UNIVERSIDADE FEDERAL
DE ALAGOAS

Questões (ênfase em OO, comparação com imperativo e lógico)

[1]Igor Mariano Alencar e Silva - 23112427,
[1]Jhony Wictor do Nascimento Santos - 23112167,
[1]Karleandro Santos da Silva - 23112207,
[1]Lucas Rosendo Farias - 23112728,
[1]Luís Gustavo Correia de Oliviera - 23112552
[1]Washington Medeiros Mazzone Gaia - 23112555

{igor.alencar, jhony.santos, karleandro.silva, lucas.farias, luis.correia, washington.gaia}@arapiraca.ufal.br

[1]Universidade Federal de Alagoas (UFAL), Campus Arapiraca — SEDE, Caixa Postal: 57309-005 – Arapiraca – AL – Brasil

Bacharelado em Ciência da Computação.

ABSTRACT. *This work is being developed by the students of the Computer Science course, class of 2025.1, in the Programming Language Paradigms course, under the guidance of Fernando Dantas. The activity is part of the evaluation criteria for the composition of the AB2 grade. The objective of this document is to present a comparative analysis between three programming paradigms: imperative, logic, and object-oriented (OO). In addition to the technical comparison, the study discusses code reuse strategies and the pedagogical path for teaching the paradigms, suggesting the sequence imperative → object-oriented → logic as the most effective.*

RESUMO. *Este trabalho está sendo desenvolvido pelos discentes do curso de Ciência da Computação, turma 2025.1, na disciplina de Paradigmas de Linguagem de Programação, sob a orientação do Fernando Dantas. A atividade faz parte dos critérios avaliativos para a composição da nota da AB2. O objetivo deste documento é apresentar uma análise comparativa entre três paradigmas de programação: imperativo, lógico e orientado à objetos (OO). Além da comparação técnica, o estudo discute estratégias de reutilização de código e a trilha pedagógica para ensino dos paradigmas, sugerindo a sequência imperativo → orientado a objetos → lógico como mais eficaz.*

1. **Problema em três paradigmas. Escolha um problema simples (ex.: calculadora de expressões aritméticas, gerenciador de tarefas) e descreva esboços de solução nos paradigmas imperativo, lógico e OO. Compare custos de implementação, legibilidade e extensibilidade.**

Para ilustrar as diferenças fundamentais entre os paradigmas de programação, é instrutivo aplicar cada um deles à resolução de um mesmo problema. A escolha de um problema adequado é crucial: ele deve ser simples o suficiente para não obscurecer os conceitos com detalhes de implementação, mas complexo o bastante para que as vantagens e desvantagens de cada abordagem se tornem evidentes.

O problema escolhido para esta análise comparativa é a criação de um gerenciador de tarefas com um conjunto mínimo de funcionalidades. O sistema deve ser capaz de executar as seguintes operações básicas:

a. Adicionar Tarefa: Permitir a criação de uma nova tarefa, identificada por uma
b. Listar Tarefas: Exibir todas as tarefas atualmente armazenadas, indicando seu status.
c. Marcar Tarefa como Concluída: Alterar o status de uma tarefa específica de "pendente" para "concluída".

O paradigma imperativo foca em descrever como uma tarefa deve ser executada. O programa é uma sequência explícita de comandos que manipulam o estado do sistema. A solução em C organiza-se em torno de estruturas de dados struct e funções que operam sobre elas.

a. Estrutura de dados e estado global em C:

```
// Definição da estrutura para uma tarefa
typedef struct {
    int id;
    char descricao;
    int concluida; // 0 para pendente, 1 para concluída
} Tarefa;

// Estado global da aplicação
Tarefa listaDeTarefas;
int contadorDeTarefas = 0;
```

b. Função para adicionar tarefa em C:

```
#include <stdio.h>
#include <string.h>

//... (definições da struct e variáveis globais acima)

void adicionarTarefa(const char* descricao) {
    if (contadorDeTarefas < 100) {
        listaDeTarefas.id = contadorDeTarefas + 1;
        strncpy(listaDeTarefas.descricao, descricao, 255);
        listaDeTarefas.descricao = '\0';
        listaDeTarefas.concluida = 0;
        contadorDeTarefas++;
        printf("Tarefa adicionada com sucesso.\n");
    } else {
        printf("A lista de tarefas está cheia.\n");
    }
```

```
}
```

