. Detalhe muito importante: observe que estamos criando o atributo nome escrevendo 'self.nome' e já inicializamos colocando algo nessa variável. Esse conteúdo colocado na inicialização é o parâmetro inserido na inicialização da classe, dizemos "na construção da classe". Essa construção sempre acontecerá quando a classe for utilizada por ocasião de uma ação de instanciar um objeto, o que será feito a seguir;
- Linha 8: definição da função interna da classe, conhecida como 'Método'. O método será chamado de MostraReg(). Terá a funcionalidade de mostrar os dados relativos ao funcionário do qual o registro estiver sendo chamado;
- Linhas 9, 10 e 11: os comandos print para mandar para o console do Pycharm as informações programadas. Importante atenção ao fato de invocarmos uma variável interna da classe com o 'self.' na frente. Isso indica que estamos fazendo referência a um atributo da classe, e não a uma variável global do programa.

2º - Instanciar cada objeto de acordo com seus dados:

Figura 4 – Corpo principal do programa (main) onde os objetos estão sendo instanciados



```
main.py x
24 print() # pular uma linha
25
26
27 # Inicio do Main
28 # o corpo principal da programa onde começa a ser rodado
29
30 # Instanciando os objetos baseados na classe Funcionario() com os parâmetros
31 func1 = Funcionario(in_nome='Antônio', in_cargo='Gerente', in_salario=5000)
32 func2 = Funcionario(in_nome='Maria', in_cargo='Diretora', in_salario=25000)
33 func3 = Funcionario(in_nome='Pedro', in_cargo='Redator', in_salario=4800)
34 func4 = Funcionario(in_nome='Zilda', in_cargo='Supervisora', in_salario=7500)
35
36 # Chamando o método MostraReg() para cada funcionário 'func' instanciado
37 func1.MostraReg()
38 func2.MostraReg()
39 func3.MostraReg()
40 func4.MostraReg()
41
```

Na Figura 3 vemos quatro objetos sendo instanciados de uma maneira direta, sem inserção de dados vindos de fora. Nesse exemplo podemos ver que



os objetos func1, func2, func3 e func4 estão sendo registrados como objetos baseados na classe Funcionario. Dessa forma, todos eles possuem a mesma estrutura de dados, os atributos da classe e o método MostraReg() que imprime os dados do registro chamado.

Nesse caso, podemos verificar que o método mostra o registro correto porque é chamado pelo objeto, ou seja, o objeto que invoca o método gera a mostra dos dados pertencentes ao objeto que o chamou. Com isso, já podemos perceber uma das grandes vantagens do uso da programação orientada a objeto, que é a abstração de um conjunto de dados em meio aos tantos que existirem.

Abstração é o ato de extrairmos um aspecto ou um assunto que esteja no meio de vários outros ao mesmo tempo. Em uma sinfonia podemos prestar atenção em somente um dos instrumentos, mesmo estando em meio a tantos outros. A programação orientada a objeto tem essa característica: a abstração, pois permite que, facilmente, façamos a segregação de um conjunto de dados em meio a um conjunto que pode ser virtualmente infinito.

1.2 Exercícios

Elaborar códigos para as tarefas solicitadas a seguir usando classe, atributo(s) e método(s) conforme necessário e instanciar objetos de acordo com as solicitações

- a) Fazer um código para cadastrar funcionários com nome, cargo e salário, usando a inserção de dados manual. Inserir os seguintes dados inicialmente e imprimir no console o cadastro inserido para confirmação:

nome	cargo	salario
'Antônio'	'Gerente'	5000
'Maria'	'Diretora'	25000
'Pedro'	'Redator'	4000
'Zilda'	'Supervisora'	7500

Resolução:

O código solicitado pode usar como ponto de partida o código do exemplo acima. A diferença é que os dados deverão ser inseridos



manualmente e deve haver uma confirmação para o usuário poder corrigir um eventual erro. O código passa a ficar da seguinte forma:

Figura 5 – Código da resolução do exercício 1

```
### Início da definição da classe 'Funcionario' #####

# Criação da classe 'Funcionario'
# A classe sempre precisa ser iniciada pela instância de um objeto pois
# sozinha e sem ser invocada em um objeto ela não faz nada !!!!
class Funcionario:
    # definição da função construtora da classe com parâmetros a serem
    # inseridos por ocasião da instância do objeto
    def __init__(self, in_RegID: int, in_nome: str, in_cargo: str, in_salario: int):
        self.RegID: int = in_RegID
        self.nome: str = in_nome
        self.cargo: str = in_cargo
        self.salario: int = in_salario

    def MostraReg(self):
        print() # pular uma linha
        print(f'Registro #{self.RegID}')
        print(f'O nome do funcionário é {self.nome}')
        print(f'O cargo do {self.nome} é {self.cargo}')
        print(f'O salário do {self.cargo} {self.nome} é ${self.salario}')
        print() # pular uma linha

### FIM da definição da classe 'Funcionário' #####

# Início do Main #####
# o corpo principal do programa onde começa a ser rodado

# Rotina para inserção dos dados dos funcionários
registro = ['zero'] # Lista para armazenar instâncias da classe Funcionario
# iniciada com a string zero na posição 0 para não haver registro
# nessa posição

Reg_ID = 0 # contador para gerar os novos objetos

while True:
    opcao = input('\r\nSelecione uma ação: ' +
                  '\r\n1 - Adicionar funcionário' +
                  '\r\n2 - Mostrar os registros existentes' +
                  '\r\n3 - Sair' +
                  '\r\n>>> ')

    if opcao == '1':
        Reg_ID += 1 # prepara o contador de registro para o registro do endereço do
        objeto

        # Colocado while True: aqui para poder manter o programa se
        # o usuário assim desejar. Break se quiser sair
        while True:
            nomeReg = input('Inserir o nome do funcionário: >> ')
            cargoReg = input('Inserir o cargo do funcionário: >> ')

            # esse while True: é para poder fazer voltar para a pergunta do
            # valor do salário, cuja resposta precisa ser numérica, se não for
            # numérica geraria um erro e precisa voltar a perguntar
            while True:
                try:
                    salarioReg = int(input('Inserir o salario do funcionário: >> '))
                    break
                except:
                    print('inserir apenas números')
                    continue

            # instancia o objeto baseado na classe 'Funcionario', passando os parâmetros
            # pelos inputs e o ID de registro RegID. Esse instanciamento retorna o
            # objeto que fica armazenado na lista registro, na posição apontada por
            Reg_ID

            regFunc = Funcionario(in_RegID=Reg_ID, in_nome=nomeReg, in_cargo=cargoReg,
            in_salario=salarioReg)
            registro.append(regFunc)
            print(f'registro {registro}')
```



```
registro[Reg_ID].MostraReg() # mostra o cadastro armazenado
break # encerra o while True para voltar ao menu inicial

elif opcao == '2' :
    # Se a opção for mostrar todos os registros executa a varredura
    # por todos a lista de objetos instanciada e guardados na lista global registro
    # a partir da posição 1 até o tamanho da lista registro menos um
    # (porque a posição inicial zero da lista não deve ser mostrada e não deve ser
    # contada também)
    for i in range(1, len(registro)):
        registro[i].MostraReg()

elif opcao == '3':
    print('Encerrando sistema')
    break # sai do primeiro while True e encerra o programa
else:
    # Se a opção for inválida avisa o usuário e deixa o while True começar novamente
    print('Opção inválida, tente novamente !')

# FIM do Main #####
```

