Semana 18 - Aula 1

Tópico Principal da Aula: Pilares da Programação Orientada a Objetos

Subtítulo/Tema Específico: Aplicação do encapsulamento em um projeto prático

Código da aula: [SIS]ANO1C3B3S18A1

Objetivos da Aula:

- Compreender a implementação do encapsulamento em um projeto prático.
- Conhecer frameworks de desenvolvimento ágil e tecnologias de CI/CD.
- Praticar a curiosidade e a resiliência de sistemas computacionais.

Recursos Adicionais:

- Caderno para anotações;
- Acesso ao laboratório de informática e/ou internet.

Exposição do Conteúdo:

Referência do Slide: 10 - Construindo conceito - Encapsulamento

- **Definição:** O encapsulamento é um dos pilares da Programação Orientada a Objetos (POO). Sua ideia central é ocultar os detalhes de implementação de um objeto, controlando o acesso aos seus atributos e métodos. Em vez de permitir o acesso direto, o objeto expõe uma interface pública que outras partes do sistema podem usar para interagir com ele.
- **Aprofundamento/Complemento:** Em Python, o encapsulamento é implementado por meio de convenções de nomenclatura para atributos.
    - **Atributos Públicos:** Acessíveis de qualquer lugar (ex: `self.nome`).
    - **Atributos Protegidos:** Indicados com um underscore (_), sinalizam que não devem ser acessados diretamente fora da classe ou de suas subclasses (ex: `self._nome`). É um aviso para outros desenvolvedores.
    - **Atributos Privados:** Indicados com dois underscores (__), o Python modifica o nome do atributo internamente (um processo chamado *name mangling*) para dificultar o acesso direto fora da classe (ex: `self.__nome`).
- **Exemplo Prático:**
- Python

```python
class ContaBancaria:
    def __init__(self, saldo_inicial):
        # Atributo privado para proteger o saldo
        self.__saldo = saldo_inicial

    # Método público para depositar
    def depositar(self, valor):
        if valor > 0:
```

```
            self.__saldo += valor
            print(f"Depósito de R${valor} realizado com sucesso.")

    # Método público para sacar
    def sacar(self, valor):
        if 0 < valor <= self.__saldo:
            self.__saldo -= valor
            print(f"Saque de R${valor} realizado com sucesso.")
        else:
            print("Saldo insuficiente ou valor inválido.")

    # Método público para consultar o saldo (Getter)
    def get_saldo(self):
        return self.__saldo

# Uso da classe
minha_conta = ContaBancaria(1000)
print(f"Saldo inicial: R${minha_conta.get_saldo()}") # Acesso correto via método público

minha_conta.depositar(500)
minha_conta.sacar(200)

# Tentativa de acesso direto (não funciona como esperado e não é recomendado)
# print(minha_conta.__saldo) # Isso resultará em um AttributeError

print(f"Saldo final: R${minha_conta.get_saldo()}")
```

Referência do Slide: 11 - Construindo o conceito - Objetivos do encapsulamento

- **Definição:** O encapsulamento tem três objetivos principais:
    - **Proteger os dados:** Evita que os dados de um objeto sejam modificados de forma acidental ou maliciosa, garantindo a integridade do estado do objeto.
    - **Reduzir a complexidade:** Ao esconder os detalhes internos, o objeto se torna mais fácil de usar. O desenvolvedor que utiliza a classe não precisa saber *como* ela funciona por dentro, apenas *o que* ela faz através de sua interface pública.
    - **Facilitar a manutenção:** Como a implementação está "escondida", ela pode ser alterada ou melhorada sem quebrar o código que utiliza a classe, desde que a interface pública (os métodos públicos) permaneça a mesma.
- **Aprofundamento/Complemento:** A combinação desses objetivos torna o software mais robusto, flexível e escalável. Quando se protege os dados com encapsulamento, a classe se torna a única responsável por gerenciar seu próprio estado, um princípio fundamental para um bom design de software.
- **Exemplo Prático:** No exemplo da `ContaBancaria` acima, a proteção do atributo `__saldo` impede que um código externo faça `minha_conta.saldo = -500`, o que deixaria o objeto em um estado inválido. A modificação do saldo é controlada pelos métodos `depositar` e `sacar`, que contêm a lógica de validação.

