Principais Paradigmas de Programação ‹›

Nos chamamos **Renato Augusto, Cauê Grande, Eduardo Cavazin e Vinícius Margonar.** Hoje, conversaremos sobre os paradigmas **Imperativo, POO, Funcional e Lógico.** Abordar suas Definições através de exemplos, apresentar seus pontos fortes, limitações e quais são seus cenários ideais.

Por que Estudar Paradigmas? <>

Moldam nosso pensamento

Organizam o código de formas distintas

Ampliam ferramentas

Escolha da abordagem ideal para cada problema

Paradigma Imperativo

Origem

O paradigma imperativo surgiu nos primórdios da computação, com linguagens como Assembly e Fortran. Ele reflete a lógica dos computadores ao executar comandos sequenciais que modificam o estado do sistema, focando em 'como fazer' por meio de instruções claras e controle explícito do fluxo.

Princípios

- Atribuição de valores: Alteração do estado por meio de variáveis.
- Loops (iteração): Repetição controlada de comandos.
- Controle de fluxo explícito: Uso de estruturas como if, else, switch, for, while, etc.
- Sequência de instruções: A ordem do código importa e é seguida linha por linha.

Exemplo

```c
#include <stdio.h>

int main() {
    // Atribuição: definição de variáveis e alocação de memória:
    int a, b;
    char op;

    // Comando sequencial:
    printf("Digite a operação (+, -, *, /): ");
    scanf(" %c", &op);   // Entrada de dados
    printf("Digite dois números: ");
    scanf("%d %d", &a, &b);

    // Controle de fluxo explícito com switch-case
    switch(op) {
        case '+':
            printf("Resultado: %d\n", a + b);
            break;
        case '-':
            printf("Resultado: %d\n", a - b);
            break;
        case '*':
            printf("Resultado: %d\n", a * b);
            break;
        case '/':
            // Controle de fluxo com if:
            if (b != 0)
                printf("Resultado: %d\n", a / b);
            else
                // Controle de erro:
                printf("Erro: divisão por zero\n");
            break;
        default:
            // Controle de fluxo para entrada não prevista:
            printf("Operação inválida\n");
    }

    return 0; // Fim do programa
}
```

Imperativo: Pontos Fortes e Limitações

Forças

- Simplicidade conceitual
- Ótimo desempenho em tarefas sequenciais
- Amplo suporte por compiladores e ferramentas

Limitações

- Pouca modularidade e reutilização de código
- Difícil manutenção em sistemas grandes
- Propenso a erros por causa do estado mutável

Cenários

- Programas com controle detalhado do fluxo
- Sistemas embarcados ou de baixo nível
- Algoritmos simples e computações lineares

Paradigma Orientado a Objetos <>

Origem

Anos 70/80, Simula67, Smalltalk, C++, Java.

Nasceu da necessidade de melhor modularidade e manutenibilidade.

Princípios

- Encapsulamento
- Herança
- Polimorfismo

Exemplo

Aqui temos um exemplo de uma Calculadora Financeira, que herda de uma calculadora Científica que herda de uma Calculadora.

```cpp
#ifndef CALCULADORA_HPP
#define CALCULADORA_HPP

class Calculadora {
protected:
    double resultado;

public:
    Calculadora();
    double somar(double a, double b);
    double subtrair(double a, double b);
    double multiplicar(double a, double b);
    double dividir(double a, double b);
    double getResultado() const;
};

#endif
```

```cpp
#ifndef CALCULADORA_CIENTIFICA_HPP
#define CALCULADORA_CIENTIFICA_HPP

#include "Calculadora.hpp"

class CalculadoraCientifica : public Calculadora {
public:
    double potencia(double base, double expoente);
    double raizQuadrada(double a);
};

#endif
```

```cpp
#ifndef CALCULADORA_FINANCEIRA_HPP
#define CALCULADORA_FINANCEIRA_HPP

#include "CalculadoraCientifica.hpp"
#include <stdexcept>

class CalculadoraFinanceira : public CalculadoraCientifica {
public:
    // Juros compostos (valor futuro)
    double calcularValorFuturo(double principal, double taxa, int periodos);

    // Valor presente
    double calcularValorPresente(double valorFuturo, double taxa, int periodos);

    // Pagamentos periódicos (PMT)
    double calcularPagamentoPeriodico(double principal, double taxaAnual, int periodos);

    // Taxa interna de retorno (retorno simples)
    double calcularTaxaRetorno(double investimentoInicial, double retornoTotal, int periodos);
};

#endif
```