O código mostrado na Figura 5 tem os seguintes blocos, em essência:

- Definição da classe 'Funcionario', sem acento propositalmente;
- Corpo principal Main com a inicialização de duas variáveis globais: registro['zero'] e Reg_ID. A primeira serve para registrar os endereços dos objetos de uma forma que faça sentido para nós humanos. Isso traz o benefício de podermos acessar os objetos depois, em outros pontos do programa, de forma a localizarmos os objetos como se fossem registros numerados em um banco de dados, somente chamando o registro pelo nome e apontando para a posição da lista registro;
- O menu de opções onde o usuário pode selecionar entre adicionar um registro, ver todos os registros e sair do programa;
- A opção 1 traz a possibilidade de colocar um conjunto de informações no registro para onde aponta o contador de registros Reg_ID. Esse será um número único e nunca será repetido. O registro se dá pela instância do objeto baseado na classe 'Funcionario'. Esse objeto pode utilizar o(s) método(s) que estiverem definidos na classe referência;
- A opção 2 traz a possibilidade de o usuário visualizar todos os registros, que são, na verdade, os objetos instanciados anteriormente. O laço for faz a varredura da lista registro de modo que os objetos podem utilizar o método MostraReg(). Quando colocamos o conteúdo da lista registro na posição indicada por i, o endereço do objeto é carregado e o método pode ser executado adequadamente em relação aos dados desse objeto. É assim com todos os objetos registrados na lista registro;
- A opção 3 somente avisa que o programa está sendo encerrado e sai do while True principal, encerrando o programa.



A saída do console para os registros dos quatro funcionários fica da forma mostrada na Figura 6.

Figura 6 – Inserção do primeiro funcionário, pela opção 1

```
Selecione uma ação:
1 - Adicionar funcionário
2 - Mostrar os registros existentes
3 - Sair
>>> 1
Inserir o nome do funcionário: >> Antônio
Inserir o cargo do funcionário: >> Gerente
Inserir o salário do funcionário: >> 5000
registro ['zero', <__main__.Funcionario object at 0x000001E351ED3FD0>]

Registro #1
0 nome do funcionário é Antônio
0 cargo do Antônio é Gerente
0 salário do Gerente Antônio é $5000
```

Figura 7 – Inserção da segunda funcionária, pela opção 1

```
Selecione uma ação:
1 - Adicionar funcionário
2 - Mostrar os registros existentes
3 - Sair
>>> 1
Inserir o nome do funcionário: >> Maria
Inserir o cargo do funcionário: >> Diretora
Inserir o salário do funcionário: >> 25000
registro ['zero', <__main__.Funcionario object at 0x000001E351ED3FD0>, <__main__.Funcionario object at 0x000001E351ED3F18>]

Registro #2
0 nome do funcionário é Maria
0 cargo do Maria é Diretora
0 salário do Diretora Maria é $25000

Selecione uma ação:
1 - Adicionar funcionário
2 - Mostrar os registros existentes
3 - Sair
```

Figura 8 – Inserção do terceiro funcionário, pela opção 1

```
Selecione uma ação:
1 - Adicionar funcionário
2 - Mostrar os registros existentes
3 - Sair
>>> 1
Inserir o nome do funcionário: >> Pedro
Inserir o cargo do funcionário: >> Redator
Inserir o salário do funcionário: >> 4000
registro ['zero', <__main__.Funcionario object at 0x0000028CD4e53FD0>, <__main__.Funcionario object at 0x0000028CD4e53F1D>, <__main__.Funcionario object at 0x0000028CD4e53F18>]

Registro #3
0 nome do funcionário é Pedro
0 cargo do Pedro é Redator
0 salário do Redator Pedro é $4000
```



Figura 9 – Inserção da quarta funcionária, pela opção 1

```
2 - Mostrar os registros existentes
3 - Sair
>>> 1
Inserir o nome do funcionário: >> Zilda
Inserir o cargo do funcionário: >> Supervisora
Inserir o salário do funcionário: >> 7500
registro ['zero', <__main__.Funcionario object at 0x000020CD4653F00>, <__main__.Funcionario object at 0x000020CD4653F10>, <__main__.Funcionario object at 0x000020CD4653F20>]

Registro #4
0 nome do funcionário é Zilda
0 cargo do Zilda é Supervisora
0 salário do Supervisora Zilda é $7500

Selecione uma ação:
1 - Adicionar funcionário
2 - Mostrar os registros existentes
3 - Sair
>>> |
```

