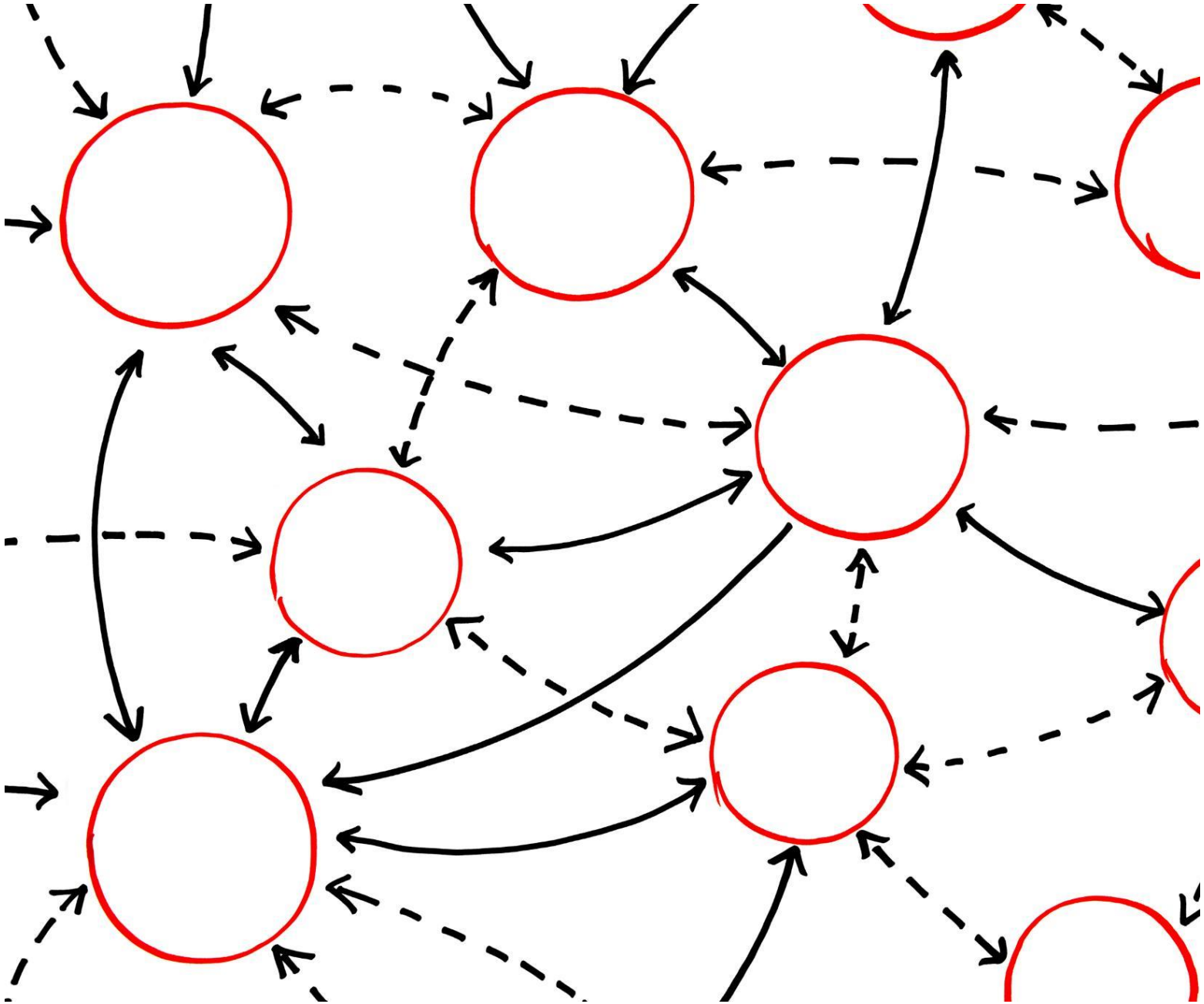
Encapsulamento: Fundamentos e Aplicação na Programação Orientada a Objetos com Python

Princípios essenciais para proteger dados e organizar código

Agenda da Sessão

- Compreendendo o conceito de encapsulamento
- Aplicação prática do encapsulamento em Python
- Exemplo de código implementando encapsulamento
- Vantagens do encapsulamento no desenvolvimento de software
- Delegação entre Classes

Compreendendo o conceito de encapsulamento



Definição de encapsulamento em orientação a objetos

Princípio do Encapsulamento

Encapsulamento consiste em ocultar dados internos para proteger informações dentro de objetos.

Acesso Controlado

Permite o acesso aos dados somente por métodos definidos, garantindo segurança e isolamento.

Objetivos e benefícios do encapsulamento

Proteção de Dados

Encapsulamento protege dados contra alterações não autorizadas, garantindo integridade e segurança das informações.

Modularidade do Código

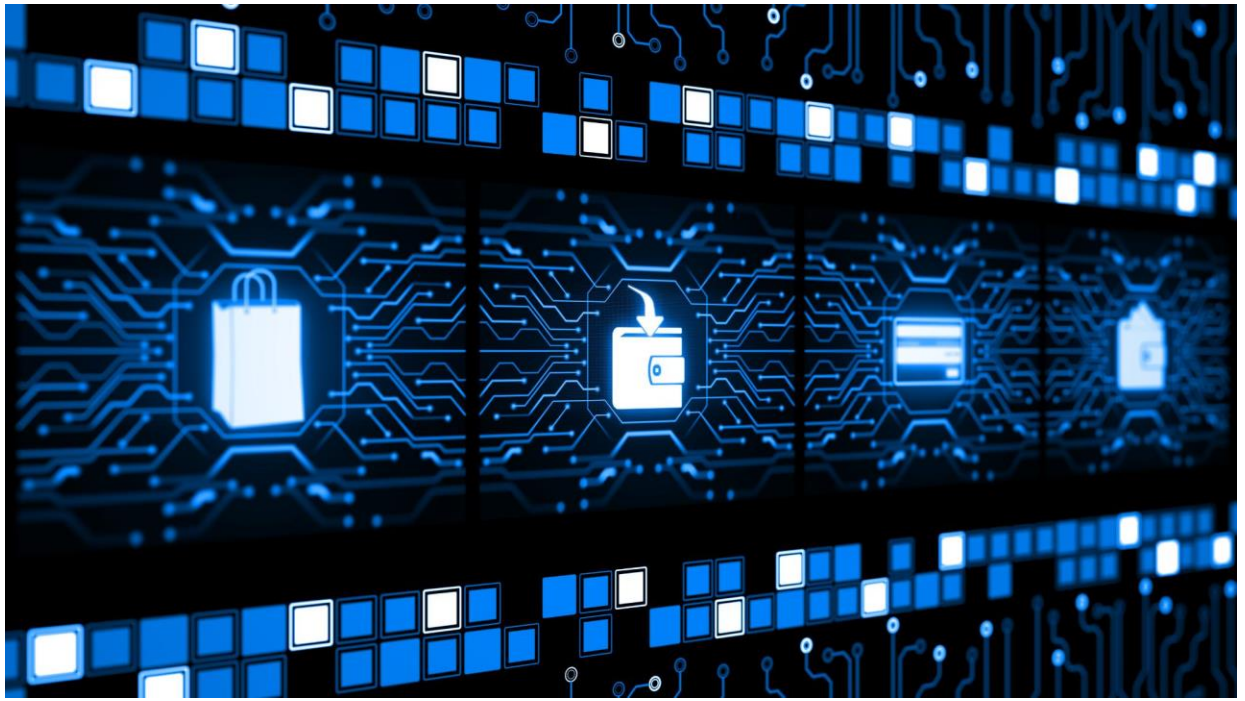
Promove a organização do código em módulos independentes, facilitando o desenvolvimento e a reutilização.

