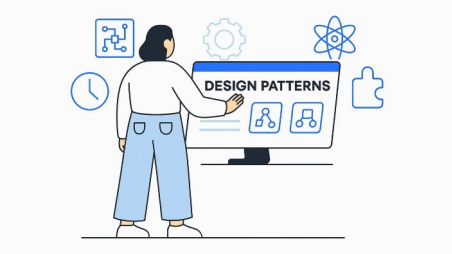
Padrões de Desenvolvimento de Software

Padrões de Desenvolvimento de Software

- Padrões de desenvolvimento de software são soluções reutilizáveis para problemas recorrentes encontrados durante o processo de construção de sistemas. Eles representam boas práticas consolidadas pela experiência de desenvolvedores ao longo do tempo;
- Esses padrões podem se manifestar em diferentes níveis, desde a organização da arquitetura de um sistema até detalhes mais específicos de interação entre classes e objetos.

Padrões de Desenvolvimento de Software

- De forma geral, existem dois grandes tipos:
  - **Padrões Arquiteturais:** definem a estrutura e a organização geral de um sistema (por exemplo, como os módulos se comunicam e se separam);
  - **Padrões de Projeto (Design Patterns)**: tratam de soluções mais pontuais, voltadas à interação entre classes e objetos para resolver problemas de design específicos.

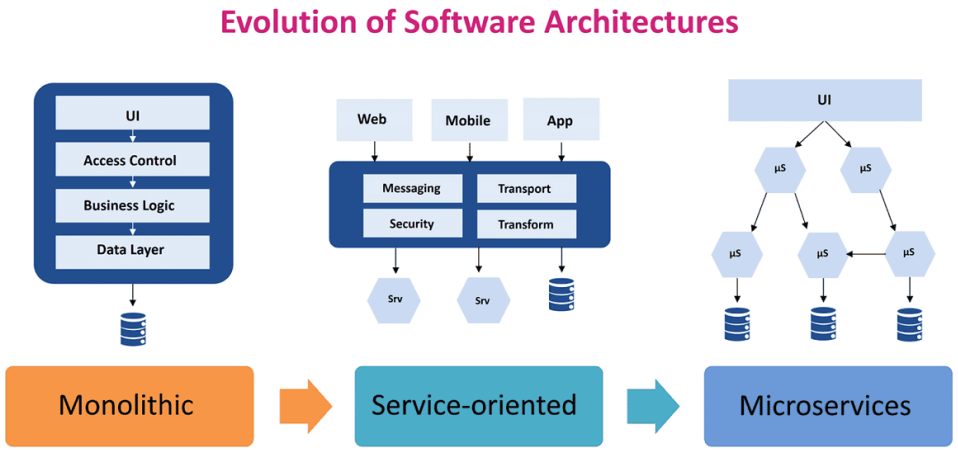


Padrões de Desenvolvimento de Software

- O uso de padrões traz diversos benefícios:
  - **Reuso de soluções testadas:** evita reinventar a roda e acelera o desenvolvimento;
  - **Organização do código:** promove clareza na estrutura do sistema e facilita a colaboração entre equipes;
  - **Facilidade de manutenção e evolução:** um sistema construído com base em padrões é mais modular e compreensível;
  - **Padronização:** desenvolvedores diferentes conseguem entender e modificar o código com mais facilidade, pois seguem convenções reconhecidas pela comunidade.

Padrões Arquiteturais

- Os padrões arquiteturais definem a estrutura e a organização global de um sistema de software. Eles indicam como os componentes principais são divididos, como se comunicam e como os dados fluem entre eles;
- Enquanto os Design Patterns resolvem problemas locais no design de classes e objetos, os padrões arquiteturais tratam do nível macro, influenciando toda a aplicação.