O paradigma lógico descreve o que é verdade sobre o domínio do problema. A solução é um conjunto de fatos e regras, e a execução é uma consulta. As tarefas são representadas como fatos na base de conhecimento.

a. Fatos dinâmicos em Prolog:

```
% Declara que o predicado tarefa/3 pode ser modificado.
:- dynamic tarefa/3.
% Exemplo de fatos iniciais (ID, Descrição, Status).
% tarefa(1, 'Comprar leite', pendente).
% tarefa(2, 'Pagar contas', pendente).

% Regra para adicionar uma nova tarefa.
adicionar(Descricao) :-
    findall(ID, tarefa(ID, _, _), IDs),
    (IDs = -> NovoID is 1 ; max_list(IDs, MaxID), NovoID is
MaxID + 1),
    assertz(tarefa(NovoID, Descricao, pendente)).

% Regra para listar todas as tarefas pendentes.
listar_pendentes :-
    write('--- Tarefas Pendentes ---'), nl,
    tarefa(ID, Descricao, pendente),
    format('~w: ~w~n',),
    fail.
listar_pendentes. % Sucesso ao final.
```

A OO modela o problema como um sistema de objetos que interagem, encapsulando dados e comportamento. A solução é modelada através de classes que servem como "plantas" para os objetos.

b. Esboço da Solução Orientada a Objetos (em Java):

```
import java.util.ArrayList;
import java.util.List;

// Classe que modela uma única tarefa
class Tarefa {
    private String descricao;
    private boolean concluida;

    public Tarefa(String descricao) {
        this.descricao = descricao;
        this.concluida = false;
    }
    // Getters e Setters omitidos para brevidade...
    @Override
    public String toString() {
        return "Tarefa: " + descricao + " | Status: " +
               (concluida? "Concluída" : "Pendente");
    }
}

// Classe que gerencia a coleção de tarefas
class GerenciadorDeTarefas {
    private List<Tarefa> tarefas = new ArrayList<>();
```

```
    public void adicionarTarefa(String descricao) {
        this.tarefas.add(new Tarefa(descricao));
        System.out.println("Tarefa adicionada: " +
descricao);
    }

    public void listarTarefas() {
        System.out.println("--- Lista de Tarefas ---");
        for (Tarefa tarefa : tarefas) {
            System.out.println(tarefa);
        }
    }
}
```

Table 1. Tabela Comparativa de Paradigmas

Critério	Imperativo (C)	Lógico (Prolog)	Orientado a Objetos (Java)
Custo de Impl.	Baixo para problemas simples.	Médio; requer mudança de mentalidade.	Médio; maior boiler-plate inicial.
Legibilidade	Média; fluxo claro, mas dados e funções são implícitos.	Alta para consultas; complexa para gerenciar estado.	Alta; espelha a estrutura do domínio.
Extensibilidade	Baixa; mudanças se propagam por todo o código.	Média; requer alteração de fatos e regras.	Alta; encapsulamento minimiza o impacto.

Logo, esta comparação revela um padrão fundamental. O paradigma imperativo é otimizado para simplicidade e controle direto, mas sua eficácia diminui drasticamente à medida que a complexidade do estado do programa aumenta. A Orientada a Objetos, por outro lado, introduz uma complexidade inicial, mas essa "dívida" é paga com juros quando a complexidade do problema cresce, pois suas ferramentas (encapsulamento, herança e polimorfismo) são projetadas especificamente para gerenciar essa complexidade. O paradigma lógico, por sua vez, brilha em domínios onde a inferência e a busca de padrões são centrais. A escolha do paradigma não é sobre qual é o "melhor", mas sobre onde o problema se localiza no "eixo da complexidade".