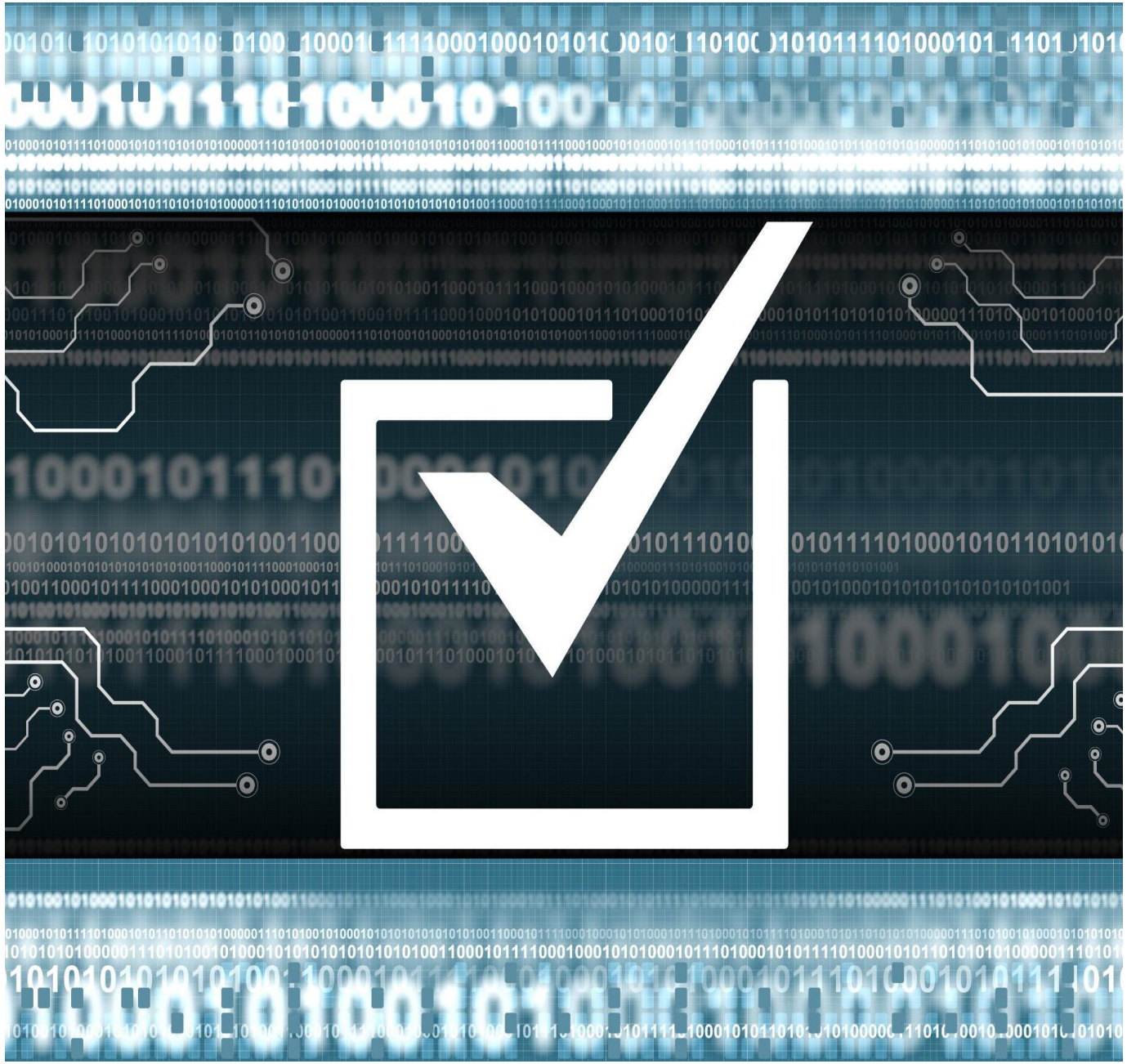
Facilidade de Manutenção

Facilita a manutenção do software ao controlar o acesso e reduzir dependências entre componentes.

Acesso Controlado

Garante que o acesso aos dados seja seguro e controlado, aumentando a robustez do software.





Encapsulamento para Validação

Proteção dos Atributos

Encapsular atributos limita o acesso direto, permitindo manipulação apenas por métodos definidos. Isso reforça o controle sobre o estado dos dados.

Validação com Getters e Setters

Getters e setters possibilitam validar e controlar valores antes da atribuição, prevenindo dados incorretos e mantendo a integridade do sistema.

Segurança e Manutenção do Código

O encapsulamento facilita a manutenção, reduz erros e aumenta a segurança, tornando o sistema mais confiável e eficiente.

Definindo ***Propriedades*** em Classes

Encapsulamento com Propriedades

O uso de setters e getters permite proteger os atributos das classes, controlando o acesso e modificações.

Facilidade com @property

O decorador @property (Python) transforma métodos em atributos, tornando o acesso aos dados mais intuitivo e seguro.

Manutenção e Flexibilidade

A abordagem com propriedades facilita a manutenção do código e oferece maior flexibilidade para futuras modificações.

Aplicação prática do encapsulamento em Python

Estrutura básica de uma classe em Python

Definição de Classe

Classes em Python são definidas usando a palavra-chave 'class' para criar novos tipos de objetos.

Atributos da Classe

Atributos armazenam dados que representam o estado dos objetos criados a partir da classe.

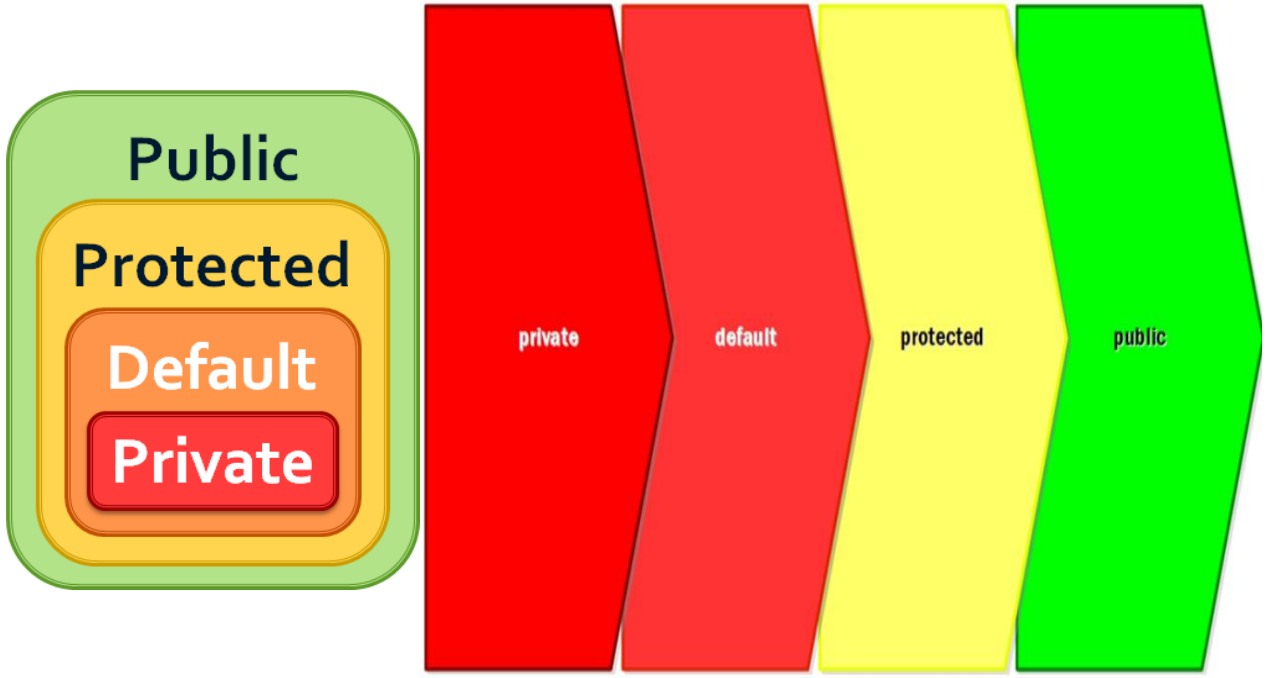
Métodos da Classe

Métodos definem o comportamento dos objetos e são funções associadas a uma classe.

Visibilidade de uma Classe

- **Visibilidade pública (+)**
  - Atributo e/ou método pode ser acessado por operações de todas classes
- **Visibilidade protegida (#)**
  - Atributo e/ou método pode ser acessado apenas pelas operações da própria classe ou por operações de classes dentro de um mesmo pacote.
- **Visibilidade privada (-)**
  - Atributo e/ou método só pode ser acessado por operações da própria classe.