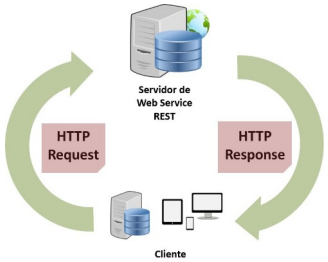


Padrões Arquiteturais

- Padrões arquiteturais mais utilizados:
  - **Arquitetura em Camadas:** Organiza o sistema em camadas horizontais, cada uma com funções e responsabilidades bem definidas. As camadas superiores utilizam os serviços das camadas inferiores, mas não o contrário. Essa separação facilita a manutenção, a reutilização de código e a substituição de partes do sistema sem afetar o todo;
  - **Arquitetura MVC (Model-View-Controller):** Divide a aplicação em três componentes principais: Model (representa os dados e a lógica de negócio), View (cuida da interface e exibição das informações) e Controller (gerencia as interações do usuário e coordena as ações entre o Model e a View). Essa arquitetura promove uma clara separação de responsabilidades, facilitando a evolução e os testes do sistema;

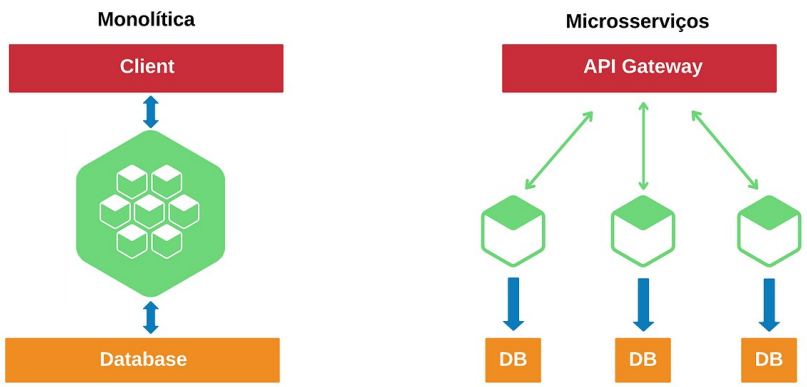
Padrões Arquiteturais

- Padrões arquiteturais mais utilizados:
  - **Arquitetura Orientada a Serviços Web:** É um modelo de desenvolvimento que organiza sistemas em torno de serviços independentes e reutilizáveis, acessíveis pela internet por meio de interfaces padronizadas. Baseia-se nos princípios da SOA (Service-Oriented Architecture), que propõe a integração entre aplicações distintas, e utiliza tecnologias como o REST (Representational State Transfer) para comunicação simples e leve via HTTP. Essa arquitetura favorece a interoperabilidade, escalabilidade e flexibilidade, permitindo que diferentes sistemas troquem informações de forma eficiente e padronizada;



Padrões Arquiteturais

- Padrões arquiteturais mais utilizados:
  - **Arquitetura em Microserviços:** Estrutura o sistema como um conjunto de pequenos serviços independentes, cada um responsável por uma funcionalidade específica. Esses serviços se comunicam por meio de APIs leves (geralmente REST) e podem ser implantados, atualizados e escalados individualmente, oferecendo grande flexibilidade e resiliência ao sistema.