Semana 18 - Aula 2

Tópico Principal da Aula: Pilares da Programação Orientada a Objetos

Subtítulo/Tema Específico: Aplicação do encapsulamento em um projeto prático

Código da aula: [SIS]ANO1C3B3S18A2

Objetivos da Aula:

- Compreender a implementação do encapsulamento em um projeto prático.
- Implementar métodos getters e setters para controlar o acesso a atributos.
- Desenvolver código de forma colaborativa.

Recursos Adicionais:

- Caderno para anotações;
- Acesso ao laboratório de informática e à internet;
- Visual Studio Code ou outra IDE Python.

Exposição do Conteúdo:

Referência do Slide: 07 - Colocando em prática - Getters e Setters

- **Definição:** Métodos *Getters* e *Setters* são a implementação prática do encapsulamento para o acesso a atributos.
    - Getters: Métodos públicos usados para *obter* ou *ler* o valor de um atributo privado. Por convenção, seus nomes geralmente começam com `get_`.
    - Setters: Métodos públicos usados para *definir* ou *modificar* o valor de um atributo privado. Geralmente começam com `set_`. Eles são essenciais para incluir lógica de validação antes de alterar o valor de um atributo.
    - Aprendemos que métodos getters e setters são fornecidos para acessar e modificar os valores dos atributos de forma controlada.
- **Aprofundamento/Complemento:** Embora seja possível criar métodos como `get_nome()` e `set_nome()`, Python oferece uma forma mais "pythônica" de fazer isso usando `property`. A `property` permite que você trate um método como se fosse um atributo público, mas executando o código do getter, setter e deleter por trás dos panos.
- **Exemplo Prático:** O slide propõe a criação da classe `Carro` e a implementação de getters e setters. Abaixo está o código completo, como solicitado na atividade.
- Python

```
class Carro:
    def __init__(self, marca, modelo, ano, cor, preco):
        self.__marca = marca
```

```
        self.__modelo = modelo
        self.__ano = ano
        self.__cor = cor
        self.__preco = preco

    # --- Getters ---
    def get_marca(self):
        return self.__marca

    def get_modelo(self):
        return self.__modelo

    def get_ano(self):
        return self.__ano

    def get_cor(self):
        return self.__cor

    def get_preco(self):
        return self.__preco

    # --- Setters ---
    def set_marca(self, nova_marca):
        self.__marca = nova_marca

    def set_modelo(self, novo_modelo):
        self.__modelo = novo_modelo

    def set_ano(self, novo_ano):
        if novo_ano > 1886: # O primeiro carro foi inventado em 1886
            self.__ano = novo_ano
        else:
            print("Ano inválido.")

    def set_cor(self, nova_cor):
        self.__cor = nova_cor

    def set_preco(self, novo_preco):
        if novo_preco > 0:
            self.__preco = novo_preco
        else:
            print("O preço deve ser um valor positivo.")

# Testes e resultados
meu_carro = Carro("Ford", "Ka", 2020, "Branco", 45000)
print(f"Marca: {meu_carro.get_marca()}")
print(f"Preço: R${meu_carro.get_preco()}")

print("\n--- Alterando o preço ---")
meu_carro.set_preco(-500) # Tentativa inválida
```

```
meu_carro.set_preco(48000) # Alteração válida
print(f"Novo Preço: R${meu_carro.get_preco()}")
```

Semana 18 - Aula 3

Tópico Principal da Aula: Pilares da Programação Orientada a Objetos

Subtítulo/Tema Específico: Aplicação do encapsulamento em um projeto prático

Código da aula: [SIS]ANO1C3B3S18A3

Objetivos da Aula:

- Compreender a implementação do encapsulamento em um projeto prático.
- Aplicar o encapsulamento para criar uma classe de controle de estoque.
- Discutir as implicações éticas relacionadas à segurança e privacidade de dados em sistemas.

Recursos Adicionais:

- Caderno para anotações;
- Acesso ao laboratório de informática e à internet;
- Visual Studio Code ou outra IDE Python.