Visibilidade	public	protected	default	private
A partir da mesma classe	✓	✓	✓	✓
Qualquer classe no mesmo pacote	✓	✓	✓	⊘
Qualquer classe filha no mesmo pacote	✓	✓	✓	⊘
Qualquer classe filha em pacote diferente	✓	✓	⊘	⊘
Qualquer classe em pacote diferente	✓	⊘	⊘	⊘



Criação de atributos público, protegidos e privados em Python

Atributos Público
Atributos públicos, basta o nome do atribuo ou método.
Ex.: idade (público)

Atributos Protegidos

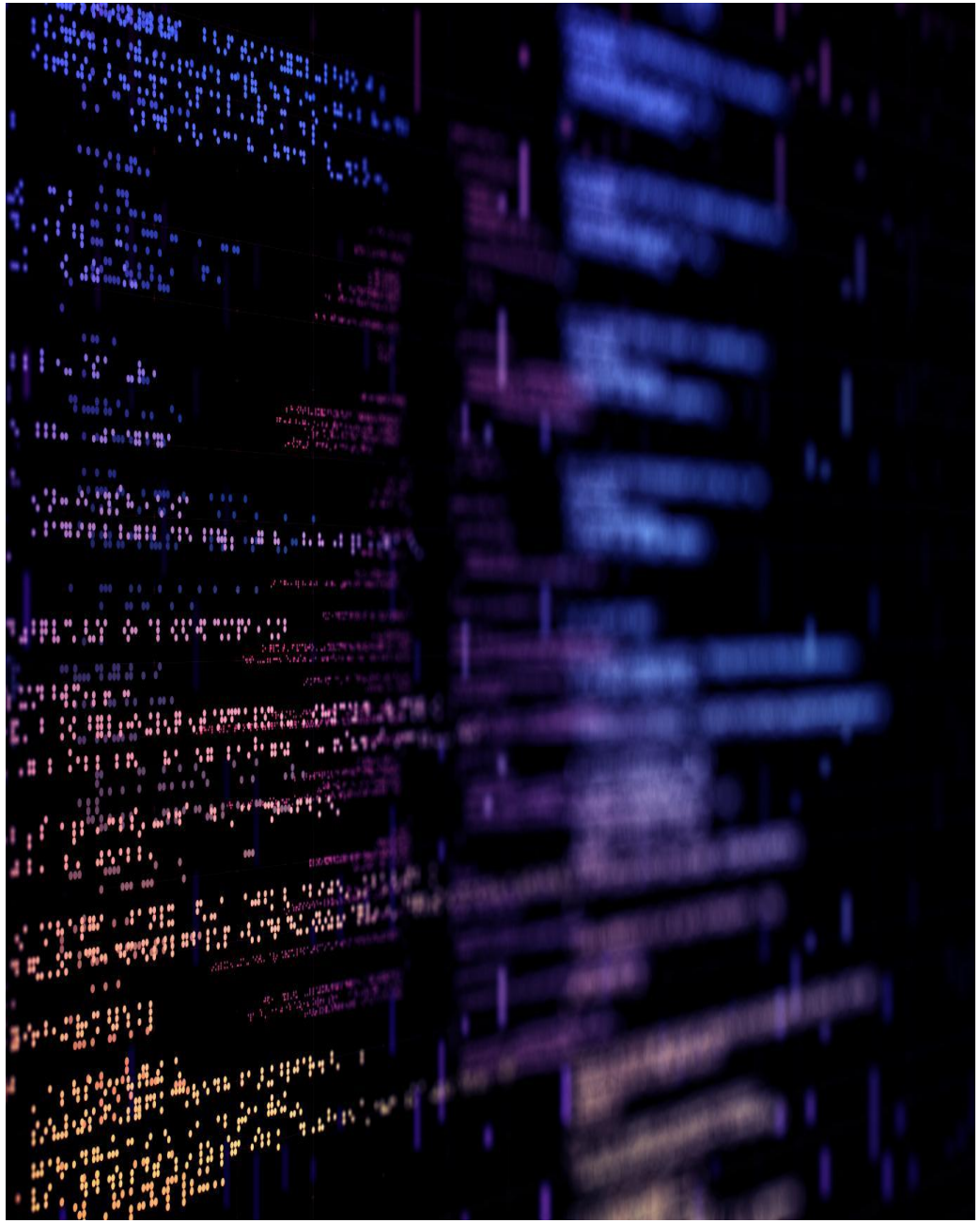
Atributos protegidos utilizam **um underscore (_)** para indicar acesso restrito dentro da classe e subclasses.

Ex.: _idade (protegido)

Atributos Privados

Atributos privados usam **dois underscores (__)** para ocultar dados, evitando acesso direto externo à classe.

Ex.: __idade (privado)



Uso de métodos getters e setters

Acesso Controlado a Atributos

Getters e ***setters*** permitem controlar o acesso a atributos privados de forma segura e eficiente.

Métodos Getters (função): Retorna (acessa) o valor do atributo.

Métodos Setters (procedimento): Altera o valor do atributo.

Aplicação de Regras de Validação

Esses métodos garantem que regras de validação sejam aplicadas ao acessar ou alterar dados.

Exemplo de código implementando encapsulamento

Construção de uma classe ***ContaBancaria***

Atributos Privados

A classe inclui atributos privados que protegem o saldo contra acesso direto e garantem segurança dos dados.

Métodos Públicos

Métodos públicos permitem operações seguras de depósito e saque, controlando o acesso ao saldo.

Encapsulamento Prático

Exemplo prático de encapsulamento, combinando dados privados com métodos públicos para controle do acesso.

Implementação dos mecanismos de encapsulamento

Atributos Privados

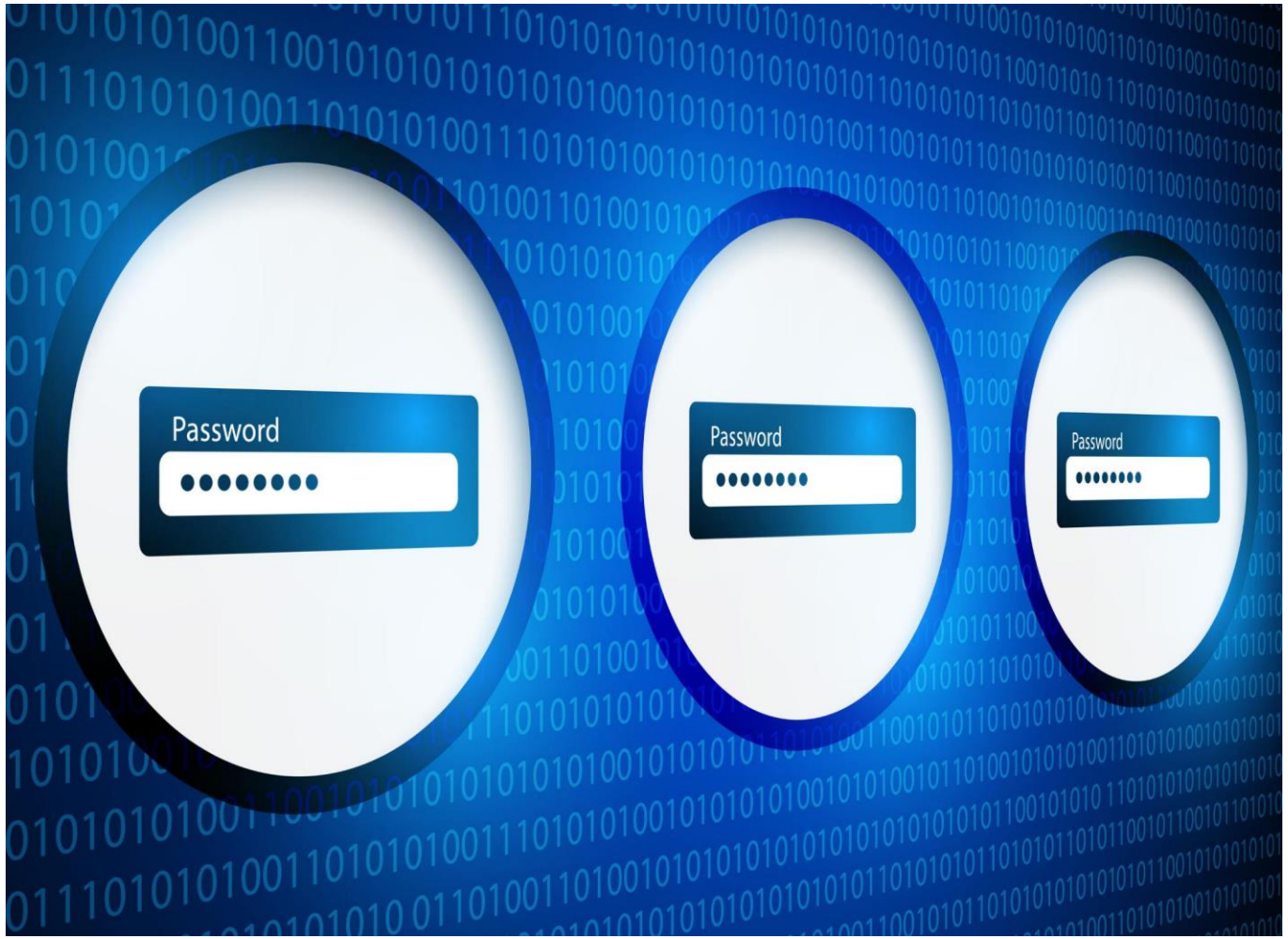
Os atributos privados garantem que os dados sensíveis, como saldo, não sejam acessados diretamente.

