



# Desenvolvimento de Sistemas

Aula 05 – Paradigmas e Sistemas de  
Tipagem

Lorrany B A Marim

**SENAI**



# Paradigmas Estruturado e Imperativo

A base da programação clássica

- **Foco no "Como":** O programador dita passo a passo o que o computador deve fazer. É uma sequência de instruções que alteram o estado do programa





# Paradigmas Estruturado e Imperativo

- **Fluxo de Execução:** A execução começa na primeira instrução e segue de cima para baixo, podendo ser alterada por desvios (funções) e repetições.
- **Estrutura:** O código é dividido em blocos lógicos usando sequência, decisão (if/else) e iteração (loops).





# Paradigmas Estruturado e Imperativo

- **Variáveis:** O uso de variáveis é central, funcionando como nomes que se referem a valores na memória que podem ser atualizados.

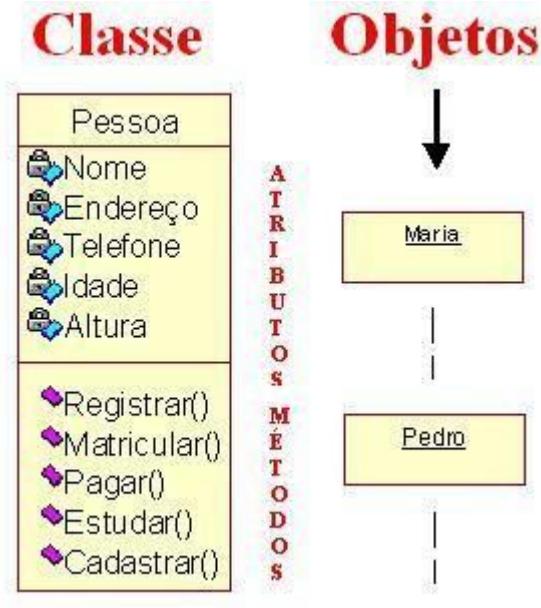




# Paradigma Orientado a Objetos (POO)

## Modelando o mundo real

- **Objetos como agentes:** Em vez de focar apenas em funções, a POO foca nos dados (objetos) e nas operações que eles podem realizar.
- **Classes e Instâncias:**
  - **Classe:** É a fábrica ou molde. Define os atributos (dados) e métodos (comportamentos).





# Paradigma Orientado a Objetos (POO)

Modelando o mundo real

- **Classes e Instâncias:**

- **Instância/Objeto:** É a concretização da classe. Se **Carro** é a classe, o **Fusca** na memória é o objeto.



classe



objeto

## Classe

Pessoa
🔒 Nome
🔒 Endereço
🔒 Telefone
🔒 Idade
🔒 Altura
◆ Registrar()
◆ Matricular()
◆ Pagar()
◆ Estudar()
◆ Cadastrar()

## Objetos

Maria

Pedro

A T R I B U T O S M É T O D O S

SENAI



# Paradigma Orientado a Objetos (POO)

- **Encapsulamento:** Os dados internos do objeto e sua representação ficam ocultos; o acesso é feito através de operações (métodos).
- **Herança:** Permite definir uma nova classe como uma versão modificada de uma existente, promovendo reutilização de código.
- **Polimorfismo:** Capacidade de substituir objetos com interfaces compatíveis em tempo de execução, permitindo que um mesmo comando atue de forma diferente dependendo do objeto.





# Paradigma Funcional

## Matemática e Imutabilidade

- **Funções Puras:** São funções que não têm efeitos colaterais. Elas recebem uma entrada e geram uma saída sem alterar variáveis globais ou o estado de outros objetos.
  - **Imutabilidade:** Evita-se a mudança de estado. Em vez de modificar uma lista, cria-se uma nova lista com as alterações desejadas.
- 
- **Vantagens:**
    - **Testabilidade:** Como a função sempre retorna o mesmo resultado para a mesma entrada, é muito fácil de testar.