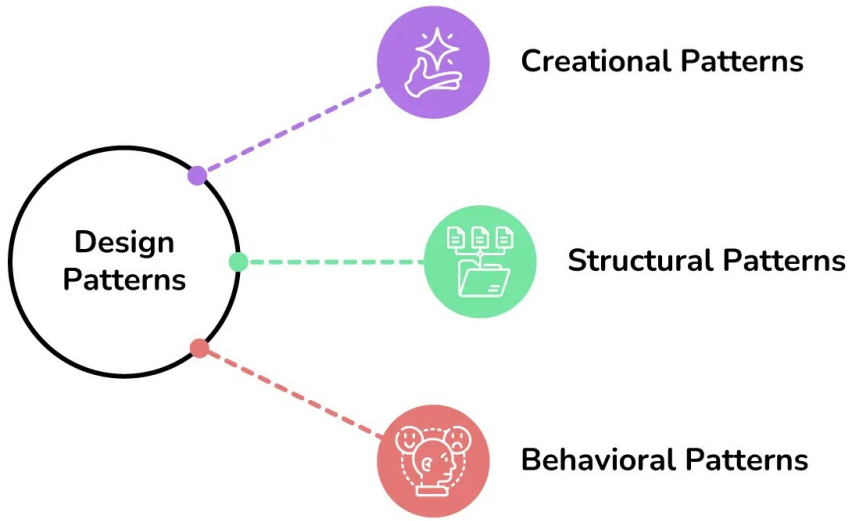
Arquitetura de Microsserviços

Padrões Arquiteturais

Arquitetura	Descrição	Foco Principal	Vantagens	Exemplo de Aplicação
Em Camadas	Organiza o sistema em camadas horizontais com responsabilidades específicas (ex: apresentação, negócio e dados).	Separação de responsabilidades e estruturação do sistema.	Facilita manutenção, testes e reuso de código.	Sistemas corporativos e aplicações com múltiplos níveis de lógica.
MVC (Model-View-Controller)	Divide a aplicação em três componentes: Model (dados), View (interface) e Controller (controle do fluxo).	Separar lógica de apresentação da lógica de negócio.	Organização, reutilização e facilidade de testes.	Aplicações web e desktop (ex: frameworks como Spring MVC, Django, Laravel).
Orientada a Serviços Web (SOA/REST)	Estrutura o sistema em serviços independentes e acessíveis via web por interfaces padronizadas, geralmente usando HTTP.	Integração e interoperabilidade entre sistemas distintos.	Escalabilidade, flexibilidade e comunicação padronizada.	Integração entre sistemas corporativos e APIs web.
Microserviços	Divide o sistema em pequenos serviços autônomos, cada um com sua própria lógica e banco de dados.	Modularidade, independência e escalabilidade dos componentes.	Implantação e atualização independentes, alta disponibilidade.	Grandes aplicações distribuídas e sistemas em nuvem.

Padrões de Projeto (Design Patterns)

- Os padrões de projeto são soluções reutilizáveis e testadas para problemas recorrentes de design de software, especialmente em sistemas orientados a objetos;
- Eles não são códigos prontos, mas modelos conceituais que orientam como as classes e objetos devem se relacionar para resolver determinado tipo de problema de forma elegante e eficiente.



Padrões de Projeto (Design Patterns)

- Os padrões de projeto são tradicionalmente classificados em três grandes grupos:
  - **Padrões Criacionais (Creational Patterns):** Tratam da criação e instanciação de objetos, promovendo flexibilidade e independência entre o código e o processo de criação;
  - **Padrões Estruturais (Structural Patterns):** Focam em como classes e objetos se organizam e se relacionam para formar estruturas maiores e mais flexíveis;
  - **Padrões Comportamentais (Behavioral Patterns):** Definem como os objetos interagem e se comunicam entre si, distribuindo responsabilidades de maneira eficiente.

Padrões de Projeto (Design Patterns) – Padrões Criacionais

Padrão	Descrição
Singleton	Garante que exista **apenas uma instância** de uma classe e fornece um **ponto global de acesso** a ela.
Factory Method	Define uma **interface para criar objetos**, permitindo que as subclasses decidam qual classe instanciar.
Abstract Factory	Fornece uma **interface para criar famílias de objetos relacionados**, sem especificar suas classes concretas.
Builder	Separa a **construção de um objeto complexo** da sua representação, permitindo diferentes configurações.
Prototype	Cria novos objetos **a partir da cópia de um objeto existente (protótipo)**, evitando a criação direta.

Padrões de Projeto (Design Patterns) – Padrões Criacionais

- Entre os padrões criacionais, o mais utilizado é o Singleton. Principal característica:
  - **Garantia de uma única instância:** muito útil para recursos que precisam ser únicos, como conexões de banco de dados, logs ou gerenciadores de configuração.
- O método __new__ em Python é um método especial responsável por criar uma nova instância de uma classe, antes mesmo do método __init__ ser chamado. Ele é chamado sempre que você cria um objeto, ou seja, quando você escreve obj = MinhaClasse();
- __new__ cria o objeto na memória. __init__ inicializa o objeto recém-criado, configurando atributos e realizando tarefas de inicialização.