Métodos Getters e Setters

Getters e setters controlam o acesso e modificações, assegurando operações seguras na conta.

Prevenção de Alterações Indevidas

Encapsulamento evita alterações indevidas, mantendo a integridade dos dados da conta.

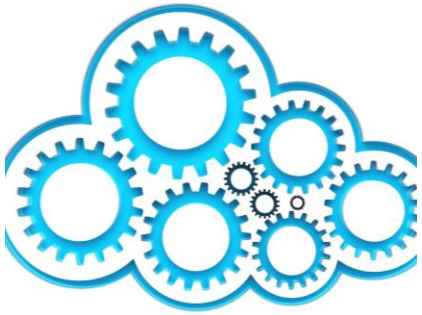


Controle de Acesso aos Dados

Métodos garantem acesso restrito aos dados protegendo informações sensíveis do software.

Garantia da Integridade dos Dados

Encapsulamento assegura que dados permaneçam consistentes e protegidos contra alterações indevidas.



Importância do Encapsulamento

Encapsulamento é fundamental para a segurança e robustez do software moderno.

```python
class ContaBancaria:
    def __init__(self, titular, saldo_inicial):
        self.titular = titular
        # atributo privado
        self.__saldo = saldo_inicial

    # Destruidor
    def __del__(self):
        print(f"Objeto destruído...")

    def depositar(self, valor):
        if valor > 0:
            self.__saldo += valor
            print(f"Depósito de R${valor:.2f} realizado com sucesso.")
        else:
            print("Valor de depósito inválido.")

    def sacar(self, valor):
        if 0 < valor <= self.__saldo:
            self.__saldo -= valor
            print(f"Saque de R${valor:.2f} realizado com sucesso.")
        else:
            print("Saldo insuficiente ou valor inválido.")

    def consultarSaldo(self):
        print(f"Saldo atual: R${self.__saldo:.2f}")
```

```python
from conta_bancaria import ContaBancaria

# Classe principal da aplicação
class Main:

    def __init__(self):
        pass

    def executar(self):
        contaBancaria = ContaBancaria("Benevaldo", 100)
        contaBancaria.consultarSaldo()
        contaBancaria.depositar(50)
        contaBancaria.consultarSaldo()
        contaBancaria.sacar(60)
        contaBancaria.consultarSaldo()

# Inicia a execução do programa
app = Main()
app.executar()
```

Vantagens do encapsulamento no desenvolvimento de software

Segurança e proteção dos dados

Encapsulamento de Dados

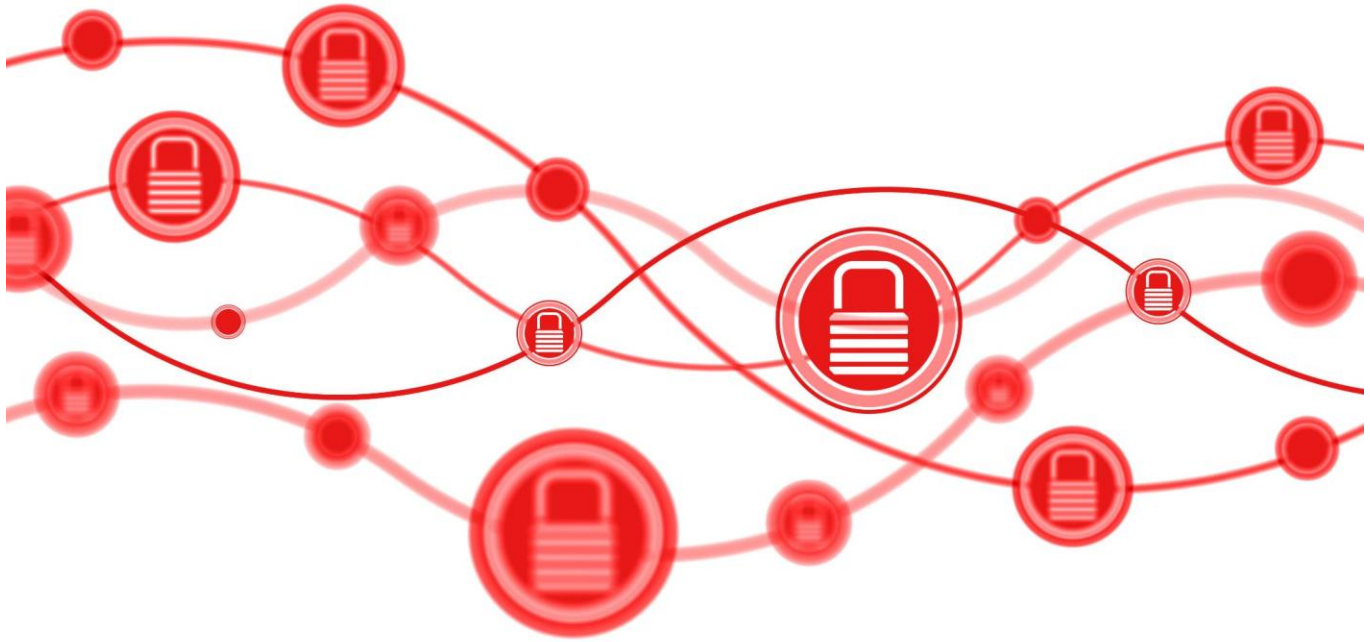
Encapsular dados evita acessos diretos não autorizados, protegendo informações sensíveis no software.

Proteção da Integridade

Encapsulamento mantém a integridade dos dados evitando modificações indevidas por usuários não autorizados.

Confidencialidade das Informações

A confidencialidade dos dados é garantida por meio do encapsulamento, impedindo vazamentos e acessos ilegais.