Figura 10 – Mostra de todos os registros, da opção 2

```
3 - Sair
>>> 2

Registro #1
0 nome do funcionário é Antônio
0 cargo do Antônio é Gerente
0 salário do Gerente Antônio é $5800

Registro #2
0 nome do funcionário é Maria
0 cargo do Maria é Diretora
0 salário do Diretora Maria é $25800

Registro #3
0 nome do funcionário é Pedro
0 cargo do Pedro é Redator
0 salário do Redator Pedro é $4000

Registro #4
0 nome do funcionário é Zilda
0 cargo do Zilda é Supervisora
0 salário do Supervisora Zilda é $7500
```

- b) Usando programação orientada a objeto, fazer um programa em Python para cadastro de peças de carros com as possibilidades de inserir uma nova peça, visualizar todas as peças registradas, visualizar o registro pelo nome da peça e sair. As peças devem ser inseridas pelo usuário. O registro de peças deve conter as seguintes peças:

Nome da Peça	Fabricante	Preço (R\$)
Homocinética	Nakata	534,00
Pneu	Pirelli	643,00
Pneu	Goodyear	729,00
Buzina	Bibi	97,00



c) Fazer um programa em Python de um Sistema de Registro de Alunos.

Crie uma classe chamada Aluno com os seguintes atributos:

nome: Nome do aluno (string).

idade: Idade do aluno (inteiro).

curso: Curso do aluno (string).

A classe deve ter um método chamado mostrar_info() que imprime as informações do aluno;

A classe deve ter um método que permita alterar conteúdo de atributos do objeto, método setData();

A classe deve ter um método que permita que os conteúdos dos atributos do objeto sejam recebidos por uma variável no corpo principal do programa, método getData();

Precisa ter uma opção que permita ao usuário cadastrar alunos imprimir as informações de todos os alunos cadastrados;

Precisa ter uma opção para alterar dados de um aluno e imprimir todo o cadastro com as novas informações do aluno aparecendo.

d) Conta Bancária: crie uma classe chamada ContaBancaria com os seguintes atributos:

- saldo: saldo atual da conta (float);
- titular: nome do titular da conta (string).

A classe deve ter os seguintes métodos:

- __init__(): o construtor que inicializa os atributos saldo e titular;
- depositar(valor): um método que recebe um valor como parâmetro e adiciona esse valor ao saldo da conta;
- sacar(valor): um método que recebe um valor como parâmetro e subtrai esse valor do saldo da conta, desde que o saldo seja suficiente. Se o saldo não for suficiente, deve exibir uma mensagem informando que o saque não pode ser realizado.

TEMA 2 – COMEÇANDO COM HERANÇA

Quando programamos usando a orientação a objetos podemos utilizar uma base de código para mais do que uma classe. Isso pode acontecer fazendo uma vinculação de uma classe com outra. Em outras palavras, uma



classe herda os atributos e métodos de outra classe. Por exemplo: vamos criar uma classe 'Veiculo', que terá seus atributos e métodos herdados por outra classe, chamada agora de subclasse.

Figura 11 – Exemplo de código usando herança

```
class Veiculo:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

    # Método de exibição da classe Veiculo:
    def exibir_info_veiculo(self):
        return f"Veículo: {self.marca} {self.modelo}"

# Classe Carro
class Carro(Veiculo):
    def __init__(self, marca, modelo, cor):
        Veiculo.__init__(self, marca, modelo)
        self.cor = cor

    # Método de exibição da classe Carro:
    def exibir_info_carro(self):
        return f"Carro: {self.marca} {self.modelo} {self.cor}"

# objeto 'carro' instanciado, herdando as
# as características da classe Veiculo
carro = Carro(marca="Toyota", modelo="Corolla", cor="Azul")

# método de impressão dos atributos que está em Veiculo
# Saída -> Veículo: Toyota Corolla
print(carro.exibir_info_veiculo())

# método de impressão dos atributos que está na classe Carro
# Saída -> Carro: Toyota Corolla Azul
print(carro.exibir_info_carro())
```

No exemplo da Figura 11 temos uma situação em que existe a classe genérica 'Veiculo' (sem acento de propósito) e uma classe mais específica chamada 'Carro'. Todos os carros são veículos. Mas existem outros tipos de veículos, tais como moto, bicicleta, carroça, entre outras possibilidades.

Com a possibilidade de criarmos classes para outros tipos de veículos, torna-se interessante a criação dessa classe abrangente, que tem atributos e métodos que podem ser utilizados por todos os tipos de veículos. Na classe 'Carro' temos a herança de todas as características de 'Veiculo' e mais uma que foi incluída somente nela, que é a cor. Dessa forma, ao instanciar o objeto 'carro', ele terá as características da classe 'Carro' juntamente com as da classe 'Veiculo', todos os atributos e métodos. Podemos ver isso no primeiro



'print' logo depois do instanciamento do objeto 'carro', no qual o método 'exibir_info_veiculo()' é chamado em relação ao objeto 'carro'. Já no segundo 'print' logo em seguida, temos o método 'exibir_info_carro()' sendo chamado, que é um método da classe 'Carro'.

Podemos criar outra classe para herdar as características da classe 'Veiculo'. Vamos incluir a classe 'Moto' no espaço das definições das classes.