Padrões de Projeto (Design Patterns) – Padrões Criacionais

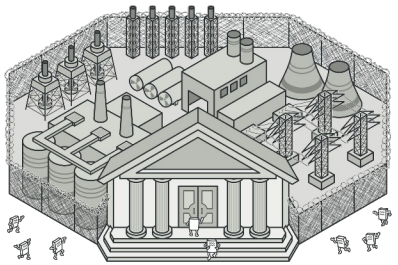
```
class Singleton:
    _instance = None  # Armazena a instância única

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(Singleton, cls).__new__(cls)
            # Inicialização de atributos
            cls._instance.valor = 0
        return cls._instance

# Testando o Singleton
obj1 = Singleton()
obj2 = Singleton()

obj1.valor = 100
print(obj2.valor)  # Saída: 100, pois obj1 e obj2 são a mesma instância

print(obj1 is obj2)  # Saída: True, confirma que é a mesma instância
```

Padrões de Projeto (Design Patterns) – Padrões Estruturais

Padrão	Descrição
Adapter	Permite que **classes com interfaces incompatíveis** trabalhem juntas, agindo como um adaptador entre elas.
Bridge	Separa uma abstração da sua implementação, permitindo **mudanças independentes em ambas**.
Composite	Organiza objetos em **estruturas hierárquicas (árvores)**, permitindo tratar objetos individuais e compostos da mesma forma.
Decorator	Adiciona **comportamentos extras a objetos dinamicamente**, sem alterar sua estrutura original.
Facade	Fornece uma **interface simplificada** para um conjunto de classes complexas, facilitando o uso de subsistemas.
Flyweight	Otimiza o uso de memória ao **compartilhar objetos comuns** entre múltiplas instâncias.
Proxy	Cria um **objeto intermediário** que controla o acesso a outro objeto (por exemplo, controle remoto, cache, segurança).

Padrões de Projeto (Design Patterns) – Padrões Estruturais

- Entre os padrões estruturais, o Facade (Fachada) é geralmente o mais utilizado. Principais características:
  - **Simplifica a complexidade:** fornece uma interface única e simplificada para um subsistema complexo, escondendo detalhes internos;
  - **Reduz acoplamento:** clientes interagem apenas com a fachada, não com várias classes internas, tornando o sistema mais modular e flexível;
  - **Facilita manutenção e evolução:** mudanças internas no subsistema normalmente não afetam os clientes que usam a fachada.



Padrões de Projeto (Design Patterns) – Padrões Estruturais

```
# Subsistema complexo
class SistemaAudio:
    def ligar_audio(self):
        print("Áudio ligado")

    def configurar_volume(self, volume):
        print(f"Volume ajustado para {volume}")

class SistemaVideo:
    def ligar_video(self):
        print("Vídeo ligado")

    def configurar_resolucao(self, resolucao):
        print(f"Resolução ajustada para {resolucao}")

class SistemaLuzes:
    def ligar_luzes(self):
        print("Luzes ligadas")

    def ajustar_intensidade(self, intensidade):
        print(f"Intensidade das luzes ajustada para {intensidade}")
```

Padrões de Projeto (Design Patterns) – Padrões Estruturais

```
# Facade: interface simplificada
class HomeTheaterFacade:
    def __init__(self):
        self.audio = SistemaAudio()
        self.video = SistemaVideo()
        self.luzes = SistemaLuzes()

    def iniciar_filme(self):
        print("Preparando o Home Theater...")
        self.luzes.ajustar_intensidade(30)
        self.audio.ligar_audio()
        self.audio.configurar_volume(50)
        self.video.ligar_video()
        self.video.configurar_resolucao("1080p")
        print("Filme pronto para começar!")

# Cliente usando a Facade
home_theater = HomeTheaterFacade()
home_theater.iniciar_filme()
```

Padrões de Projeto (Design Patterns) – Padrões Comportamentais

Padrão	Descrição
Observer	Define uma dependência entre objetos de modo que, quando um muda de estado, **todos os dependentes são notificados automaticamente**.
Strategy	Permite **alterar o algoritmo usado por um objeto** em tempo de execução, trocando estratégias dinamicamente.
Command	Encapsula uma **solicitação (ação) em um objeto**, permitindo que ela seja executada, desfeita ou armazenada.
Iterator	Fornece uma maneira de **percorrer os elementos de uma coleção** sem expor sua estrutura interna.
State	Permite que um objeto **mude seu comportamento quando seu estado interno muda**, parecendo alterar sua classe.