POO: Pontos Fortes e Limitações

Forças

Reuso e estrutura modular natural

Limites

Hierarquias rígidas e boilerplate

Cenários

Ideal para GUIs e domínios ricos

Fraco para pipelines e funcional puro

Paradigma Funcional <>

Definição

Estilo de programação em que a computação é vista como avaliação de funções matemáticas, sem efeitos colaterais.

Princípios

- Imutabilidade
- Higher-order functions
- Lazy evaluation

O paradigma funcional trata a computação como avaliação de funções matemáticas, enfatizando imutabilidade e eliminação de efeitos colaterais.”

Como funciona <>

Imutabilidade

Cada operação retorna uma nova estrutura, sem alterar a original

Funções de Ordem Superior

Funções recebem e retornam novas funções

Lazy Evaluation

Cálculos são realizados apenas quando necessários, viabilizando listas infinitas.

Composição de Funções

Encadeamento em *pipelines* concisos ou via operadores “|>”

Exemplo Prático <>

```haskell
valores :: [Int]
valores = [1..10]

resultado :: Int
resultado =
  foldl' (+) 0
    $ map (*2)
    $ filter even valores
```

map (*2)

Dobra cada elemento

filter even

Mantém apenas os elementos pares

foldl ‘ (+) 0

Reduz a lista ao somatório de forma eficiente e sem a mutação de estado.

Funcional: Pontos Fortes e Cenários

Vantagens

Testabilidade e concorrência seguras

Código declarativo e conciso

Desafios

Curva de aprendizado íngreme

I/O via monads e overhead de GC

Cenários

Ótimo para ETL, DSLs e sistemas distribuídos

Fraco para UIs interativas e performance crítica

Paradigma Lógico

Origem

- A base do paradigma vem da lógica formal, um ramo da matemática.
- A primeira linguagem prática de programação lógica, nomeada Prolog, foi criada em 1972, por Alain Colmerauer e Robert Kowalski.
- Ganhou destaque na Europa durante os anos 1980 com o projeto japonês “Fifth Generation Computer Systems”.

Princípios

- Programação baseada em lógica matemática: Fatos e Regras.
- Expressividade sem controle explícito.
- Declaração de relações.
- Inferência automática de respostas por meio de unificação e backtracking.
- Resolução de problemas por busca de prova lógica.

Exemplo

CÓDIGO

```
:- initialization(calculadora).

% Definindo as operacoes
calcular(soma, X, Y, Resultado) :- Resultado is X + Y.
calcular(subtracao, X, Y, Resultado) :- Resultado is X - Y.
calcular(multiplicacao, X, Y, Resultado) :- Resultado is X * Y.
calcular(divisao, X, Y, Resultado) :- Y \= 0, Resultado is X / Y.
calcular(divisao, _, 0, 'Erro: divisao por zero').
calcular(potencia, X, Y, Resultado) :- Resultado is X ** Y.
calcular(raiz, X, _, Resultado) :- X >= 0, Resultado is sqrt(X).
calcular(raiz, X, _, 'Erro: raiz de numero negativo') :- X < 0.

% Funcao principal
calculadora :-
    write('Digite o primeiro numero: '),
    read(X),
    write('Digite o segundo numero (ou 0 se for raiz): '),
    read(Y),
    write('Escolha a operacao (soma, subtracao, multiplicacao,
divisao, potencia, raiz): '),
    read(Operacao),
    calcular(Operacao, X, Y, Resultado),
    write('Resultado: '), write(Resultado), nl,
    perguntar_novamente.

% Pergunta se o usuario quer continuar
perguntar_novamente :-
    write('Deseja realizar outra operacao? (s/n): '),
    read(Resposta),
    (Resposta == s -> calculadora ; write('Fim da calculadora.'),
nl).
```