Figura 12 – Código com a classe 'Moto' adicionada, que também herda as características da classe pai 'Veiculo'

```
##### DEFINIÇÕES DAS CLASSES #####
class Veiculo:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

    # Método de exibição da classe Veiculo:
    def exibir_info_veiculo(self):
        return f"Veículo: {self.marca} {self.modelo}"

# Classe Carro
class Carro(Veiculo):
    def __init__(self, marca, modelo, cor):
        Veiculo.__init__(self, marca, modelo) # CUIDADO !! Se
        # precisa colocar o 'self' nos parâmetros !!
        self.cor = cor

    # Método de exibição da classe Carro:
    def exibir_info_carro(self):
        return f"Carro: {self.marca} {self.modelo} {self.cor}"

# Classe moto sendo criada herdando as características da classe
# Veiculo
class Moto(Veiculo):
    def __init__(self, marca, modelo, cilindrada):
        super().__init__(marca, modelo) # CUIDADO !! Se usar o
        # NÃO PODE colocar o 'self' nos parâmetros !!
        self.cilindrada = cilindrada

    def exibir_info_moto(self):
        return f"Moto: {self.marca} {self.modelo}, Cilindrada:
        {self.cilindrada}cc"

##### FIM - DEFINIÇÕES DAS CLASSES #####
```




```
# ***** CORPO PRINCIPAL DO PROGRAMA - main
# *****

# objeto 'carro' instanciado, herdando as
# as características da classe Veiculo
carro = Carro(marca="Toyota", modelo="Corolla", cor="Azul")
moto = Moto(marca="Honda", modelo="CBR600RR", cilindrada=600)

# método de impressão dos atributos que está em Veiculo
# chamado pelo objeto 'carro' . Saída -> Veículo: Toyota
Corolla
print(carro.exibir_info_veiculo())

# método de impressão dos atributos que está em Veiculo
# chamado pelo objeto 'moto' . Saída: Veículo: Honda CBR600RR
print(moto.exibir_info_veiculo())

# método de impressão dos atributos que está na classe Carro
# chamado pelo objeto 'carro' . Saída -> Carro: Toyota Corolla
Azul
print(carro.exibir_info_carro())

# método de impressão dos atributos que está na classe Moto
# chamado pelo objeto 'moto' . Saída -> Carro: Toyota Corolla
Azul
print(moto.exibir_info_moto()) # Saída: Moto: Honda CBR600RR,
Cilindrada: 600cc

# ***** FIM - CORPO PRINCIPAL DO PROGRAMA - main
# *****
```

Agora adicionamos uma terceira classe que também herda as características da classe 'Veiculo', que é a classe 'Moto'. Nessa nova classe foi incluído um atributo que existe somente em nossa classe 'Moto', que é a cilindrada. Da mesma forma, incluímos um método nessa nova classe que mostrará todos os atributos do objeto 'moto' instanciado com base na classe 'Moto' no programa principal. Observar que tanto esse método da própria classe 'Moto' quanto o método da classe 'Veículo' estão disponíveis para serem utilizados pelo objeto 'moto'. Outra observação muito importante é que ao criarmos o construtor da subclasse usando o nome da classe pai, precisa usar a palavra self na primeira posição dos parâmetros. Se formos usar o método super(), a palavra reservada não pode estar nos parâmetros. Muita atenção com isso!!

2.1 Exercício

Criar um programa com uma classe pai e duas filhas para uma fábrica de bolos, em que todos os bolos são feitos com uma receita básica usando água, farinha, fermento e açúcar, alterando somente as quantidades. E existe o



bolo de chocolate e o de baunilha, em que cada bolo recebe o ingrediente que o caracteriza e mais alguma cobertura também específica. Os ingredientes terão seus pesos em quilograma ou fração dessa unidade. Imprimir as quantidades dos ingredientes em um método da classe bolo nos dois tipos e um método exclusivo para cada tipo de bolo.

TEMA 3 – COMEÇANDO COM DIAGRAMAÇÃO UML

Uma das principais características dos objetos e as classes em que são baseados é a abstração. A abstração é a ação de separar um aspecto dentre os muitos possíveis e presentes em um cenário. Pode ocorrer de o próprio cenário, o todo, ser uma abstração. Isso significaria que o cenário já é uma separação, um destaque, de alguma realidade maior e mais abrangente.

Por exemplo, podemos observar um determinado cenário do setor de vendas em uma empresa. Pois o todo é a empresa e, por mais que haja uma certa complexidade no departamento comercial, o cenário da empresa sempre é mais complexo que o de um departamento. Mas o departamento em análise possui sua complexidade e pode ter assuntos a serem destacados. Podemos, de saída, verificar que existe os clientes e os pedidos, que são entes diferentes em suas naturezas, mas fazem parte do cenário do departamento comercial. Dessa forma, a construção de qualquer sistema complexo pode ser, também, tão complexa quanto os desenvolvedores consigam ou queiram que seja. Na prática, a melhor coisa é que um sistema computacional seja o mais simples possível para os usuários, mas para isso acontecer, é bastante complicado para os desenvolvedores.

Na medida em que precisamos fazer as devidas abstrações torna-se absolutamente necessário utilizarmos as devidas ferramentas. Claro que os ambientes de desenvolvimento são muito importantes, as linguagens de programação. Mas fazer uma “planta” do que será construído é tão ou mais importante.

Nesse sentido, a ferramenta que ajuda enormemente é o diagrama de classes. Nesse diagrama fazemos as principais abstrações nos diversos níveis e podemos visualizar que relações as classes possuem entre si. Esse tipo de forma de projetar traz à luz os atributos necessários para que as classes, e seus consequentes objetos, executem suas funções de maneira sincronizada e



com limites estabelecidos, ou seja, até que ponto as classes podem fazer seu papel e onde não poderão mais. É o chamado escopo da classe e do projeto.

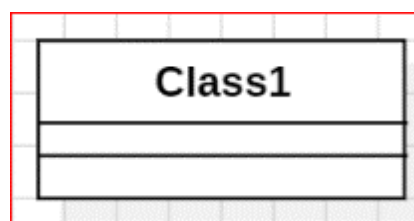
Nos tópicos anteriores, vimos exemplos de classes que foram pensadas e implementadas até de forma simples. Porém, em sistemas mais complexos, isso não é possível ser feito, pois com o aumento da complexidade podemos não visualizar as relações entre os objetos e desses com outras partes do sistema.

Para introduzir as técnicas de execução dos diagramas de classe vamos utilizar os programas que já fizemos anteriormente e os termos que já utilizamos até esse capítulo. E para desenharmos os diagramas vamos utilizar a versão de evaluation do programa StarUML. Essa versão é gratuita e contém os recursos que nesse momento. Existe uma infinidade de softwares para fazer diversos tipos diferentes de diagramas classes UML e outros tipos. Esse programa, nessa versão, será suficiente para a introdução que faremos.

Diagrama UML significa Diagrama de Modelagem de Linguagem Unificada (*Unified Modeling Language*, em inglês). É uma linguagem padronizada para visualizar, especificar, construir e documentar os artefatos de um sistema de software.