# Paradigma Funcional

## Matemática e Imutabilidade

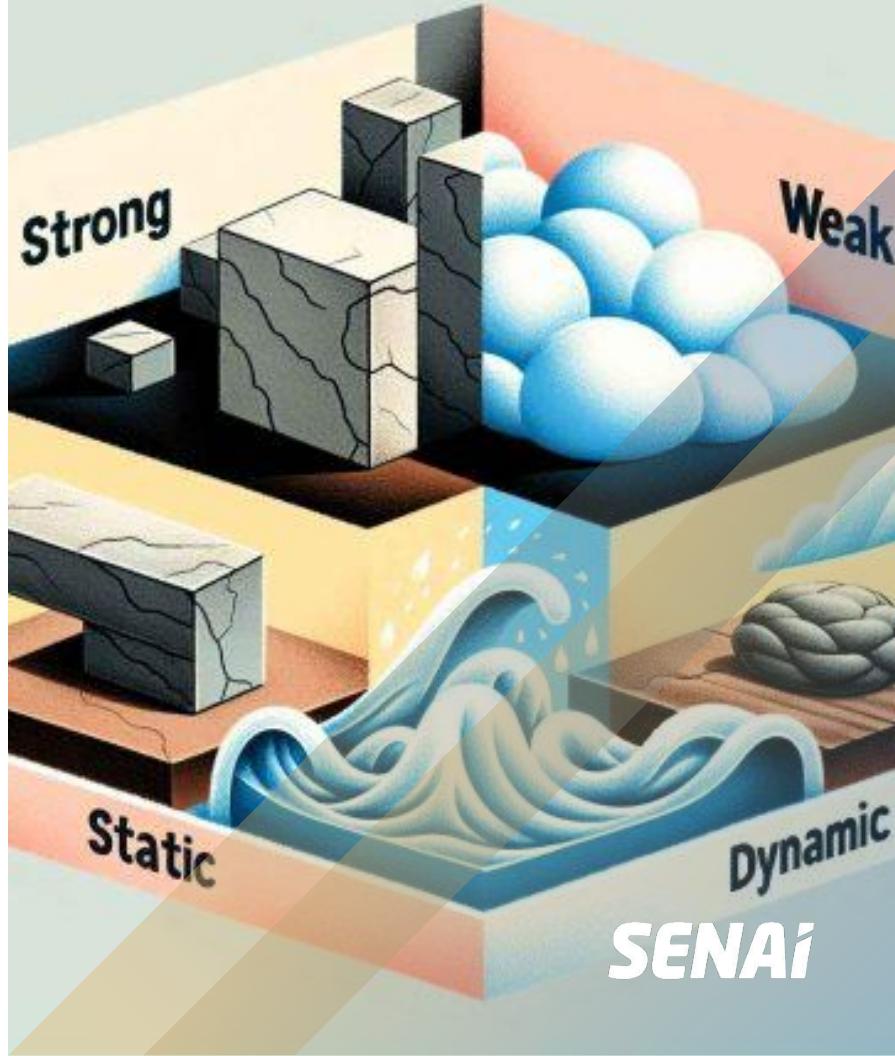
- **Vantagens:**
  - **Concorrência:** Facilita a execução paralela (várias threads), pois não há risco de uma função alterar o dado que outra está usando.
- **Ferramentas:** Uso intensivo de recursos como `map`, `filter` e comprehensões de lista.



# Tipagem Estática

## Verificação em Tempo de Compilação

- **Definição Explícita:** Em linguagens estáticas, as variáveis têm tipos definidos na declaração e isso é verificado antes do programa rodar (compilação).