Exposição do Conteúdo:

Referência do Slide: 09 - Colocando em prática - Sistema com Estoque

- **Definição:** A atividade propõe estender o sistema de carros da aula anterior criando uma nova classe, `Estoque`. Essa classe será responsável por gerenciar uma coleção de objetos da classe
- `Carro`. O atributo que armazena os carros dentro do estoque (`self.__carros`) será privado, e métodos públicos como `adicionar_carro` serão usados para manipulá-lo de forma controlada.
- **Aprofundamento/Complemento:** Essa abordagem, onde uma classe (`Estoque`) é composta por objetos de outra classe (`Carro`), é chamada de **Composição**. É uma maneira poderosa de modelar relacionamentos do tipo "tem-um" (um estoque *tem-um* ou mais carros). O encapsulamento aqui é duplo: a classe `Carro` encapsula os dados de um carro, e a classe `Estoque` encapsula a coleção de carros e as regras para gerenciá-la.
- **Exemplo Prático:** A seguir, o código completo da classe `Estoque`, que interage com a classe `Carro` da aula anterior, e adiciona métodos para listar e buscar carros.
- Python

```
# Reutilizando a classe Carro da Aula 2
class Carro:
    # ... (código completo da classe Carro da aula anterior) ...
```

```
    def __init__(self, marca, modelo, ano, cor, preco):
        self.__marca = marca
        self.__modelo = modelo
        self.__ano = ano
        self.__cor = cor
        self.__preco = preco

    def get_marca(self): return self.__marca
    def get_modelo(self): return self.__modelo
    def __str__(self): # Método para facilitar a impressão do objeto
                return f"{self.get_marca()} {self.get_modelo()} ({self.__ano}) - Cor: {self.__cor} -
R${self.__preco}"


class Estoque:
    def __init__(self):
        # O atributo __carros é privado para proteger a lista
        self.__carros = [] # Inicializa a lista de carros vazia [cite: 392, 393]

    def adicionar_carro(self, carro):
        # Adiciona um objeto Carro à lista
        self.__carros.append(carro) [cite: 395]
        print(f"Carro {carro.get_marca()} {carro.get_modelo()} adicionado ao estoque.") [cite: 396]

    def listar_carros(self):
        if not self.__carros:
            print("O estoque está vazio.")
            return

        print("\n--- Carros em Estoque ---")
        for carro in self.__carros:
            print(carro)

# --- Testes e Resultados ---
# 1. Criar o estoque
estoque_loja = Estoque()

# 2. Criar instâncias de Carro
carro1 = Carro("Toyota", "Corolla", 2022, "Prata", 120000)
carro2 = Carro("Honda", "Civic", 2023, "Preto", 135000)
carro3 = Carro("Ford", "Mustang", 2024, "Vermelho", 450000)


# 3. Adicionar carros ao estoque
estoque_loja.adicionar_carro(carro1)
estoque_loja.adicionar_carro(carro2)
estoque_loja.adicionar_carro(carro3)

# 4. Listar os carros do estoque
estoque_loja.listar_carros()
```

Referência do Slide: 10 - Ser sempre + (Questões Éticas)

- **Definição:** Ao desenvolver sistemas que manipulam dados, especialmente dados de clientes, surgem importantes questões éticas. Os principais pontos de preocupação são:
    - **Privacidade dos Dados:** O sistema deve proteger as informações pessoais dos clientes (nome, endereço, etc.) contra acessos não autorizados.
    - **Segurança da Informação:** O sistema precisa de defesas robustas contra ameaças cibernéticas, como ataques de hackers, para evitar vazamentos de dados e roubo de identidade.
    - **Transparência no Uso dos Dados:** Os clientes devem ser claramente informados sobre como seus dados serão coletados e utilizados, e devem dar consentimento explícito para esse uso.
- **Aprofundamento/Complemento:** No Brasil, a **Lei Geral de Proteção de Dados (LGPD - Lei nº 13.709/2018)** regulamenta o tratamento de dados pessoais. Desenvolver um sistema em conformidade com a LGPD não é apenas uma questão ética, mas uma obrigação legal. Isso inclui ter bases legais para o tratamento de dados, garantir os direitos dos titulares (como acesso e exclusão de seus dados) e implementar medidas de segurança.
- **Exemplo Prático:** No sistema de carros, se a classe `Carro` fosse associada a um `Proprietario` com nome, CPF e endereço, seria crucial que esses dados fossem armazenados de forma criptografada no banco de dados. Além disso, o acesso a esses dados dentro do sistema deveria ser restrito apenas a funcionários autorizados, com registros (logs) de quem acessou e quando.