No diagrama de classes o símbolo básico é o de classe. Ele tem o aspecto mostrado na Figura 13.

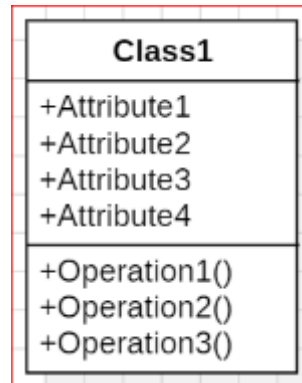
Figura 13 – Aspecto do símbolo de uma classe no StarUML



Esse quadro possui três espaços. O primeiro de cima é onde colocamos o nome da classe. Logo abaixo o espaço recebe os atributos da classe. E, no terceiro espaço, mais abaixo, ficam os métodos. Uma classe com seus atributos e métodos pode ser montada como mostrado na Figura 14.



Figura 14 – Símbolo de classe com o nome, os atributos e métodos em seus espaços



O programa StarUML é construído em inglês e, por isso, os termos aparecem nesse idioma. Na Figura 15 vemos como pode ficar uma classe chamada 'Funcionario' com os atributos Nome, Cargo e Salario (sem acento de propósito) e os métodos Mostra_Infos(), Cria_Reg(), Altera_Reg() e Apaga_Reg().

Figura 15 – Esquema da classe 'Funcionario' (sem acento de propósito)

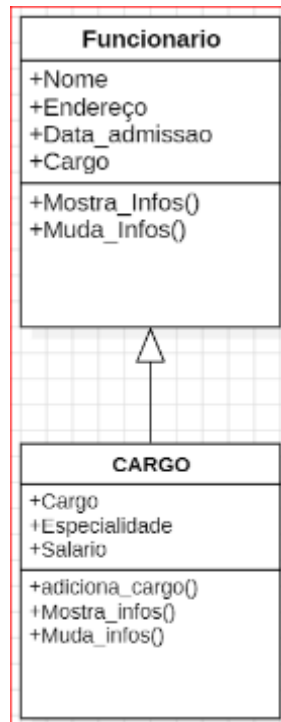


TEMA 4 – AVANÇANDO NA DIAGRAMAÇÃO UML

No diagrama de classes podemos representar uma classe que herda as características de uma classe pai. A classe que recebe os atributos e métodos é chamada subclasse, pois está “abaixo” de alguma outra classe. A classe pai, a que transmite os atributos e métodos, é chamada de superclasse, pois está “acima” na hierarquia de classes. Na Figura 16 vemos uma superclasse e uma subclasse associada.



Figura 16 – Exemplo simples de superclasse e sua subclasse



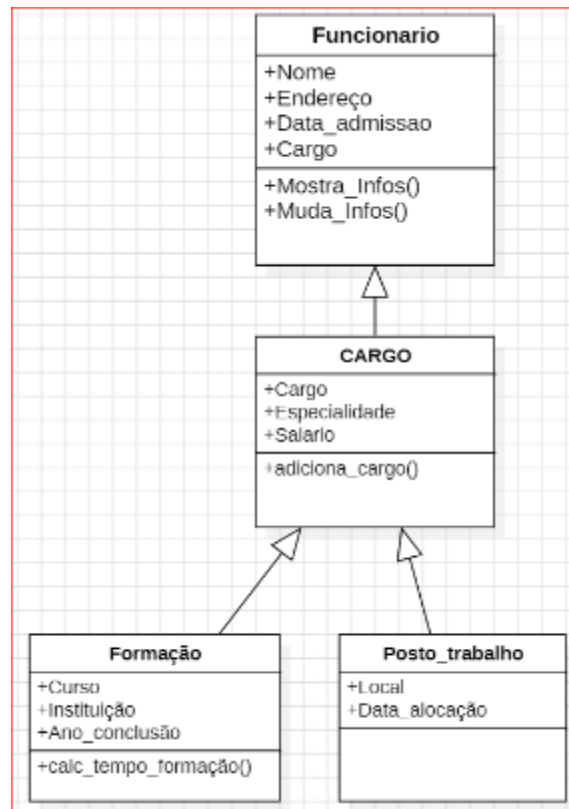
Na Figura 16 vemos a superclasse 'Funcionario' acima e a subclasse Cargo abaixo. Nesse caso, consideramos que uma empresa possui vários funcionários e esses estão alocados nos diversos setores e exercem seus respectivos cargos. Então um gerente de vendas possui esse cargo e teria as informações específicas do cargo colocadas na classe 'Cargo'. Mas esse cargo, de forma mais geral, é um funcionário. Dessa forma, possui todas as informações que todos os funcionários têm. Em termos do diagrama de classes, o gerente de vendas é um cargo definido pela classe 'Cargo' e herda todas as informações e métodos da classe funcionário. Isso pode permitir, por exemplo, que um funcionário mude de cargo, mantendo o mesmo cadastro como funcionário.

Outra forma de herança pode ser uma classe que herda as propriedades de uma outra classe que já é herdeira de uma primeira classe. Na Figura 17 temos uma possibilidade de como podemos constituir classes netas, que herdaram atributos e métodos de uma classe que já está recebendo essas características de outra. Nesse exemplo podemos instanciar um objeto baseado na classe 'Formação' e ele terá todas as características da superclasse 'Cargo' e da super-classe 'Funcionário'. Por exemplo, vamos instanciar um objeto chamado 'objeto' baseado em 'Cargo'. Em Python faríamos isso simplesmente da seguinte forma: `objeto = Cargo()`. Isso significa que poderá



ser requisitado a executar no código funcionalidades de um método do tipo objeto.Mostra_Infos(). Notar que o método Mostra_Infos() é da super-classe 'Funcionario', mas está disponível para o 'objeto'.