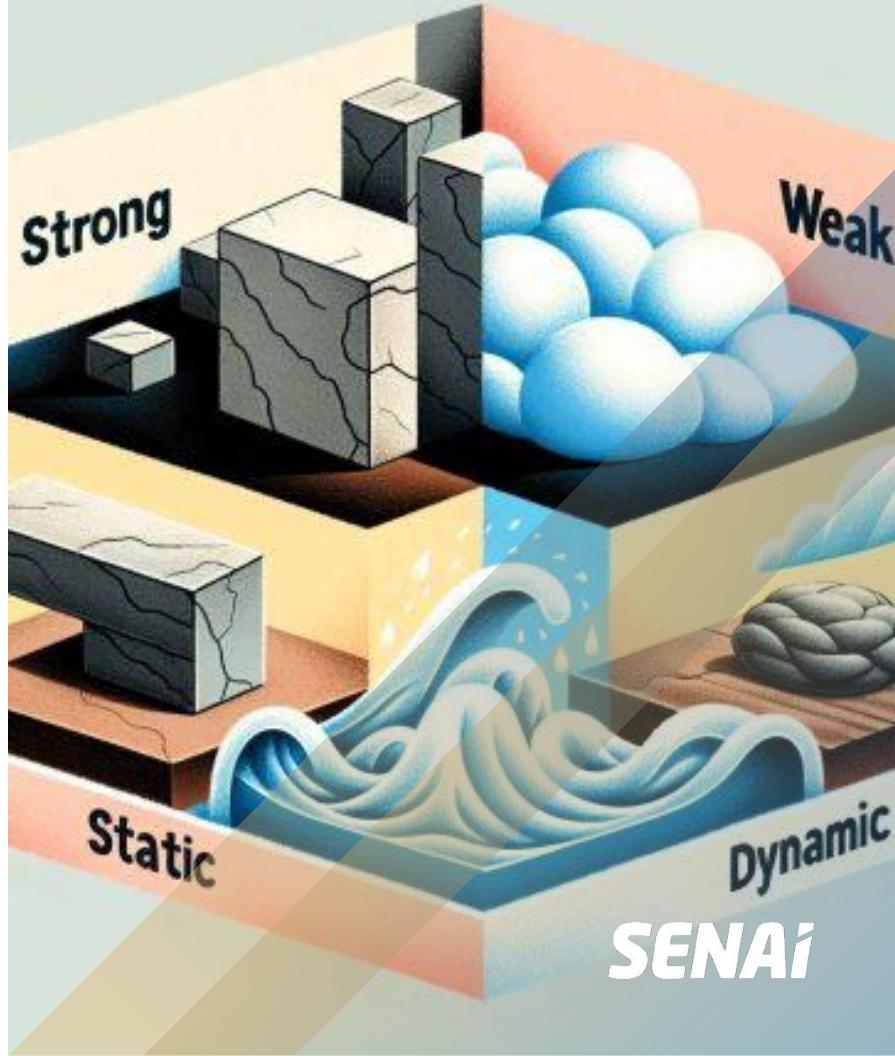




# Tipagem Estática

## Verificação em Tempo de Compilação

- **Segurança:** O compilador garante que você não está tentando usar um método que não existe naquele tipo de objeto [Gamma, 240].
- **Rigidez:** Uma vez declarada como `int`, a variável não pode receber uma `string`.

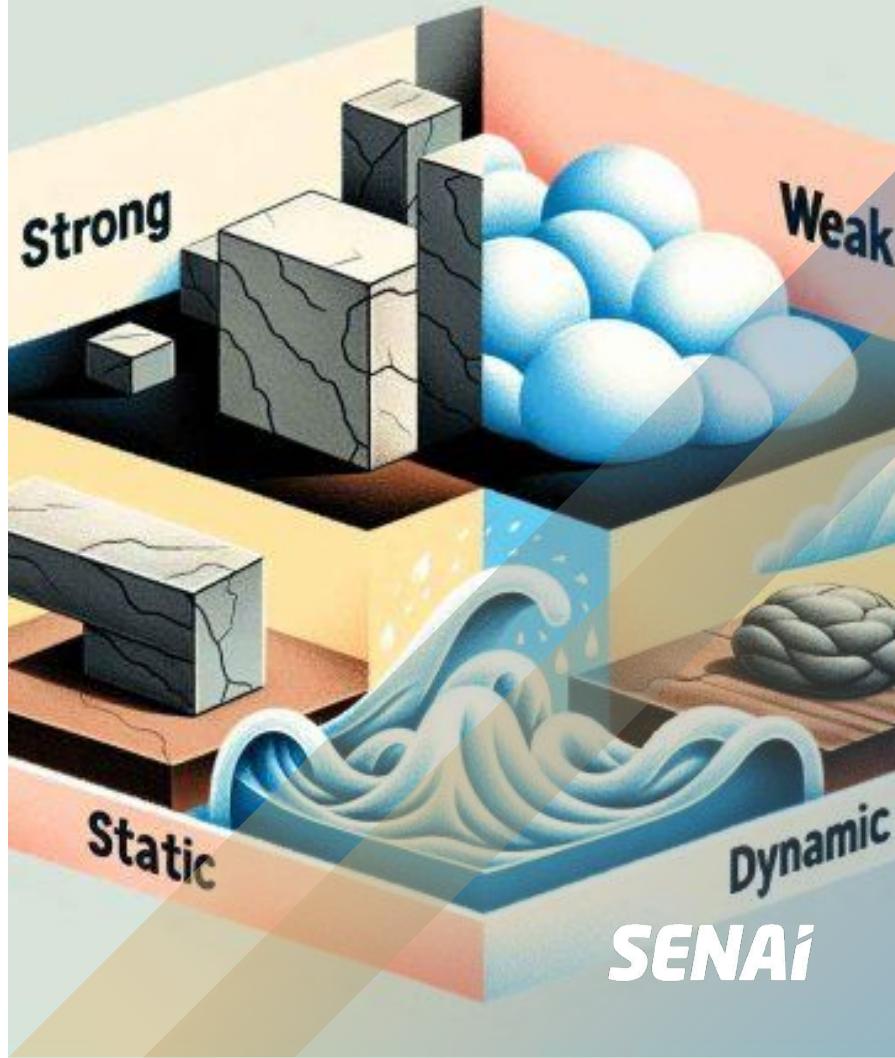




# Tipagem Estática

## Verificação em Tempo de Compilação

- **Interface:** A herança é essencial para permitir que objetos diferentes sejam tratados da mesma forma (polimorfismo), já que o tipo precisa ser compatível.