2. Critérios de escolha de paradigma. Liste e justifique ao menos cinco critérios (não-técnicos e técnicos) para escolher um paradigma em um projeto real (ex.: equipe, domínio, ferramental, requisitos de desempenho, manutenção, compliance).

Os critérios de escolha de paradigma incluem, mas não estão limitados a: domínio do problema, requisitos não funcionais do projeto, manutenção e custo a longo prazo, ferramental disponível e experiência da equipe de desenvolvimento. Paradigmas diferentes são naturalmente mais adequados para resolver problemas diferentes. Um paradigma tem estrutura e abstrações únicas, que simplificam a implementação se utilizados no domínio correto. Por exemplo: Orientação a Objetos é ótimo para sistemas complexos, com muitas partes que interagem entre si, como sistemas financeiros; ou que lidam com entidades reais, como clientes e funcionários.

Ademais, requisitos não funcionais (uso de memória, tempo de resposta, etc.) podem restringir as opções de paradigma; cada sistema tem uma necessidade maior por um desempenho bom em um ou mais requisitos. Por exemplo: em sistemas embarcados, onde o controle sobre o hardware é crítico, paradigmas de baixo nível como o procedural são frequentemente preferidos por oferecer maior previsibilidade.

Contudo, a maior parte do ciclo de vida de um software é gasta em manutenção e evolução, não no desenvolvimento inicial. Um paradigma que promove código legível, modular e com baixo acoplamento tende a reduzir os custos a longo prazo. O paradigma Orientação a Objetos, por exemplo, foi amplamente adotado com a promessa de facilitar a manutenção através do encapsulamento, que protege partes do sistema contra modificações inesperadas.

No entanto, um paradigma não existe no vácuo, mas é apoiado por um ecossistema de linguagens, bibliotecas, frameworks, ferramentas de desenvolvimento (IDEs), depuradores e comunidades. Por isso a maturidade desse ferramental é importante; por mais que o paradigma seja uma solução perfeita para o problema, a ausência de ferramentas para acesso a banco de dados, comunicação web ou interface gráfica, por exemplo, tornaria o projeto inviável.

Por fim, o processo de desenvolvimento de um projeto também costuma ser mais suave quando a equipe é familiar com as ferramentas e paradigma escolhidos. Impor um paradigma totalmente novo leva a atrasos, baixa qualidade de código e frustração aos envolvidos.

3. Imperativo vs. OO em cenários reais. Dê dois cenários em que a abordagem imperativa é preferível à OO e dois em que OO é claramente superior. Explique por quê em cada caso.

Embora a análise teórica forneça uma base, a verdadeira validação de um paradigma de programação ocorre em sua aplicação a problemas do mundo real. A escolha entre abordagens como a imperativa e a orientada a objetos é, portanto, uma decisão pragmática, guiada pelas demandas específicas de cada cenário.

O paradigma imperativo, ou procedural, demonstra sua excelência em domínios onde o controle explícito e a eficiência são cruciais. No Cenário 1, o desenvolvimento de sistemas embarcados e drivers de baixo nível, o software opera em microcontroladores com severas restrições de recursos e requisitos de tempo real. Nesses ambientes, a programação procedural, tipicamente em C, é dominante por oferecer controle granular sobre a memória e previsibilidade de desempenho, minimizando o overhead que abstrações de alto nível introduziram. Já no Cenário 2, em scripts de automação e processamento de dados em lote (tarefas de curta duração, lineares e com propósito único), um script procedural é a expressão mais direta e eficiente da solução. A complexidade da orientação a objetos seria desnecessária e contraproducente, enquanto a rapidez de desenvolvimento e a clareza de uma sequência de passos são vantagens decisivas.