Figura 17 – Exemplo de subclasses herdando propriedades de uma classe que já é herdeira de outra classe



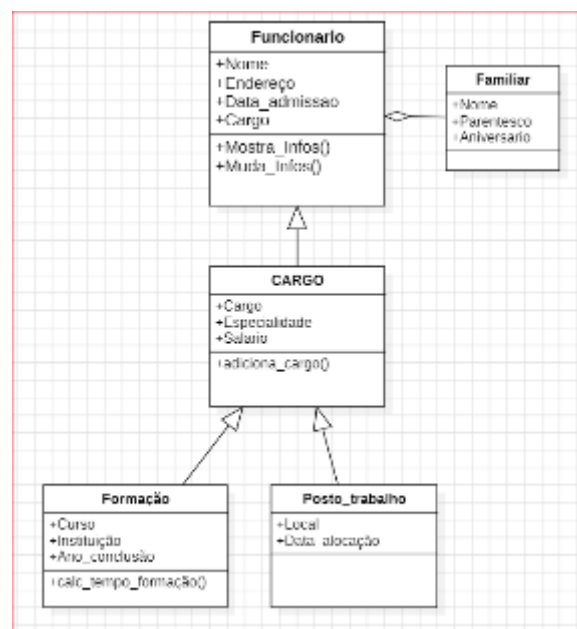
Outra possibilidade na composição de programas baseados no paradigma de orientação a objetos é a agregação. Essa técnica consiste em criar um conjunto de atributos que é associado a uma classe, como se fossem diretamente dessa. Na Figura 18 podemos ver um exemplo de uma agregação entre classes. A classe 'Familiar' é agregada à classe 'Funcionario'. Dessa forma, a classe 'Funcionario' possui todas as características da classe 'Familiar' e as transmite por herança para todas as subclasses subsequentes da estrutura. Mas qual seria a diferença entre uma agregação e uma herança? A diferença é que na herança as classes possuem um relacionamento lógico. Praticamente são uma classe que é desmembrada em subclasses para atender alguma finalidade na estrutura do sistema em construção. Por outro lado, a classe agregada existe por si, ou seja, ela não é uma parte da estrutura da qual está sendo agregada, ela pertence a outra estrutura. Fazemos a agregação



para poder interligar duas ou mais estruturas que são, em termos lógicos, independentes.

No exemplo da Figura 18, temos uma estrutura que define entidades de informação e ações relativas aos funcionários da empresa, aprofundando em cada subclasse os atributos e métodos necessários para o trabalho em relação a cada funcionário da empresa e qual ligação ele possui com essa. A classe 'Familiar' diz respeito a um aspecto que não tem uma ligação direta com a empresa, somente com o funcionário, é um conjunto de informações da vida da pessoa que está trabalhando aqui na empresa. Essa estrutura lógica, 'Familiar', não traz nenhum tipo de informação para a relação do funcionário com a empresa. Mas é uma estrutura necessária para algum setor, como o de marketing, por exemplo, para mandar congratulações no aniversário do familiar do funcionário ou estabelecer as extensões de benefícios aos familiares, tais como plano de saúde, entre outras possibilidades. Então o assunto 'Familiar' não faz parte direta da relação do funcionário com a empresa, mas existe em algum setor da empresa e tem sua própria estrutura independente, que pode ser agregada à estrutura do funcionário.

Figura 18 – Exemplo de agregação de classe



Uma classe agregada pode ser instanciada em um objeto. No entanto, a classe agregada não está diretamente vinculada à classe agregadora em termos de ciclo de vida. Isso significa que a agregação permite que as instâncias da classe agregada existam independentemente das instâncias da



classe agregadora. As instâncias da classe agregada podem ser criadas, acessadas e manipuladas por objetos da classe agregadora ou por outros objetos, dependendo da visibilidade e dos relacionamentos definidos em seu desenho.

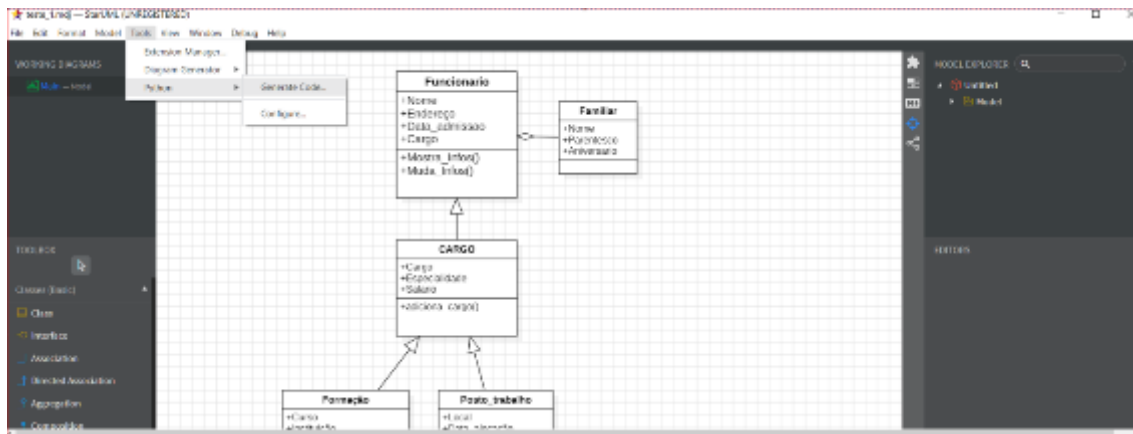
A utilização de agregação ou de herança é uma decisão de projeto e permite tornar um sistema complexo em algo superorganizado e de fácil manipulação. Outro fato extremamente importante é que a maior característica da programação orientada a objetos é a abstração. Abstração é o ato de separar um assunto de uma realidade complexa, de um cenário que possui vários temas que podem receber uma atenção de forma isolada. Podemos perceber cada instrumento em uma orquestra sinfônica. A orquestra está tocando a música de forma completa, com todos seus instrumentos componentes, mas podemos prestar atenção somente nos violinos, por exemplo, ou nos tambores. Podemos prestar atenção aos sabores de uma bebida, aos aromas ou a alguma outra característica. Ao fazermos isso, estamos selecionando algum dos muitos aspectos que compõem uma determinada realidade, o que é chamado de abstração.