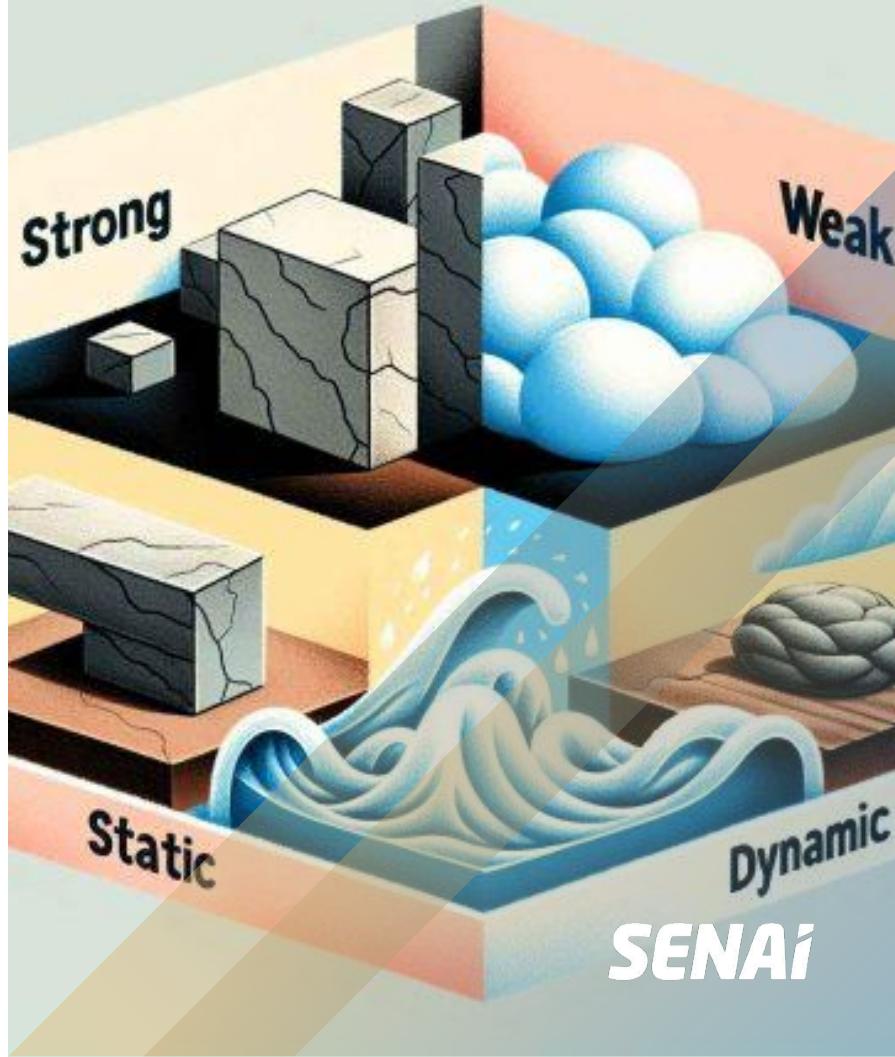




# Tipagem Dinâmica

## Verificação em Tempo de Execução

- **Flexibilidade:** As variáveis não têm tipo fixo; os valores (objetos) é que têm tipos. Uma variável pode referenciar um inteiro agora e uma string depois.