Em contrapartida, o paradigma orientado a objetos (OO) é claramente superior em cenários dominados pela complexidade, pela colaboração em larga escala e pela necessidade de manutenção a longo prazo. No Cenário 1, sistemas corporativos complexos como ERPs e CRMs, a OO é essencial para modelar processos de negócios intrincados, sendo desenvolvida por grandes equipes e possuindo um ciclo de vida de anos. Nesses casos, o encapsulamento gerencia a complexidade, enquanto a herança e o polimorfismo promovem a reutilização e a flexibilidade. No Cenário 2, a construção de frameworks de interface gráfica, o domínio se alinha perfeitamente ao

paradigma. Uma interface é uma composição de objetos (*Janela, Botão*), a herança modela a hierarquia de tipos (um CheckBox "é um" tipo de Botão), e o polimorfismo é crucial para o tratamento de eventos, permitindo que o sistema responda de forma específica a cada componente.

A jornada através dessa análise comparativa revela uma verdade fundamental na engenharia de software: não existe um "melhor" paradigma em absoluto. A avaliação de uma abordagem é intrinsecamente contextual, e a escolha ideal é um ato de engenharia que exige uma análise criteriosa do domínio do problema, das restrições do projeto e dos objetivos de longo prazo. O engenheiro de software moderno e eficaz não é um dogmático, mas sim um poliglota de paradigmas, capaz de pensar proceduralmente ao otimizar um driver, de pensar em objetos ao arquitetar um sistema corporativo e, talvez, de pensar funcionalmente ao transformar fluxos de dados. A crescente popularidade de linguagens multiparadigma, como Python, JavaScript e C#, é um testemunho dessa necessidade de flexibilidade, confirmando que a capacidade de empregar o paradigma certo, para o problema certo, no momento certo, representa o estado da arte no desenvolvimento de software.

4. Encapsulamento e manutenibilidade. Explique como o encapsulamento contribui para o isolamento de mudanças. Dê um exemplo de code smell que viola encapsulamento e proponha uma refatoração.

Encapsulamento é o princípio de esconder os detalhes internos de uma classe ou módulo e expor apenas o necessário por meio de interfaces bem definidas. Isso contribui para o isolamento de mudanças porque, se a implementação interna precisar ser alterada, o restante do sistema não precisa ser modificado desde que a interface pública permaneça a mesma. Dessa forma, o acoplamento entre partes do código é reduzido e a manutenibilidade aumenta, já que as mudanças ficam contidas dentro de uma unidade específica em vez de se espalharem pelo sistema inteiro.

Um exemplo de code smell que viola o encapsulamento é o **Data Class**, que aparece quando uma classe serve primariamente como um recipiente para dados sem ter nenhum comportamento real, contendo apenas dados e métodos de acesso. Ainda que os dados não sejam públicos, são expostos por getters/setters que não fazem nenhum trabalho relevante além disso, o que torna a classe vulnerável a ser manipulada por fora sem nenhuma regra ou lógica interna.

Refatorar esse code smell significa dar à classe o comportamento que deveria ter. É necessário identificar quem está acessando os atributos dela, mover o comportamento destes para dentro da Data Class, e encapsular o estado, tornando os atributos privados e expondo apenas métodos que deveriam estar expostos. Por fim, substituir getters/setters triviais por métodos com intenção clara.

5. Herança: uso e "abuso'. Discuta benefícios e riscos do uso intensivo de herança. Apresente uma alternativa baseada em composição para um caso em que herança cria acoplamento ou fragilidade.

Herança é um mecanismo de reaproveitamento de código em linguagens orientadas a objetos, no qual uma classe filha (subclasse) herda atributos e métodos de uma classe pai (superclasse). Tem como benefícios:

1. Reuso de código, que evita duplicações de atributos e métodos comuns;
2. Organização Hierárquica, que facilita modelar domínios.

3. Polimorfismo, que torna o código mais flexível;

   Contudo, o abuso da herança, ou seja, o uso intensivo dela, tem como riscos:
4. Alto Acoplamento, em que subclasses ficam dependentes da implementação de superclasses;
5. Fragilidade Estrutural, em que hierarquias profundas dificultam manutenção e aumentam a complexidade;
6. Rigidez, que é a reutilização forçada. Nem sempre a relação "é-um" representa corretamente o domínio;

Em vez de criar hierarquias rígidas, é possível usar **composição**, onde uma classe é formada por outras classes. Isso promove flexibilidade e baixo acoplamento.