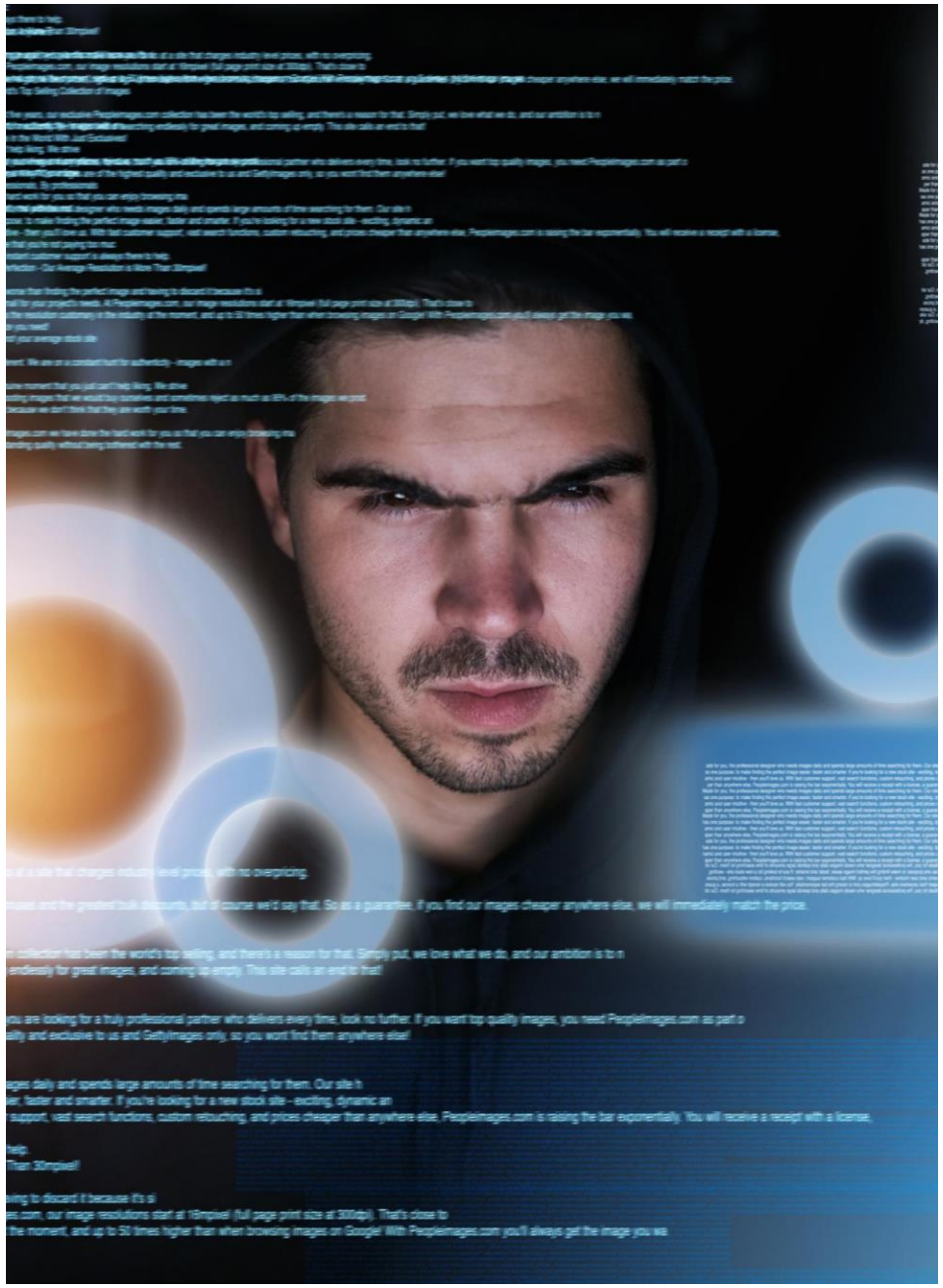
Facilidade de manutenção e atualização do código

Encapsulamento e Isolamento

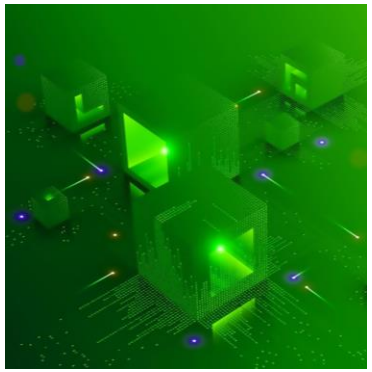
Alterações internas das classes são feitas sem afetar outras partes do sistema, garantindo isolamento eficiente.

Facilidade na Evolução

Encapsulamento facilita a manutenção e atualização contínua do software sem riscos de falhas externas.



Contribuição para código limpo e reutilizável

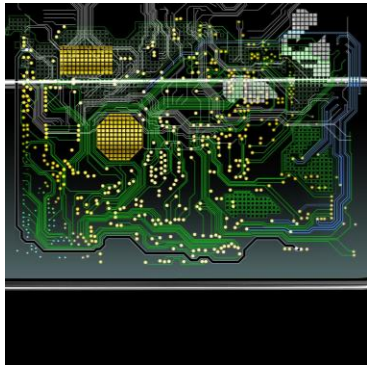


Encapsulamento e Modularidade

Encapsulamento melhora a modularidade e organização do código para maior clareza.

Código Limpo e Legível

Código encapsulado torna-se mais limpo e fácil de entender para desenvolvedores.



Reutilização de Código

Encapsulamento facilita a reutilização do código em diferentes projetos e sistemas.

Delegação em Orientação a Objetos: Conceitos, Aplicações e Relação com Encapsulamento

Princípios essenciais para código organizado e reutilizável

Compreendendo o conceito de Delegação em orientação a objetos

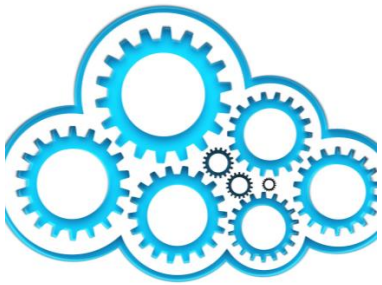
Definição formal de delegação

Conceito de Delegação

Delegação envolve um objeto transferindo uma tarefa a outro objeto especializado para execução adequada.

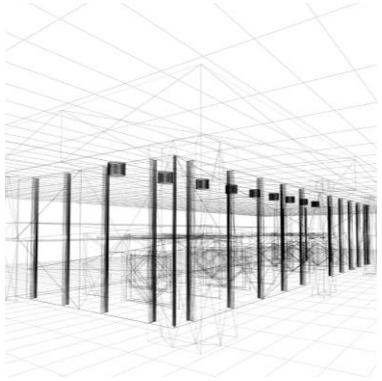
A **delegação** ocorre quando um objeto **confia a outro objeto** a responsabilidade de executar uma tarefa. Em vez de implementar diretamente uma funcionalidade, ele **encaminha** a chamada para outro objeto que sabe como lidar com ela.

Separação de Responsabilidades

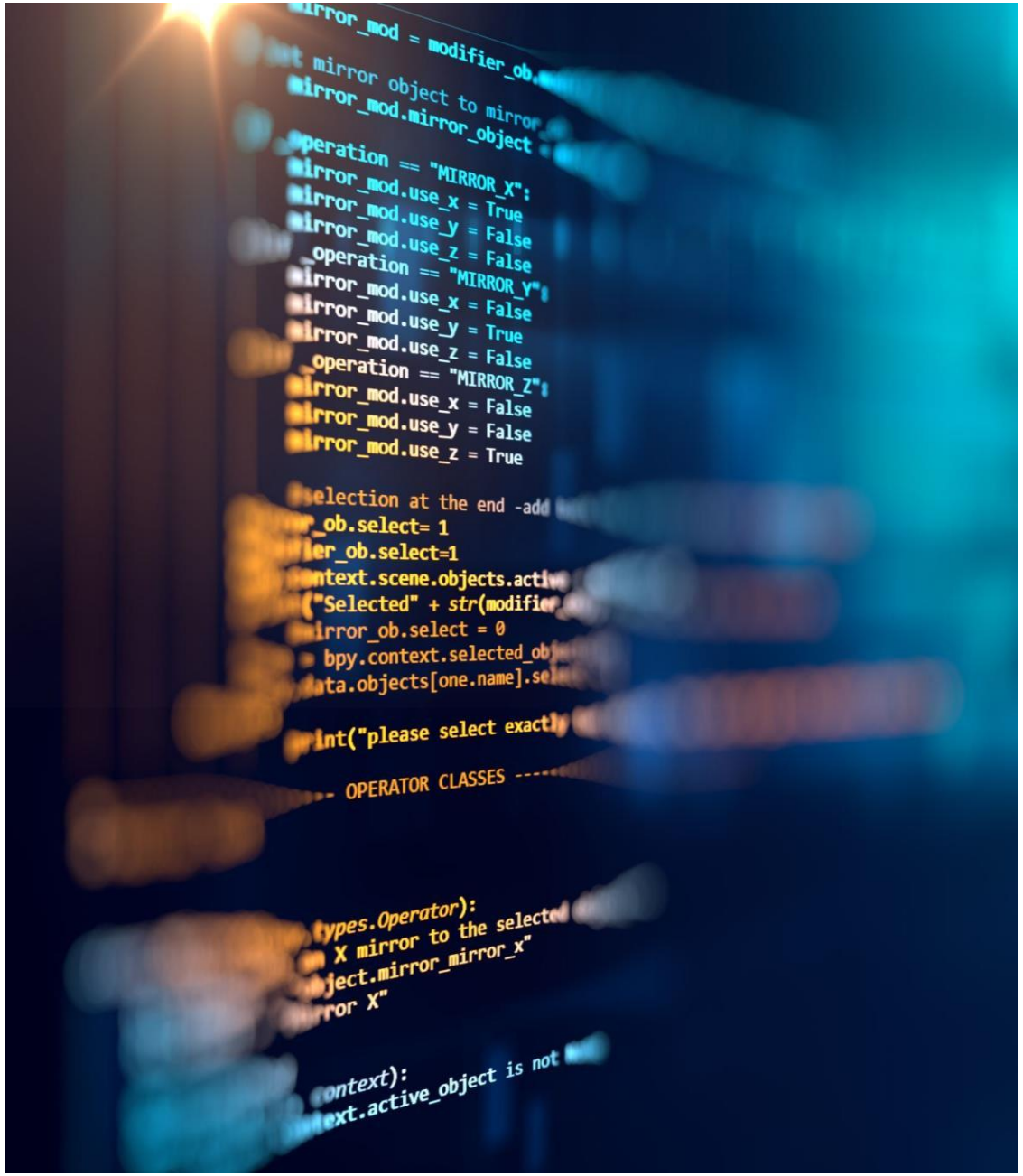


Delegação promove a divisão clara de responsabilidades entre diferentes objetos ou componentes do sistema.