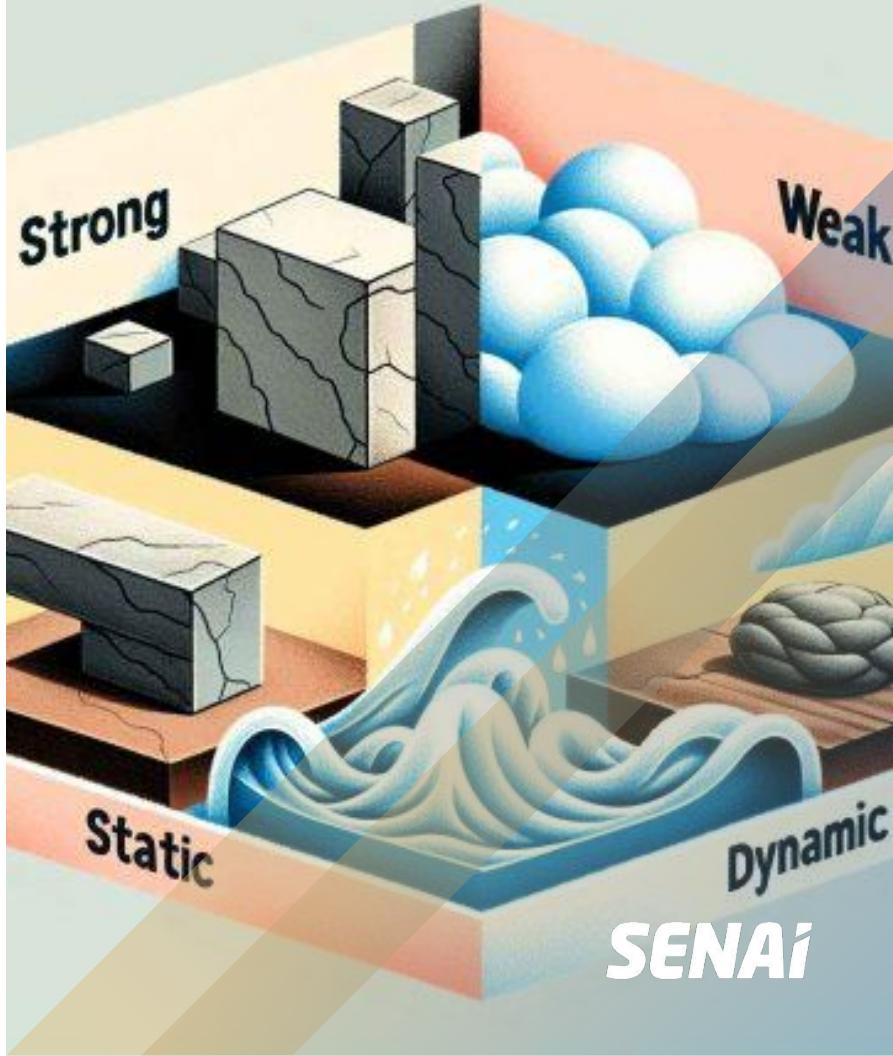




# Tipagem Dinâmica

## Verificação em Tempo de Execução

- **Verificação Tardia:** O interpretador verifica se a operação é válida apenas quando o código é executado.
- **Duck Typing:** "Se anda como pato e grasna como pato, é um pato".

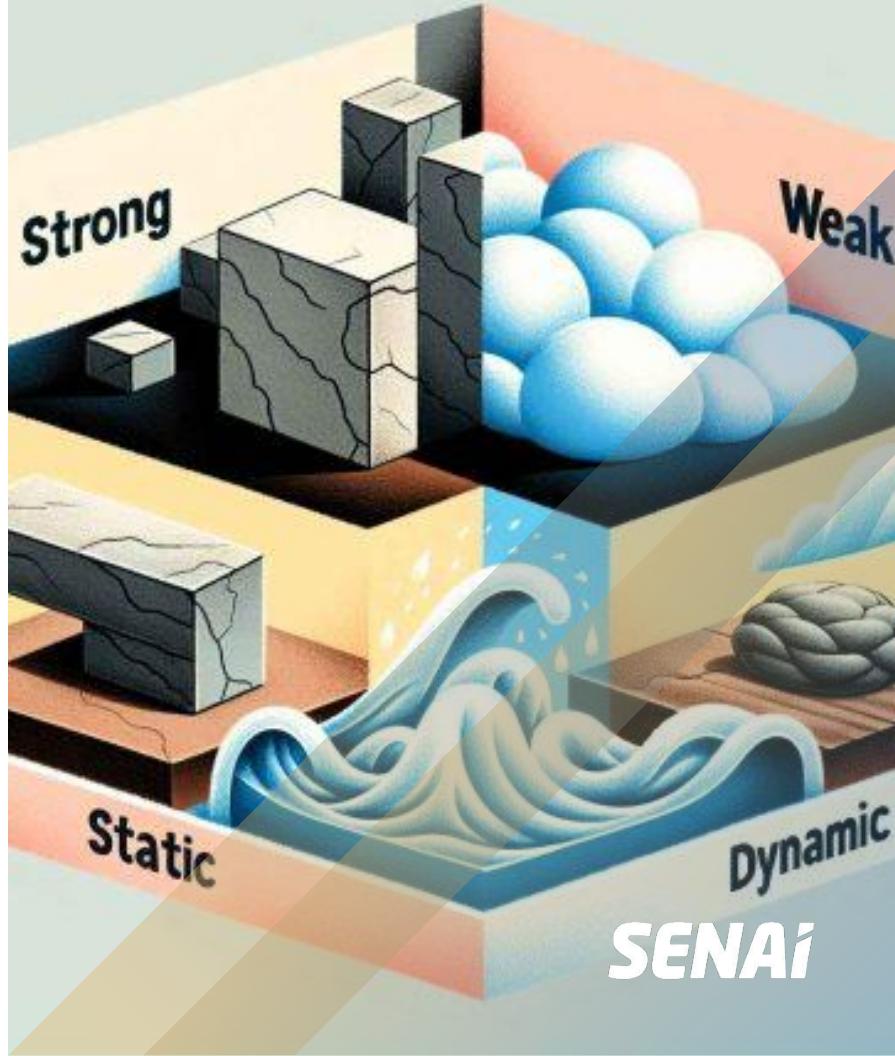




# Tipagem Dinâmica

## Verificação em Tempo de Execução

- **...Duck Typing:** Se o objeto possui o método chamado, ele funciona, independente da sua classe base.
- **Menos Código:** Não é necessário declarar tipos explicitamente, tornando o código mais conciso.