Exemplo de problema com Herança:

```
class Veiculo {
    void mover() { System.out.println("O veículo está se
movendo"); }
}

class Carro extends Veiculo {
    void abrirPorta() { System.out.println("Abrindo a porta do
carro"); }
}

class Bicicleta extends Veiculo {
    void pedalar() { System.out.println("Pedalando a
bicicleta"); }
}

Se quisermos criar um Carro voador, por exemplo, usar herança
geraria problemas, tendo em vista que criar VeiculoVoador
quebraria a hierarquia.

Exemplo de Solução com Composição:
class Motor {
    void ligar() { System.out.println("Motor ligado!"); }
}

class Asa {
    void voar() { System.out.println("Voando!"); }
}

class Carro {
    private Motor motor = new Motor();
    void mover() { motor.ligar(); }
    void abrirPorta() { System.out.println("Abrindo porta do
carro"); }
}

class CarroVoador {
    private Motor motor = new Motor();
    private Asa asa = new Asa();

    void mover() { motor.ligar(); }
    void voar() { asa.voar(); }
}
```

Logo, conclui-se que a herança é útil para reduzir duplicação e organizar hierarquias, mas seu uso excessivo cria acoplamento e fragilidade. Onde, sua composição é uma alternativa mais flexível, permitindo combinar comportamentos sem precisar recorrer a hierarquias complexas.

6. **Modelagem OO crítica. Modele um domínio simples (ex.: biblioteca, locadora ou e-commerce) com um pequeno diagrama de classes (pode ser esquemático). Aponte dois pontos de má modelagem frequentes e como corrigi-los.**

O domínio modelado representa um sistema simples de E-commerce onde um cliente pode realizar múltiplos pedidos. Cada pedido é composto por um ou mais itens (*ItemPedido*), que se referem a um produto específico e registram a quantidade e o preço no momento da transação.

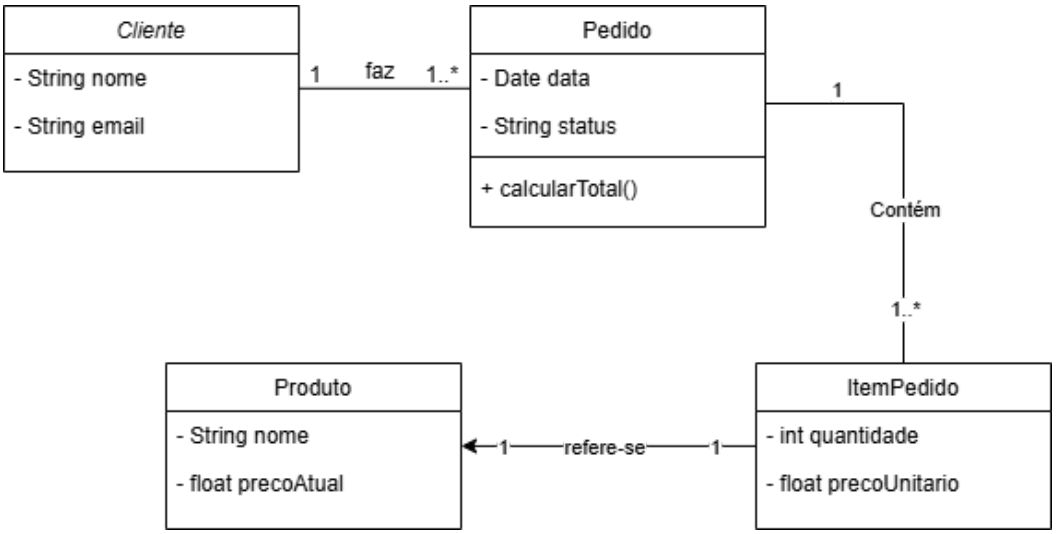

Figura 1. Representação de Domínio Simples

Um erro comum é atribuir responsabilidades excessivas a uma única classe, violando o Princípio da Responsabilidade Única, conhecido como SRP, resultando em baixa coesão. A classe Pedido é modificada para gerenciar, além de seus próprios dados, a lógica de processamento de pagamentos, comunicação com sistemas de logística e persistência em banco de dados. Ela se torna uma "Classe Deus", centralizando funcionalidades que não lhe pertencem.