Flexibilidade no Design



A prática de delegação aumenta a flexibilidade e facilita mudanças no design do sistema.



Como a delegação facilita a reutilização de código

Prevenção de Duplicação

Delegar tarefas evita a repetição de código, garantindo maior eficiência e clareza no desenvolvimento.

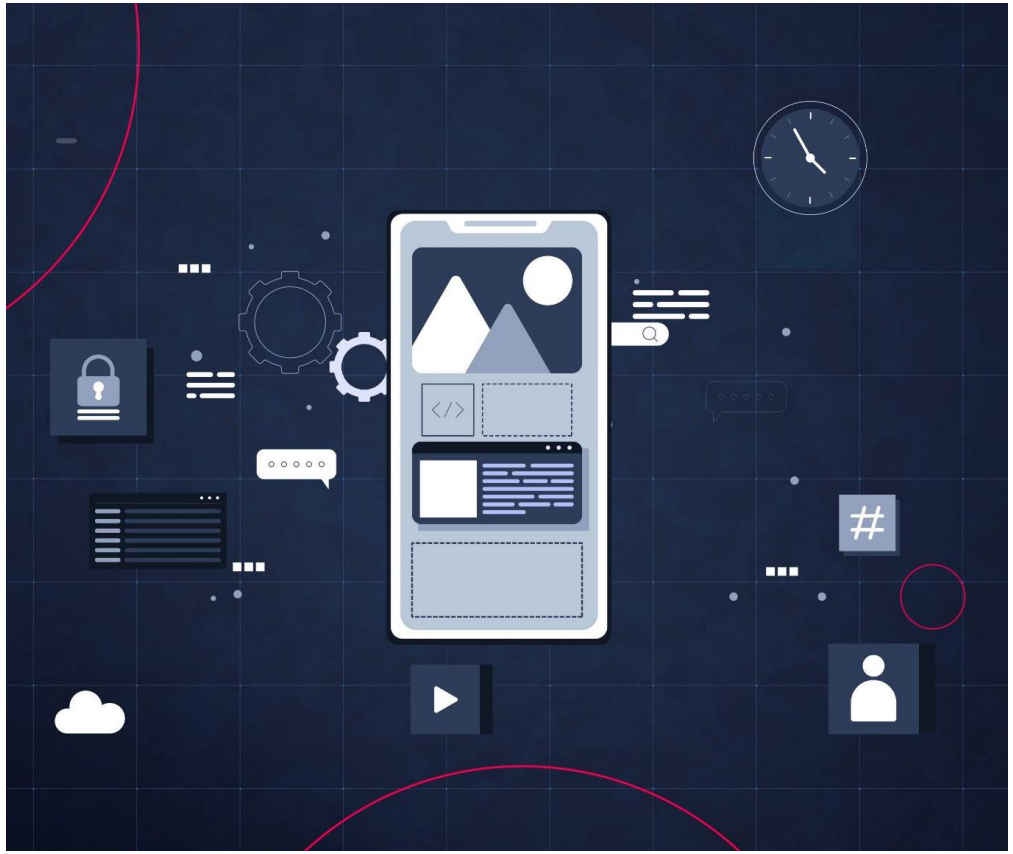
Composição de Comportamentos

A delegação permite combinar comportamentos complexos a partir de componentes simples e reutilizáveis.

Código Modular e Manutenível

Delegar tarefas torna o código mais modular, facilitando sua manutenção e evolução ao longo do tempo.

Exemplo prático de delegação em linguagens orientadas a objetos



Conceito de Delegação

Delegação permite que um objeto execute tarefas confiando em outro objeto especializado. Isso promove reuso e organização do código.

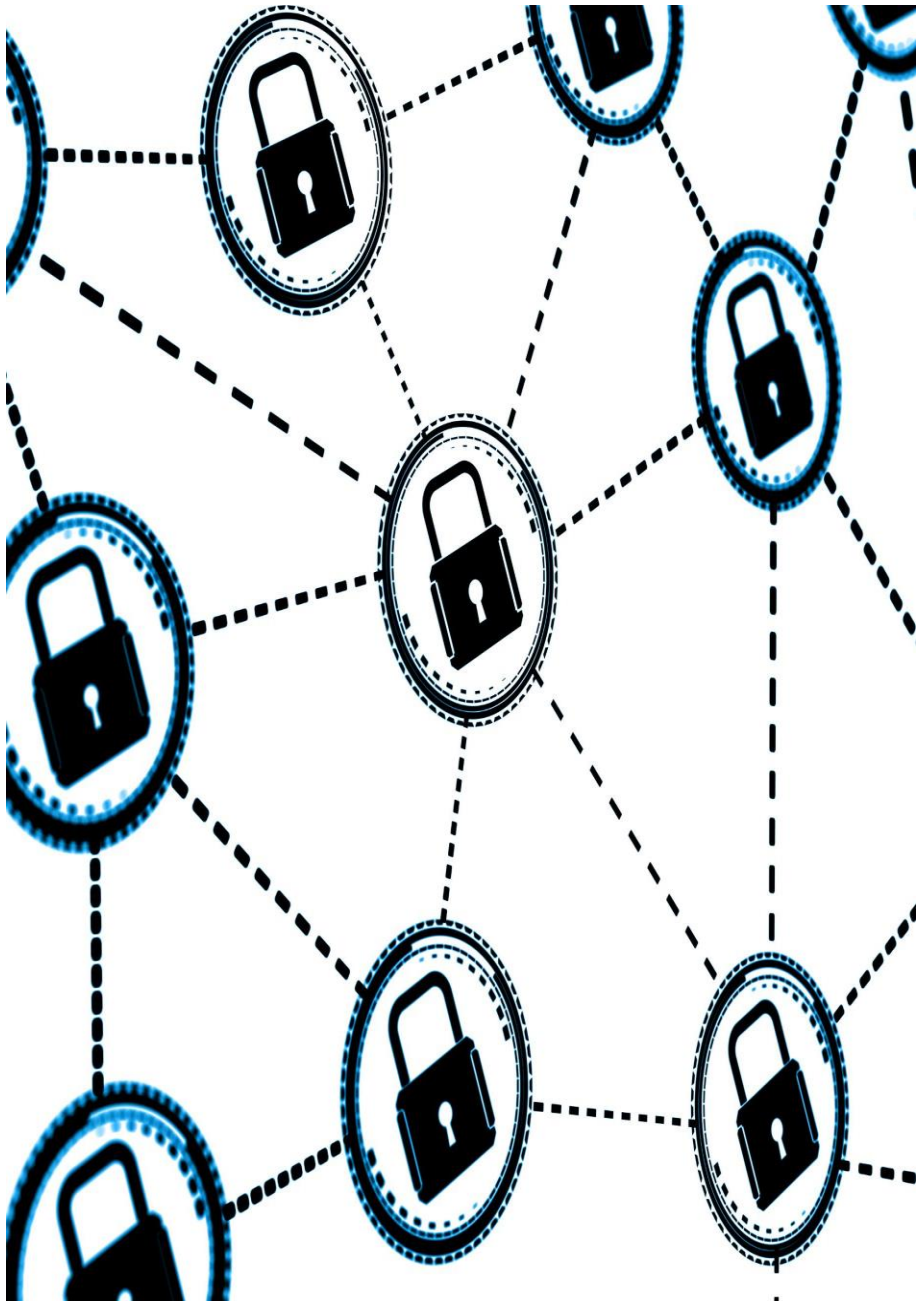
Benefícios de Design

O uso da delegação facilita a manutenção do código e melhora a clareza da estrutura do programa.