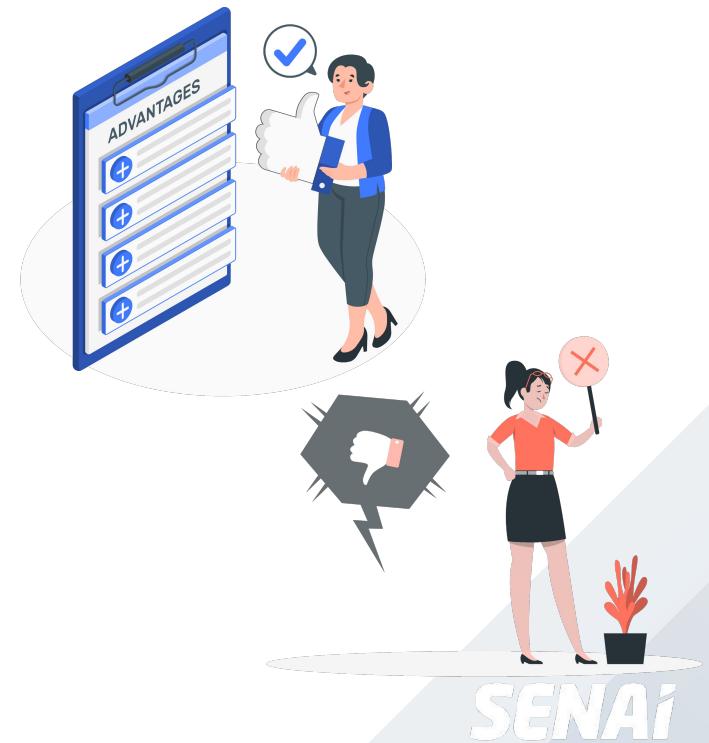




# Vantagens e Desvantagens

## Qual escolher?

- **Orientado a Objetos vs. Funcional:**
  - *OO* é excelente para modelar sistemas complexos e gerenciar estado, mas o estado compartilhado pode gerar bugs de concorrência.
  - *Funcional* é mais conciso e seguro para processamento de dados e paralelismo, mas pode ser menos intuitivo para quem está acostumado a pensar em "passo a passo".



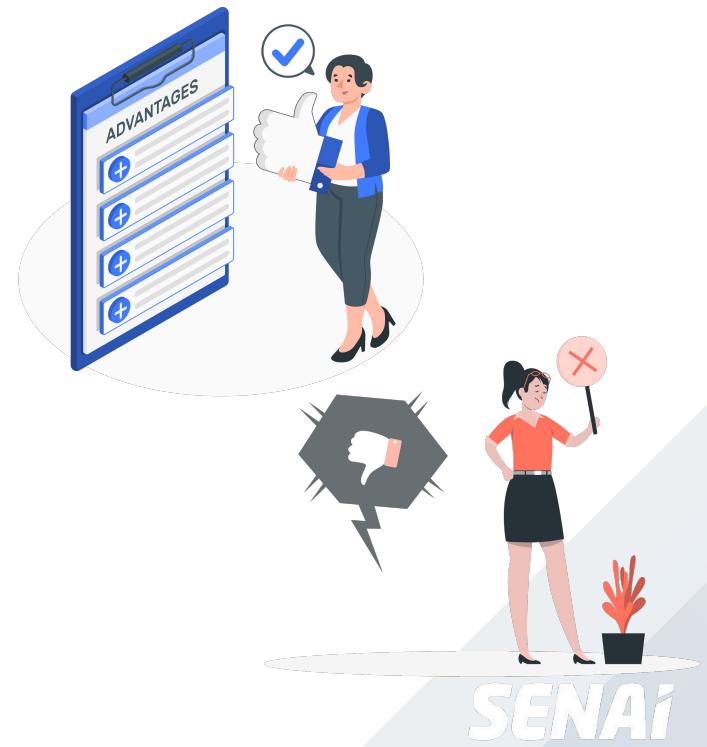


# Vantagens e Desvantagens

## Qual escolher?

- **Estática vs. Dinâmica:**

- *Estática* oferece segurança e ajuda a detectar erros cedo (na compilação), mas é mais verbosa.
- *Dinâmica* permite desenvolvimento rápido e flexibilidade (como criar tipos genéricos facilmente), mas erros de tipo podem aparecer só na mão do usuário (Runtime Error).