Padrões de Projeto (Design Patterns) – Padrões Comportamentais

Padrão	Descrição
Template Method	Define o **esqueleto de um algoritmo**, deixando que subclasses implementem partes específicas.
Chain of Responsibility	Evita o acoplamento entre remetente e receptor, **encaminhando uma solicitação por uma cadeia de objetos** até que um a trate.
Mediator	Centraliza a comunicação entre objetos, **reduzindo dependências diretas entre eles**.
Memento	Permite **salvar e restaurar o estado interno** de um objeto sem violar o encapsulamento.
Visitor	Permite **definir novas operações sobre uma estrutura de objetos** sem alterar suas classes.

Padrões de Projeto (Design Patterns) – Padrões Comportamentais

- Entre os padrões comportamentais, o Observer é geralmente o mais utilizado. Principais características:
  - **Atualização automática:** permite que vários objetos sejam notificados automaticamente quando o estado de outro objeto muda;
  - **Desacoplamento:** os objetos observadores não precisam conhecer os detalhes internos do objeto observado, tornando o sistema mais flexível e modular;
  - **Versatilidade:** é amplamente usado em interfaces gráficas, sistemas de eventos, notificações e fluxos de dados reativos.

Padrões de Projeto (Design Patterns) – Padrões Comportamentais

```
# Sujeito (Subject)
class Assunto:
    def __init__(self):
        self._observadores = []
        self._estado = None

    def adicionar_observador(self, observador):
        self._observadores.append(observador)

    def remover_observador(self, observador):
        self._observadores.remove(observador)

    def notificar_observadores(self):
        for observador in self._observadores:
            observador.atualizar(self._estado)

    def set_estado(self, estado):
        self._estado = estado
        print(f"\n[Sujeito] Estado alterado para: {self._estado}")
        self.notificar_observadores()
```

Padrões de Projeto (Design Patterns) – Padrões Comportamentais

```
# Observador (Observer)
class Observador:
    def __init__(self, nome):
        self.nome = nome

    def atualizar(self, estado):
        print(f"{self.nome} recebeu atualização:
{estado}")
```

```
# Testando o Observer
assunto = Assunto()

obs1 = Observador("Observador 1")
obs2 = Observador("Observador 2")
obs3 = Observador("Observador 3")

assunto.adicionar_observador(obs1)
assunto.adicionar_observador(obs2)
assunto.adicionar_observador(obs3)

# Alterando o estado do sujeito
assunto.set_estado("ON")
assunto.set_estado("OFF")
```

Princípios SOLID

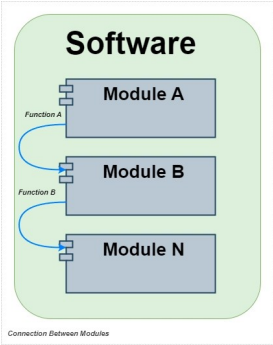
- Os princípios SOLID são diretrizes de design orientado a objetos que visam tornar o código mais compreensível, flexível e fácil de manter. O acrônimo representa cinco princípios:
  - **S** — Single Responsibility Principle (Princípio da Responsabilidade Única);
  - **O** — Open/Closed Principle (Aberto/Fechado);
  - **L** — Liskov Substitution Principle (Substituição de Liskov);
  - **I** — Interface Segregation Principle (Segregação de Interface);
  - **D** — Dependency Inversion Principle (Inversão de Dependência).
- Esses princípios servem de base para a aplicação eficaz de padrões de projeto, garantindo um código limpo, modular e sustentável.

S – Single Responsibility Principle (Princípio da Responsabilidade Única)

- Uma classe deve ter apenas uma razão para mudar, ou seja, uma única responsabilidade.
  - Objetivo: reduzir acoplamento e facilitar manutenção;
  - Exemplo: uma classe que somente gerencia dados do cliente e outra que envia e-mails, em vez de juntar tudo em uma só.

S – Single Responsibility Principle (Princípio da Responsabilidade Única)

- O acoplamento é um conceito de design de software que descreve o grau de dependência entre diferentes módulos ou componentes de um sistema.
  - Alto acoplamento: módulos estão fortemente dependentes uns dos outros. Alterações em um módulo podem exigir mudanças em outros, tornando o sistema mais rígido, difícil de manter e testar;
  - Baixo acoplamento: módulos são independentes e comunicam-se apenas pelo necessário, usando interfaces ou abstrações. Isso aumenta a flexibilidade, reutilização e facilidade de manutenção.