Exemplo Prático

Um objeto chama métodos de outro para realizar tarefas, demonstrando o padrão de delegação na prática.

Encapsulamento e sua relação com a delegação



Como o encapsulamento potencializa a delegação

Controle de Acesso Seguro

O encapsulamento controla o acesso a métodos e dados, garantindo que apenas componentes autorizados interajam.

Evita Efeitos Colaterais

Ao limitar o acesso, o encapsulamento previne alterações inesperadas no estado dos objetos, evitando falhas.

Preservação do Estado Interno

Encapsulamento mantém a integridade do estado interno dos objetos, facilitando uma delegação confiável.

Cenários onde delegação e encapsulamento atuam juntos

Distribuição de Responsabilidades

A delegação distribui tarefas entre componentes para melhorar organização e eficiência do sistema.

Proteção dos Componentes

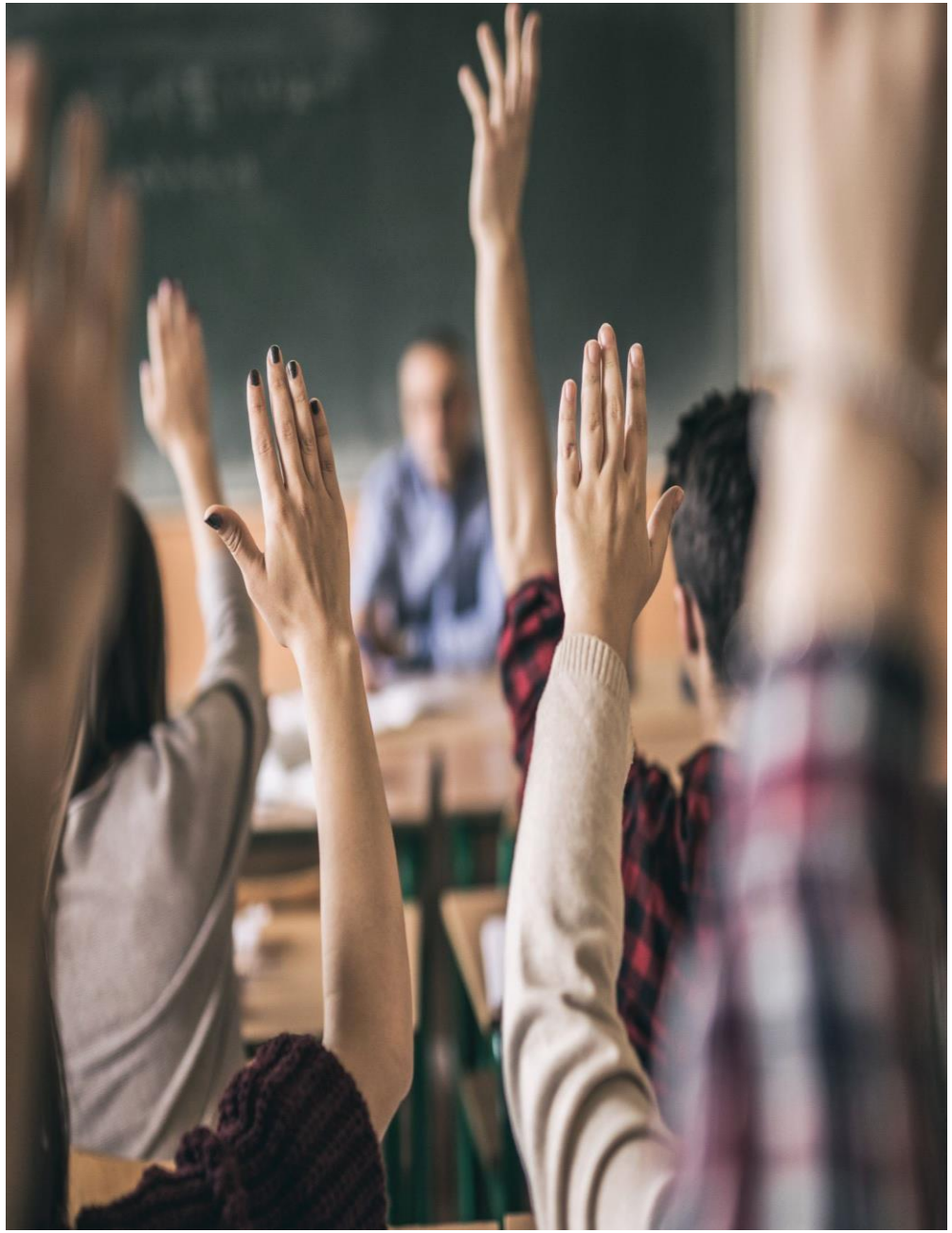
O encapsulamento protege os dados internos, mantendo a integridade e segurança dos componentes.

Manutenção Facilitada

A combinação de delegação e encapsulamento facilita a manutenção e evolução do sistema.

Conceito	Propósito Principal	Relação
Encapsulamento	Ocultar detalhes internos e proteger dados	Delegação usa interfaces públicas, respeitando o encapsulamento
Delegação	Reutilizar comportamento de outro objeto	Permite modularidade sem quebrar o encapsulamento

Aplicação prática: exemplo de código em Python utilizando delegação



Estrutura básica do código e explicação das classes

Classe de Delegação

Esta classe é responsável por delegar tarefas para outras classes que executarão as ações necessárias.

Classe de Execução

Esta classe executa as tarefas recebidas da classe de delegação, realizando as funções designadas.

Interação entre Classes

As duas classes interagem por meio da delegação, garantindo organização e divisão clara de responsabilidades.

Demonstração do uso de delegação no código Python

Conceito de Delegação

A delegação permite que uma classe utilize métodos de outra para realizar tarefas específicas, promovendo modularidade.

Interação entre Classes

A classe delegante chama métodos da classe delegada para executar funcionalidades específicas, facilitando o reuso de código.