SENAI



# A Realidade do Mercado

- **Multiparadigma:** Linguagens modernas (como Python e Java atual) misturam paradigmas. Podemos usar objetos para estrutura e funções puras para processamento.
- **Não existe bala de prata:** Cada abordagem tem custos e benefícios. A escolha depende do problema a ser resolvido.
- **Boas Práticas:** Independente da tipagem, manter interfaces claras e separar a implementação da interface é fundamental para a manutenção do software.



# A Arquitetura MVC (Model – View – Controller)



# O que é Arquitetura de Software

## Organizando o sistema em alto nível

- **Definição:** A arquitetura trata da organização de um sistema em um nível de abstração mais alto do que classes individuais ou algoritmos.
- **Elementos:** Envolve a definição de pacotes, camadas, serviços e como eles se comunicam.
- **Objetivo:** Garantir que o sistema seja flexível, manutenível e escalável.
- **Padrões Arquiteturais:** O MVC é um dos padrões arquiteturais mais famosos, usado para separar responsabilidades e organizar o código.



# Introdução ao MVC

## A origem e o conceito

- **Sigla:** MVC significa *Model-View-Controller* (Modelo-Visão-Controlador).
- **Origem:** Foi criado originalmente para a linguagem Smalltalk-80 para construir interfaces de usuário.
- **Problema:** Antes do MVC, os sistemas misturavam dados, lógica de negócio e interface visual em um lugar só, dificultando a manutenção.
- **Solução:** O MVC separa esses objetos em três tipos distintos para aumentar a flexibilidade e a reutilização.



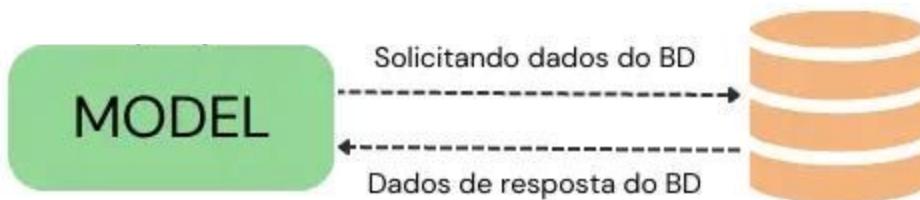
# O Modelo (Model)

## O coração da aplicação

- **Função:** É o objeto da aplicação que contém os dados e a lógica de negócio.
- **Independência:** O Modelo não sabe como seus dados serão apresentados. Ele foca apenas nas regras de negócio e no estado dos dados.
- **Comportamento:** Quando os dados mudam (ex: saldo de uma conta), o Modelo notifica as Visões de que houve uma alteração, mas sem saber quem são essas visões.
- **Boas Práticas:** Deve-se evitar que o Modelo dependa diretamente do banco de dados (ORM) sem uma camada de abstração adequada, para facilitar testes.



## BASE DE DADOS





# O Controlador (Controller)

## O maestro da interação

- **Função:** Define como a interface reage às entradas do usuário (teclado, mouse, cliques).
- **Intermediação:** O usuário não interage diretamente com o Modelo; ele interage com o Controlador, que interpreta a ação e atualiza o Modelo.
- **Flexibilidade:** O MVC permite mudar a forma como uma Visão responde ao usuário apenas trocando o Controlador, sem mudar a parte visual.
- **Padrão Strategy:** A relação entre Visão e Controlador é um exemplo do padrão de projeto *Strategy*.