TEMA 5 – GERAÇÃO DE CÓDIGO COM A DIAGRAMAÇÃO UML

Nos programas de diagramação em UML da atualidade, uma funcionalidade muito interessante é a de converter um diagrama UML diretamente para um código de programação conforme configuração prévia. Para utilização dessa ferramenta é necessário a instalação, dentro do StarUML, da extensão Python. Deve-se clicar na função ‘tools’ e em ‘extensions’. Na caixa de diálogo que abre, deve-se procurar pela extensão ‘Python’ e clicar para instalar. O programa vai baixar e instalar a extensão automaticamente e pedir para recarregar o StarUML, que deve ser executado. Na caixa de diálogo pode-se ver o manual de instruções da extensão Python para StarUML. Uma vez reaberto o StarUML, abrimos o arquivo que estamos trabalhando e queremos que seja convertido para o código em Python. Vamos em ‘tools’ novamente, mas agora encontramos ali a opção ‘Python’. Nessa opção clicamos em ‘Generate Code’. A Figura 19 mostra o detalhe do comando para geração de código a partir do diagrama de classes do projeto.



Figura 19 – Geração do código a partir do diagrama de classes



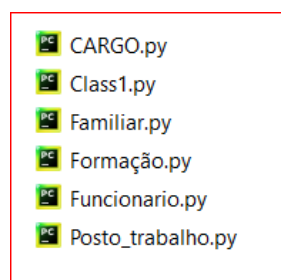
Ao clicar para gerar o código o StarUML solicita que seja atribuído um modelo para esse projeto, selecionamos 'model'.

Figura 20 – Seleção do modelo de geração de código Python no StarUML



Em seguida o StarUML solicita a indicação de qual pasta deverá receber os arquivos gerados. Essa pasta deve estar criada previamente, antes da geração dos arquivos com os códigos em Python.

Figura 21 – Arquivos Python gerados a partir do diagrama de classes no StarUML



Esses arquivos gerados servem como base para o desenvolvimento do programa como um todo. Porém, é necessário que sejam feitas correções e



acréscimos nas classes criadas. De forma geral, é necessário corrigir os construtores das classes, ou seja, as funções `__init__` conforme cada caso.

FINALIZANDO

Nesta etapa, percorremos os conceitos relativos à programação orientada a objetos com abrangência em códigos da linguagem Python e o planejamento em diagramas UML, utilizando o software StarUML, ainda que de uma forma superficial, mas o suficiente para você poder iniciar um projeto com bom grau de planejamento do sistema.

Verificamos como podemos criar códigos em Python dentro do paradigma da orientação a objeto. Para isso, mostramos como as classes podem ser criadas, juntamente com seus atributos e métodos.

Indo além, fizemos uso dessas características e mostramos como os objetos podem ser instanciados com base nessas classes. Mais ainda, mostramos como uma subclasse ou classe filha pôde ser criada, manipulada e instanciada de forma que herdasse de uma super-classe ou classe pai todas as suas características e funcionalidades. Experimentamos como utilizar os métodos da própria subclasse e da super-classe mesmo em um objeto instanciado na classe filha. Ainda assim, o objeto podia acionar os métodos da classe pai.

Na sequência, entramos na ideia do planejamento do sistema utilizando a programação orientada a objeto. Esse planejamento pode mostrar um mapa que mostra como as classes são compostas e como elas se relacionam. Utilizamos o ambiente de desenvolvimento StarUML de modo a enxergarmos um verdadeiro mapa contendo todos os elementos associados ao projeto. Dessa forma, temos uma maior clareza sobre quais classes devem ser criadas, como essas classes compõem a estrutura do sistema e quais recursos podem ser adicionados. Tendo isso em mente, podemos adicionar, remover ou realocar aspectos importantes do sistema como um todo e, também, evitar redundâncias desnecessárias, tornando o sistema mais organizado e enxuto. Isso contribui fortemente para uma otimização nos usos de recursos, bem como no desempenho final do sistema como um todo. Essa otimização é uma característica mandatória em programas que exigem maior desempenho, tais como os jogos, por exemplo.



Por fim, tivemos uma abordagem introdutória sobre como o software StarUML permite converter diretamente o diagrama de classes para um primeiro esquema de códigos Python, criando na linguagem um conjunto com as classes dimensionadas no diagrama de classes. Com isso, estamos preparados para os próximos passos da disciplina, nos quais aplicaremos esses aspectos da programação, de modo a podemos criar programas bem planejados e bem estruturados.

Códigos dos exercícios

Tema 1. b)

```
##### inicio - Classe Peca #####
class Peca:
    def __init__(self, ID: int, nome: str, fab: str, preco: int):
        self.ID: int = ID
        self.nome: str = nome
        self.fab: str = fab
        self.preco: int = preco

    def MostraRegistro(self):
        print() # pular uma linha
        print(f'Registro #{self.ID}')
        print(f'Peça: {self.nome}')
        print(f'Fabricante: {self.fab}')
        print(f'Preço: ${self.preco}')

    # Método que retorna os valores das variáveis internas da classe
    def GetRegistro(self):
        return {'ID': self.ID, 'nome': self.nome, 'fab': self.fab, 'preço': self.preco}

##### FIM - Classe Peca #####

# Inicio do Main #####

### inicio main #####
registro = ['zero']
Reg_ID = 0

while True:
    print('Selecione uma ação: ')
    print('1 - Adicionar Peça')
    print('2 - Mostrar todas as peças')
    print('3 - Mostrar peça por nome')
    print('4 - Sair')
    opcao = input('>>> ')
    if opcao == '1':

        Reg_ID += 1
        while True:
            nomeReg = input('Inserir o nome da peça >> ')
            fabReg = input('Inserir o fabricante >> ')

            while True:
                try:
                    precoReg = int(input('Inserir o preço >> '))
                    break
                except:
                    print('inserir apenas números !')
                    continue

            regPeca = Peca(ID=Reg_ID, nome=nomeReg, fab=fabReg, preco=precoReg)
            registro.append(regPeca)
            registro[Reg_ID].MostraRegistro()
            break

    elif opcao == '2':

        for i in range(1, len(registro)):
            registro[i].MostraRegistro()

    elif opcao == '3':
        nome_peca = input("Digite o nome da peça que deseja visualizar: ")
        encontrada = False
        for i in range(1, len(registro)):
            regGot = registro[i].GetRegistro()
            # print(regGot) # printa a variável que recebe a resposta do
            # método GetRegistro() para debug
            if regGot['nome'].lower() == nome_peca.lower():
                print("\nRegistro da peça:")
                registro[i].MostraRegistro()
                encontrada = True

        if not encontrada:
            print(f"\nA peça '{nome_peca}' não foi encontrada.")
```



```
elif opcao == '4':
    print('Saindo...')
    break
else:
    print('Opção inválida, tente novamente !')

# FIM do Main #####
```