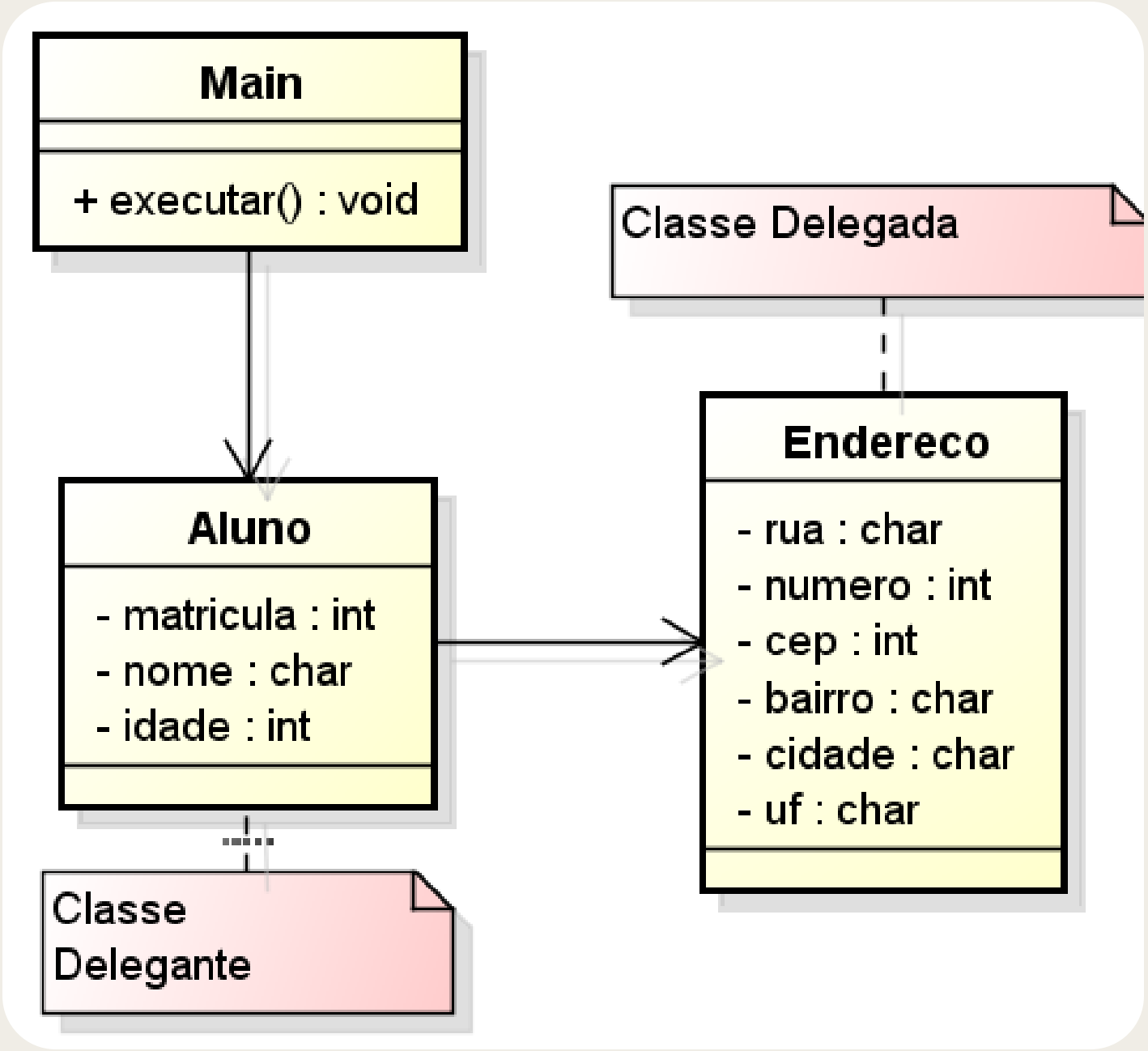
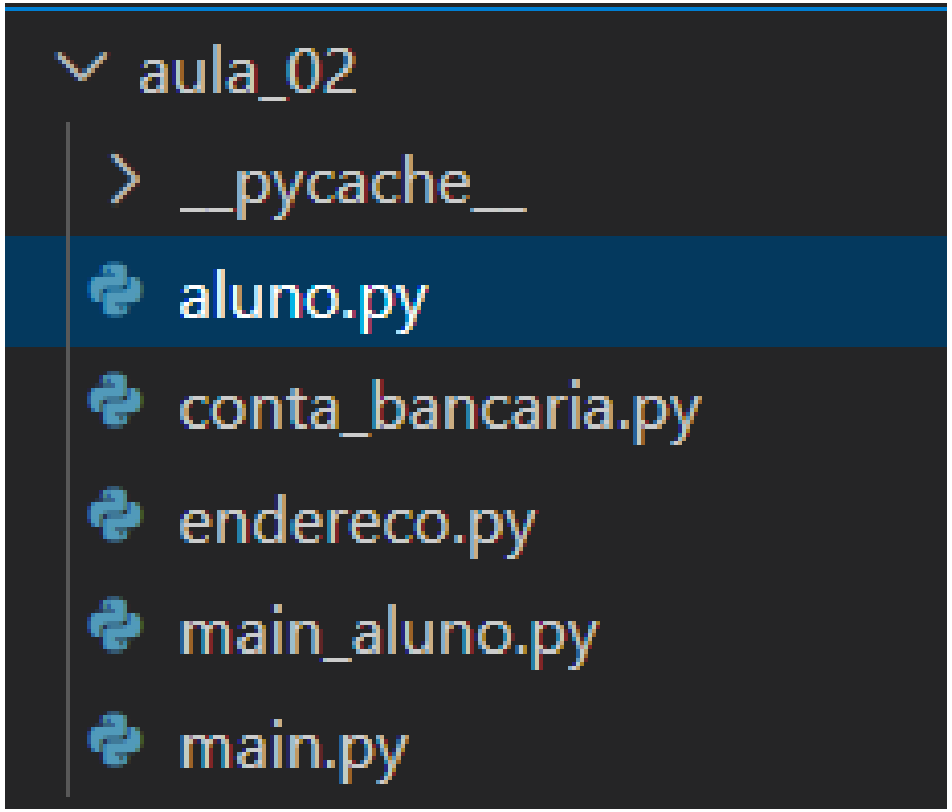


Diagrama de Classe

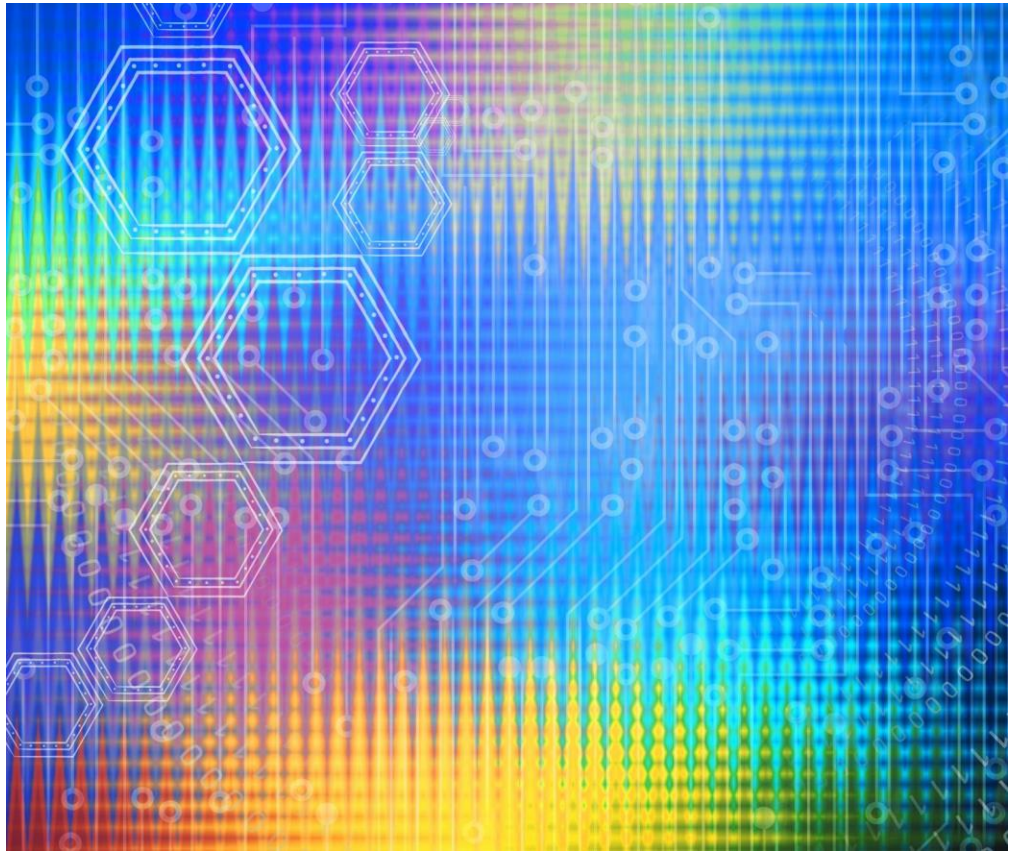
1. A classe Main não tem acesso aos atributos da classe Endereço diretamente, somente através da classe Aluno.
2. A classe Endereço fica encapsulada para a a classe Main, devido aplicação do conceito de Delegação.

Código em Python sobre Encapsulamento e Delegação

- O Código exemplo está no Projeto : aula_02



Análise dos resultados e benefícios do exemplo aplicado



Modularidade do Código

A delegação torna o código mais modular, facilitando a organização e o entendimento das funções.

Facilidade de Manutenção

A delegação simplifica a manutenção ao permitir alterações isoladas sem afetar todo o sistema.

Reutilização de Código

Promove a reutilização do código ao separar responsabilidades entre diferentes objetos.

Separação de Responsabilidades

A delegação assegura uma clara separação de responsabilidades entre objetos, melhorando a organização.

Conclusão

Princípio Fundamental

Encapsulamento é essencial na programação orientada a objetos para proteger dados e manter a integridade do software.

Benefícios em Python

Implementar encapsulamento corretamente em Python resulta em sistemas seguros, com manutenção facilitada e código de alta qualidade.

Conclusão

Importância da Delegação

Delegação permite dividir responsabilidades em objetos distintos, aumentando modularidade e clareza do código.

Papel do Encapsulamento

Encapsulamento protege dados internos, garantindo segurança e controlando acesso ao código.

Benefícios Gerais

Juntos, esses conceitos promovem código modular, seguro e reutilizável para design eficaz de sistemas.