S – Single Responsibility Principle (Princípio da Responsabilidade Única)

```
class Relatorio:
    def __init__(self, dados):
        self.dados = dados

    def gerar_relatorio(self):
        print("Gerando relatório com os dados:", self.dados)

    def salvar_em_arquivo(self, nome_arquivo):
        with open(nome_arquivo, "w") as file:
            file.write(str(self.dados))
        print(f"Relatório salvo em {nome_arquivo}")
```

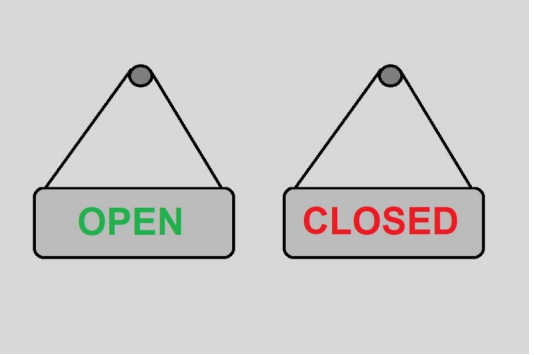
S – Single Responsibility Principle (Princípio da Responsabilidade Única)

```
# Responsabilidade 1: gerar relatório
class Relatorio:
    def __init__(self, dados):
        self.dados = dados

    def gerar(self):
        print("Gerando relatório com os dados:", self.dados)
        return str(self.dados)

# Responsabilidade 2: salvar relatório
class Arquivo:
    @staticmethod
    def salvar(nome_arquivo, conteudo):
        with open(nome_arquivo, "w") as file:
            file.write(conteudo)
        print(f"Relatório salvo em {nome_arquivo}")
```

```
# Uso
dados = {"vendas": 1000, "clientes": 50}
relatorio = Relatorio(dados)
conteudo = relatorio.gerar()
Arquivo.salvar("relatorio.txt", conteudo)
```

O – Open/Closed Principle (Princípio Aberto/Fechado)

- Uma entidade (classe, módulo, função) deve estar aberta para extensão, mas fechada para modificação.
  - Objetivo: permitir adicionar funcionalidades sem alterar o código existente, evitando que mudanças causem erros;
  - Exemplo: adicionar novos tipos de pagamento através de subclasses ou interfaces, sem modificar a classe principal.



O – Open/Closed Principle (Princípio Aberto/Fechado)

```
class CalculadoraDesconto:
    def calcular(self, tipo_cliente, valor):
        if tipo_cliente == "normal":
            return valor * 0.9  # 10% de desconto
        elif tipo_cliente == "premium":
            return valor * 0.8  # 20% de desconto

# Problema: toda vez que aparecer um novo tipo de cliente, a classe precisa ser modificada
```

