ES-SOLID

Aplicação de principios SOLID

S — Responsabilidade Única (Single Responsibility Principle)

A responsabilidade única diz respeito ao nosso trecho de código (classe, função, etc.) realizar apenas uma coisa de forma bem feita. Ou seja, como o próprio nome diz, ter uma única responsabilidade. Ao fazer essa separação, o código fica mais legível, evita bugs e, caso ocorram, será bem mais fácil de controlá-los

Exemplos sem SOLID

```
class Aluno():
    def __init__(self,id: int = 0, nome: str = '', idade: int = 0, curso: str = '' ):
        self.id = id
        self.nome = nome
        self.idade = idade
        self.curso = curso
        self.lerAlunoTerminal = LerAlunoTerminal()

    def criarAluno(self, alunos):
        #Logica

    def editarAluno(self,id: int, alunos):
        #Logica
    def excluirAluno(self,id: int, alunos):
        #Logica

    def verAlunos(self,alunos):
        #Logica
```

Nossa classe Aluno tem métodos de serviço do aluno que não são de sua responsabilidade. Então, a classe Aluno agora fica responsável apenas por instanciar o aluno, e a classe AlunoService faz as funções básicas do CRUD. Agora, cada função tem sua responsabilidade única.

Exemplos com SOLID

```
class Aluno():
    def __init__(self,id: int = 0, nome: str = '', idade: int = 0, curso: str = '' ):
        self.id = id
        self.nome = nome
        self.idade = idade
        self.curso = curso
```

```
class AlunoServicos:
    def __init__(self):
        self.lerAlunoTerminal = LerAlunoTerminal()

    def criarAluno(self, alunos):
        #Logica

    def editarAluno(self, id: int, alunos):
        #Logica

    def excluirAluno(self, id: int, alunos):
        #Logica

    def verAlunos(self, alunos):
        #Logica
```

O — Aberto-Fechado (Open/Closed Principle)

O princípio aberto/fechado diz o seguinte: 'As classes devem ser abertas para extensão, mas fechadas para modificação'. Ou seja, classes que já estão funcionando não devem ser modificadas. Caso seja necessário adicionar algo novo, deve-se realizar uma extensão.

Técnicas para fazer isso:

1. Herdar classes

2. Uso de interfaces ou abstrações

```
class ChecarPagamento:
    def processar(self,tipo):
        if tipo == 'credito':
            pass
        elif tipo == 'debito':
            pass
```

Para cada novo tipo de pagamento temos que alterar a função

```
class Pagamento(ABC):
    @abstractmethod
    def processar(self):
        pass


class Credito(Pagamento):
    def processar(self):
        pass

class Debito(Pagamento):
    def processar(self):
        pass

class ChecarPagamento:
    def __init__(self, tipoPagamento: Pagamento):
        self.tp = tipoPagamento
    def processarPagamento(self):
        self.tp.processar()
```

Dessa forma, para cada tipo de pagamento diferente, posso apenas adicionar uma nova interface/classe, e não será necessário mexer no checarPagamento

Substituição de Liskov

Princípio da substituição de Liskov — Uma classe derivada deve ser substituível por sua classe base. Ou seja, no nosso código, a classe AlunoServicos deriva da classe CRUD e implementa todos os métodos da mesma. Caso a classe filha seja passada no lugar da classe pai, o código funcionará normalmente, como esperado.

Classe Base

```
class Crud(ABC):
    def __init__(self):
        self.data = []
    @abstractmethod
    def criar(self):
        pass

    @abstractmethod
    def editar(self,id):
        pass

    @abstractmethod
    def ver(self):
        pass

    @abstractmethod
    def excluir(self,id):
        pass
```

Classe derivada

```
class AlunoServicos(Crud):
    def __init__(self):
        super().__init__()
        #Logica


    def criar(self):
         #Logica

    def ver(self):
         #Logica

    def editar(self, id):
         #Logica

    def excluir(self,id: int):
         #Logica
```

ISP — Interface Segregation Principle:

O Princípio da Segregação da Interface diz que uma classe não deve ser forçada a implementar interfaces e métodos que não irá utilizar.

```
class Pessoa(ABC):
    @abstractmethod
    def estudar(self):
        pass

    @abstractmethod
    def ensinar(self):
        pass

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

Pensando na classe Pessoa, quando o Aluno for herdar, ele vai implementar métodos que não fazem sentido para ele, como o método avaliar. Já o Professor pode implementar todos os métodos.

```
class Aluno(Pessoa):
    def __init__(self,id: int = 0, nome: str = '', idade: int = 0, curso: str = '' ):
        #Logica
    def estudar(self):
        print("Estudando!")

    def ensinar(self):
        raise NotImplementedError("Aluno não pode ensinar")

    def avaliar(self):
        raise NotImplementedError("Aluno não pode avaliar")
```

```
class Professor(Pessoa):
    def __init__(self,matricula: int = 0 ,  nome: str = '', idade: int = 0, cargaHoraria: int = 0, salario: float = 0 ):
        #Logica

    def estudar(self):
        print("Estudando conteúdo novo!")

    def ensinar(self):
        print("Ensinando alunos!")

    def avaliar(self):
        print("Avaliando aluno!")
```

Resolvendo

Para resolvermos isso, basta criar uma classe Estudante com o método estudar, e a classe Aluno herdar esse método. Também criamos a classe Educador com os métodos ensinar e avaliar, e na classe Professor, ela herda tanto da classe Estudante quanto da classe Educador.

```
class Estudante(ABC):
    @abstractmethod
    def estudar(self):
        pass
```

```
class Educador(ABC):
    @abstractmethod
    def ensinar(self):
        pass

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

```
class Aluno(Estudante):
    def __init__(self,id: int = 0, nome: str = '', idade: int = 0, curso: str = '' ):
        #LOgica
    def estudar(self):
        print("Estudando!")
```

```
class Professor(Estudante, Educador):
    def __init__(self,matricula: int = 0 ,  nome: str = '', idade: int = 0, cargaHoraria: int = 0, salario: float = 0 ):
        #Logica

    def estudar(self):
        print("Estudando conteúdo novo!")

    def ensinar(self):
        print("Ensinando alunos!")

    def avaliar(self):
        print("Avaliando aluno!")
```

Inversão de Dependência

Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender da abstração. Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.