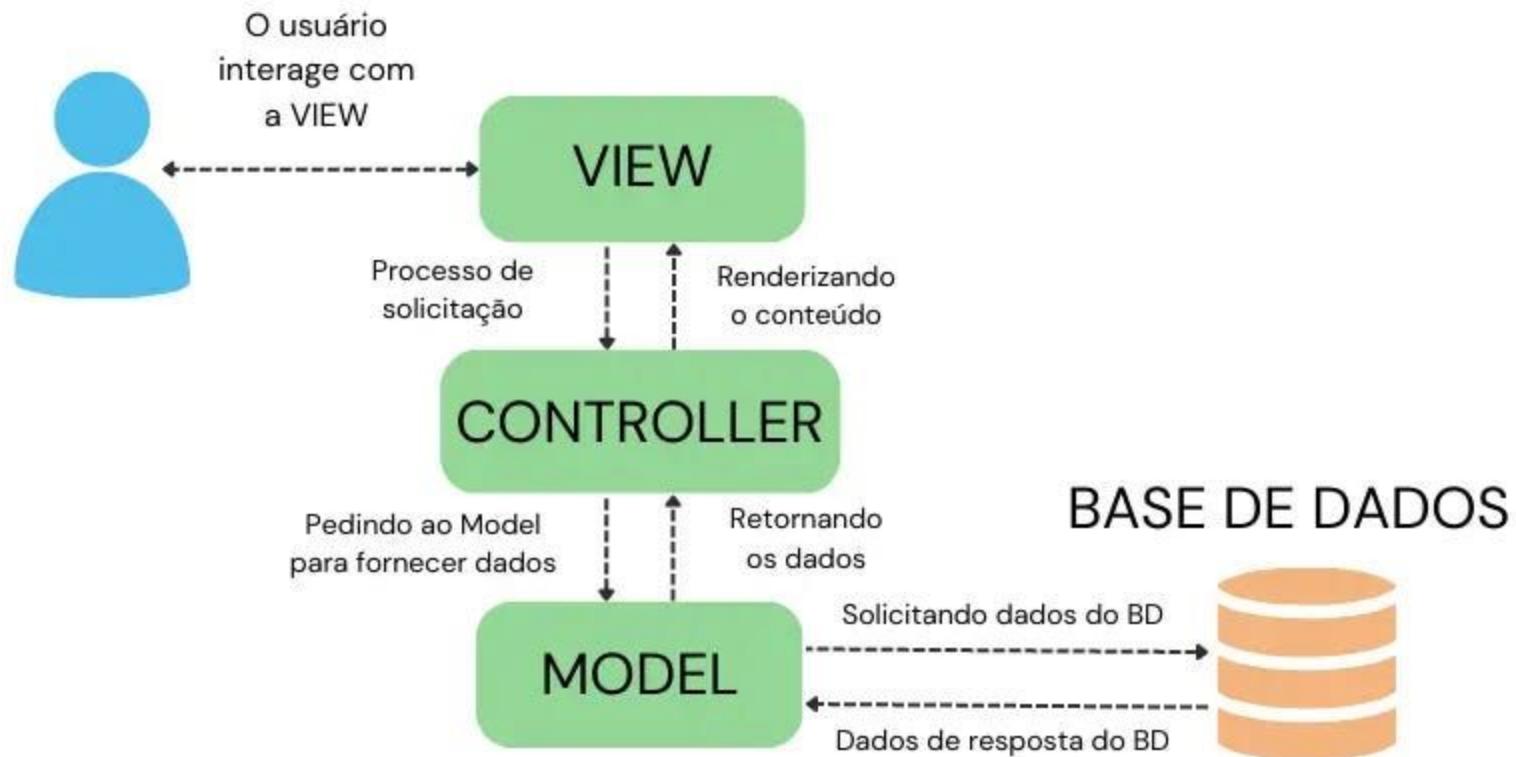




# A Visão (View)

## A interface com o usuário

- **Função:** É a apresentação visual na tela. Responsável por exibir os dados ao usuário.
- **Múltiplas Visões:** Um mesmo Modelo pode ter várias Visões diferentes (ex: uma planilha, um gráfico de barras e um gráfico de pizza mostrando os mesmos dados).
- **Atualização:** A Visão deve garantir que sua aparência reflita o estado atual do Modelo. Quando recebe uma notificação de mudança, a Visão se atualiza.
- **Estrutura:** Visões podem ser aninhadas (uma dentro da outra), usando o padrão *Composite*.





# O Fluxo de Comunicação no MVC

## Como as peças conversam

- **Separação:** O objetivo é separar objetos para que mudanças em um não afetem drasticamente o outro.
- **O Ciclo:**
  - O Usuário realiza uma ação (clique/entrada).
  - O **Controlador** recebe a ação e altera o Modelo.
  - O **Modelo** muda seu estado e notifica que mudou.
  - A **Visão** recebe a notificação e solicita os novos dados ao Modelo para se atualizar na tela.

**Padrão Observer:** Esse mecanismo de notificação (o Modelo avisa que mudou sem saber quem está ouvindo) implementa o padrão de projeto *Observer*.



# Por que o MVC é Essencial?

## A arquitetura da internet moderna

- **Adaptação:** Embora tenha nascido para Desktop, o MVC é a base da maioria dos frameworks Web modernos (como Django, Rails, Spring MVC).
- **Single Page Applications (SPA):** Frameworks modernos de Front-end (React, Vue, Angular) aplicam variações do MVC diretamente no navegador.
- **Manutenção:** Em sistemas Web complexos, separar o HTML (Visão) da Lógica de Banco de Dados (Modelo) e do Roteamento de URLs (Controlador) é vital para que o projeto não vire uma bagunça.



## Fontes Utilizadas

- Downey, A. B. "Pense em Python".
- Gamma, E. et al. "Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos".
- Danjou, J. "Python Levado a Sério".
- Valente, M. T. "Engenharia de Software Moderna".





Obrigado!  
**SENAI**