Tema 2. c)

```
class Aluno:
    def __init__(self, nome, idade, curso):
        self.nome = nome
        self.idade = idade
        self.curso = curso

    def mostrar_info(self):
        print(f"Nome: {self.nome}, Idade: {self.idade}, Curso: {self.curso}")

    def setData(self, nome, idade, curso):
        self.nome = nome
        self.idade = idade
        self.curso = curso

    def getData(self):
        return [self.nome, self.idade, self.curso]

def cadastrar_aluno(alt=""):
    if alt == "":
        nome = input("Digite o nome do aluno: ")
        while True:
            try:
                idade = int(input("Digite a idade do aluno: "))
                break
            except:
                print('Somente números !!')
                continue
        curso = input("Digite o curso do aluno: ")
        return Aluno(nome, idade, curso)

    else:
        print(f'Alterando as informações do aluno {alt}')
        while True:
            try:
                idade = int(input("Digite a idade do aluno: "))
                break
            except:
                print('Somente números !!')
                continue
        curso = input("Digite o curso do aluno: ")
        return [alt, idade, curso]

alunos = []

while True:
    opcao = input("\n Selecione uma ação: +
        \n 1 - Cadastrar aluno" +
        \n 2 - Mostrar registros de alunos" +
        \n 3 - Alterar um registro de aluno" +
        \n 4 - Sair" +
        \n >>> ")

    if opcao == '1':
        aluno = cadastrar_aluno()
        alunos.append(aluno)
        print("Aluno cadastrado com sucesso!")

    elif opcao == '2':
        print("\nRegistros de alunos:")
        for aluno in alunos:
            aluno.mostrar_info()

    elif opcao == '3':
        altReg = input("Qual aluno deseja alterar informações ? >> ")

        encontrada = False
        ctAluno = 0
        for aluno in alunos:

            regGot = aluno.getData()

            print(regGot) # printa a variável que recebe a resposta do
                          # método GetRegistro() para debug

            if regGot[0].lower() == altReg.lower():
                print(f'\nAlterar informações do aluno: {altReg}')
                infoAluno = cadastrar_aluno(altReg)
                aluno.setData(infoAluno[0], infoAluno[1], infoAluno[2])

                print("\n Informações alteradas com sucesso!")

                aluno.mostrar_info()
                encontrada = True
                break

        if not encontrada:
            print(f'\nAluno {altReg} não encontrado !')
```



```
elif opcao == '4':
    print("Encerrando o programa...")
    break

else:
    print("Opção inválida. Tente novamente.")
```

Tema 3. d)

```
class ContaBancaria:
    def __init__(self, titular, saldo=0.0):
        self.titular = titular
        self.saldo = saldo

    def depositar(self, valor):
        self.saldo += valor
        print(f"Depósito de R${valor} realizado. Saldo atual: R${self.saldo:.2f}")

    def sacar(self, valor):
        if valor <= self.saldo:
            self.saldo -= valor
            print(f"Saque de R${valor} realizado. Saldo atual: R${self.saldo:.2f}")
        else:
            print("Saldo insuficiente. Saque não pode ser realizado.")

    def mostrar_saldo(self):
        print(f"Saldo atual da conta de {self.titular}: R${self.saldo:.2f}")

# Criação de uma conta bancária
conta = ContaBancaria("João")

# Realização de operações na conta
conta.mostrar_saldo()
conta.depositar(1000.0)
conta.sacar(500.0)
conta.mostrar_saldo()
conta.sacar(700.0)
```

Tema 4. a)

```
class Bolo:
    def __init__(self, agua, farinha, fermento, acucar):
        self.agua: float = agua
        self.farinha: float = farinha
        self.fermento: float = fermento
        self.acucar: float = acucar

    def show_bolo(self):
        return (self.agua, self.farinha, self.fermento, self.acucar)

class Bolo Chocolate(Bolo):
    def __init__(self, agua, farinha, fermento, acucar, chocolate):
        Bolo.__init__(self, agua, farinha, fermento, acucar)
        self.chocolate: float = chocolate

    def show_bolo_choco(self):
        return (self.agua, self.farinha, self.fermento, self.acucar, self.chocolate)

class Bolo Baunilha(Bolo):
    def __init__(self, agua, farinha, fermento, acucar, baunilha):
        Bolo.__init__(self, agua, farinha, fermento, acucar)
        self.baunilha: float = baunilha

    def show_bolo_bauni(self):
        return (self.agua, self.farinha, self.fermento, self.acucar, self.baunilha)

# main
bolo Choco = Bolo Chocolate(agua=1, farinha=0.5, fermento=0.1, acucar=0.3, chocolate=0.5)
bolo Bauni = Bolo Baunilha(agua=0.8, farinha=0.7, fermento=0.2, acucar=0.6, baunilha=0.9)

print(bolo Choco.show_bolo())
print(bolo Choco.show_bolo_choco())

print(bolo Bauni.show_bolo())
print(bolo Bauni.show_bolo_bauni())
```



REFERÊNCIAS

ROSEWOOD, E. **Programação Orientada a Objetos em Python**: dos fundamentos às técnicas avançadas. [edição independente]. 2023.

SARAIVA JUNIOR, O. **Introdução à Orientação a Objetos com C++ e Python**: uma abordagem prática. São Paulo: Novatec, 2017.

SILVA, F. M.; LEITE, M. C. D.; OLIVEIRA, D. B. **Paradigmas de programação**. Porto Alegre: Grupo A, 2019.