INICIALIZAÇÃO

```
?- calculadora.
```

RESULTADO

 ?- calculadora.

Digite o primeiro numero:

10

Digite o segundo numero (ou 0 se for raiz):

2

Escolha a operacao (soma, subtracao, multiplicacao, divisao, potencia, raiz):

multiplicacao

Resultado: 20

Deseja realizar outra operacao? (s/n):

s

Digite o primeiro numero:

2

Digite o segundo numero (ou 0 se for raiz):

3

Escolha a operacao (soma, subtracao, multiplicacao, divisao, potencia, raiz):

potencia

Resultado: 8

Deseja realizar outra operacao? (s/n):

Lógico: Pontos Fortes e Limitações

Forças

- Inferência.
- Abstração Elevada.
- Facilidade para Raciocínios Complexos.
- Simplicidade para certos tipos de Problemas.

Limitações

- Performance.
- Escalabilidade
- Dificuldade de Controle de Fluxo da Execução.

Cenários

- Sistemas Especialistas
- Inteligência Artificial.
- Consultas dedutivas em BD.
- Processamento de Linguagem Natural (PLN).

Comparação Entre Paradigmas <>

Foi desenvolvido o mesmo projeto em cada um dos paradigmas de programação para possibilitar uma comparação mais técnica de como cada paradigma se comporta. O projeto em questão foi uma **calculadora.**

Comparação Entre Paradigmas

Critério	Prolog	C	C++	Haskell
Paradigma	**Lógica**	**Imperativa**	**Orientada a Objetos**	**Funcional**
Modularização	Predicados separados, mas sem encapsulamento	Funções livres; agrupamento por arquivo, sem abstrações	Classes e heranças	Funções Puras e *module Main;* separação clara entre cálculo e I/O
Tratamento de Erros	Rótulos em cláusulas(divisão por zero retorna átomo)	Condicionais que imprimem mensagens imediatamente	Lançamento de std::runtime_error e captura em camada de interface	Uso de *Maybe* pra sinalizar falhas sem exceções
I/O vs Lógica	I/O e lógica misturados em predicados	I/O e cálculo misturados nas mesmas funções	InterfaceCLI, cálculo nos objetos	isolados: calcular :: String -> String e main usa I/O
Extensibilidade	Adicionar operação -> nova cláusula calcular	Adicionar operação -> mais case no switch e loop	Derivar nova classe (ex: CalculadoraEstatistica) e estender hierarquia	Basta definir nova função pura e estender o case em calcular
Legibilidade	Sintaxe concisa para regras lógicas, mas curva de aprendizado alta para iniciantes	Código familiar, mas verboso(printf/scanf, loops e switch)	Estrutura clara de classes, porém boilerplate de headers e exceções	Código conciso e declarativo; punções de case diretas
Facilidade de Teste	Testes via chamadas no predicado; dificil isolar I/O	Testes em C requer mocks ou separar funções	Testar métodos individualmente; dependências em exceções	Funções puras permitem testes unitário e property-based(QuickCheck)

Conclusão <>

Entender a base dos paradigmas de programação é o que diferencia um verdadeiro **profissional de um amador,** principalmente quando olhamos para o contexto geral do mercado, é comum que se adote linguagens que sejam **multi-paradigmas**, contudo, ainda haverão situações onde será necessário uma linguagem mais especializadas, por isso a **importância** de estudar os paradigmas.

Perguntas?