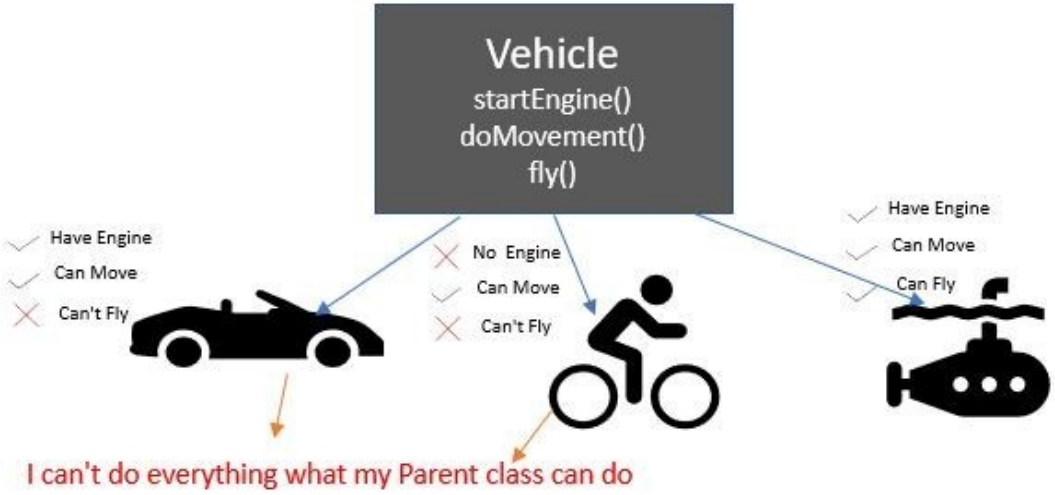
O – Open/Closed Principle (Princípio Aberto/Fechado)

```
from abc import ABC, abstractmethod

# Interface para desconto
class Desconto(ABC):
    @abstractmethod
    def aplicar(self, valor):
        pass

# Implementações concretas
class DescontoNormal(Desconto):
    def aplicar(self, valor):
        return valor * 0.9  # 10% de desconto

class DescontoPremium(Desconto):
    def aplicar(self, valor):
        return valor * 0.8  # 20% de desconto

# Calculadora usa abstração
class CalculadoraDesconto:
    def __init__(self, estrategia_desconto: Desconto):
        self.estrategia_desconto = estrategia_desconto

    def calcular(self, valor):
        return self.estrategia_desconto.aplicar(valor)
```

```
# Uso
valor = 1000

desconto_normal = CalculadoraDesconto(DescontoNormal())
print(desconto_normal.calcular(valor))  # 900.0

desconto_premium = CalculadoraDesconto(DescontoPremium())
print(desconto_premium.calcular(valor))  # 800.0
```

L – Liskov Substitution Principle (Princípio da Substituição de Liskov)

- Subclasses devem ser substituíveis por suas classes base sem alterar o comportamento esperado do sistema.
  - Objetivo: garantir coerência no polimorfismo;
  - Exemplo: se Cachorro é uma subclasse de Animal, todas as funções que usam Animal devem funcionar corretamente com Cachorro.



L – Liskov Substitution Principle (Princípio da Substituição de Liskov)

```
class Retangulo:
    def __init__(self, largura, altura):
        self.largura = largura
        self.altura = altura

    def set_largura(self, largura):
        self.largura = largura

    def set_altura(self, altura):
        self.altura = altura

    def area(self):
        return self.largura * self.altura

class Quadrado(Retangulo):
    def set_largura(self, largura):
        self.largura = largura
        self.altura = largura  # força altura igual à largura

    def set_altura(self, altura):
        self.altura = altura
        self.largura = altura  # força largura igual à altura
```

```
# Problema
retangulo = Quadrado(5, 5)
retangulo.set_largura(10)
print(retangulo.area())  # Saída: 100,
esperado 50 se fosse um Retangulo
```

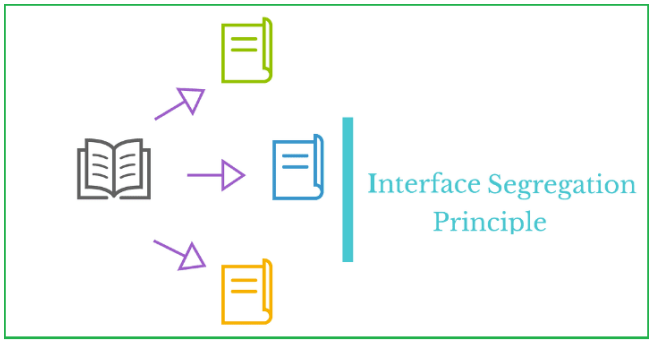
L – Liskov Substitution Principle (Princípio da Substituição de Liskov)

```
from abc import ABC, abstractmethod

class Forma(ABC):
    @abstractmethod
    def area(self):
        pass

class Retangulo(Forma):
    def __init__(self, largura, altura):
        self.largura = largura
        self.altura = altura

    def area(self):
        return self.largura * self.altura

class Quadrado(Forma):
    def __init__(self, lado):
        self.lado = lado

    def area(self):
        return self.lado * self.lado
```

```
# Uso
formas = [Retangulo(5, 10), Quadrado(5)]
for forma in formas:
    print(forma.area())
```

I – Interface Segregation Principle (Princípio da Segregação de Interface)

- Os clientes não devem ser obrigados a depender de interfaces que não utilizam.
  - Objetivo: criar interfaces menores e mais específicas, evitando métodos inúteis;
  - Exemplo: em vez de uma única interface Impressora com métodos de imprimir e digitalizar, criar interfaces separadas Impressora e Scanner.



I – Interface Segregation Principle (Princípio da Segregação de Interface)

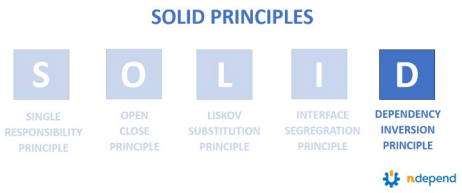
```
from abc import ABC, abstractmethod

class Impressora (ABC):
    @abstractmethod
    def imprimir(self, documento):
        pass

    @abstractmethod
    def digitalizar(self, documento):
        pass

    @abstractmethod
    def enviar_fax(self, documento):
        pass

# Problema: algumas impressoras podem não suportar digitalizar ou enviar fax,
# mas ainda assim seriam obrigadas a implementar esses métodos.
```

I – Interface Segregation Principle (Princípio da Segregação de Interface)

```
from abc import ABC, abstractmethod

# Interfaces menores e específicas
class Impressora(ABC):
    @abstractmethod
    def imprimir(self, documento):
        pass

class Scanner(ABC):
    @abstractmethod
    def digitalizar(self, documento):
        pass

class Fax(ABC):
    @abstractmethod
    def enviar_fax(self, documento):
        pass

# Implementações concretas
class ImpressoraSimples(Impressora):
    def imprimir(self, documento):
        print(f"Imprimindo: {documento}")

class Multifuncional(Impressora, Scanner, Fax):
    def imprimir(self, documento):
        print(f"Imprimindo: {documento}")

    def digitalizar(self, documento):
        print(f"Digitalizando: {documento}")

    def enviar_fax(self, documento):
        print(f"Enviando fax: {documento}")

# Uso
imp = ImpressoraSimples()
imp.imprimir("Relatório")

multi = Multifuncional()
multi.imprimir("Relatório")
multi.digitalizar("Relatório")
multi.enviar_fax("Relatório")
```

D – Dependency Inversion Principle (Princípio da Inversão de Dependência)

- Módulos de alto nível não devem depender de módulos de baixo nível, ambos devem depender de abstrações (interfaces ou classes abstratas).
  - Objetivo: reduzir acoplamento e aumentar flexibilidade;
  - Exemplo: uma classe ControlePagamento depende de uma interface Pagamento, e não de implementações concretas como PagamentoCartao ou PagamentoBoleto.



D – Dependency Inversion Principle (Princípio da Inversão de Dependência)

```
class MySQLDatabase:
    def salvar(self, dados):
        print(f"Salvando {dados} no MySQL")

class UsuarioService:
    def __init__(self):
        self.db = MySQLDatabase()  # Dependência direta de implementação concreta

    def criar_usuario(self, nome):
        self.db.salvar(nome)
        print(f"Usuário {nome} criado com sucesso!")

# Problema: UsuarioService está fortemente acoplado ao MySQLDatabase
```

D – Dependency Inversion Principle (Princípio da Inversão de Dependência)

```
from abc import ABC, abstractmethod

# Abstração
class Database(ABC):
    @abstractmethod
    def salvar(self, dados):
        pass

# Implementações concretas
class MySQLDatabase(Database):
    def salvar(self, dados):
        print(f"Salvando {dados} no
MySQL")

class PostgreSQLDatabase(Database):
    def salvar(self, dados):
        print(f"Salvando {dados} no
PostgreSQL")
```

```
# Módulo de alto nível depende da abstração, não
da implementação concreta
class UsuarioService:
    def __init__(self, db: Database):
        self.db = db

    def criar_usuario(self, nome):
        self.db.salvar(nome)
        print(f"Usuário {nome} criado com sucesso!")

# Uso
mysql_service =
UsuarioService(MySQLDatabase())
mysql_service.criar_usuario("Henrique")

postgres_service =
UsuarioService(PostgreSQLDatabase())
postgres_service.criar_usuario("Sofia")
```

FIM