No entanto, a classe fica difícil de manter (qualquer mudança em sistemas externos afeta ela), complexa para testar (exige a configuração de múltiplos serviços) e o código não é reutilizável (a lógica de pagamento está presa dentro da lógica de pedido).

Portanto, a solução é delegar as responsabilidades externas a classes especialistas. A classe Pedido deve se preocupar apenas com as regras de negócio intrínsecas a um pedido. Lógicas de pagamento, logística e persistência devem ser abstraídas em suas próprias classes (*ServicoDePagamento, ServicoDeLogistica, PedidoRepository*), resultando em um sistema desacoplado, coeso e modular.

7. **Comparação: OO vs. Lógico em busca/inferência. Explique diferenças conceituais ao expressar busca e restrições em OO (objetos/métodos) versus no**

paradigma lógico (regras, unificação e retrocesso). No paradigma lógico, unificação é o processo de verificar se dois termos podem ser tornados iguais por meio de substituições de variáveis, e retrocesso (backtracking) é a estratégia de explorar alternativas automaticamente quando uma tentativa de prova falha. Dê um micro exemplo em Prolog que utilize unificação e retrocesso, e um exemplo equivalente em uma linguagem OO (Java, Python, etc.) para resolver o mesmo problema.

No paradigma orientado a objetos, a lógica do programa é encapsulada em objetos que contêm tanto dados (atributos) quanto comportamentos (métodos). A resolução de um problema de busca ou a verificação de restrições é implementada de forma explícita pelo programador, que deve construir o algoritmo passo a passo, geralmente utilizando estruturas de controle como laços (for, while) e condicionais (if). É nítido o contraste, no paradigma lógico, a lógica é expressa em termos de fatos (o que é verdade) e regras (relações entre fatos), sem detalhar o "como" executar a busca. A solução de problemas é delegada a um motor de inferência que utiliza mecanismos poderosos como a unificação, processo de casar termos através da substituição de variáveis, e o retrocesso (backtracking), estratégia de explorar alternativas automaticamente quando uma tentativa falha.

1. **Exemplo em Prolog (Lógico):**
    a. **Problema:** queremos saber quem é pai de quem e se uma pessoa é ancestral de outra.

```
% Fatos
pai(joao, maria).
pai(maria, ana).
pai(ana, carlos).
% Regra: X é ancestral de Y se X é pai de Y
ancestral(X, Y) :- pai(X, Y).
% Ou se X é pai de Z e Z é ancestral de Y (recursão +
backtracking)
ancestral(X, Y) :- pai(X, Z), ancestral(Z, Y).
```

    b. **Consulta(Query):**

```
?- ancestral(joao, carlos).
```

    c. **Funcionamento:**
        i. Unificação: prolog tenta casar ancestral(joao, carlos) com as regras;
        ii. Retrocesso: ao não encontrar diretamente, ele volta e tenta a regra recursiva, explorando alternativas até provar que joao é ancestral de carlos;

    Resposta: **true.**

    d. **Exemplo Equivalente em OO com Python:**

```
class Pessoa:
    def __init__(self, nome):
        self.nome = nome
        self.filhos = []
    def adicionar_filho(self, filho):
```

```
            self.filhos.append(filho)
        def is_ancestral(self, pessoa):
            # busca explícita (recursão)
            if pessoa in self.filhos:
                return True
            for filho in self.filhos:
                if filho.is_ancestral(pessoa):
                    return True
            return False

    # Construção da árvore
    joao = Pessoa("João")
    maria = Pessoa("Maria")
    ana = Pessoa("Ana")
    carlos = Pessoa("Carlos")
    joao.adicionar_filho(maria)
    maria.adicionar_filho(ana)
    ana.adicionar_filho(carlos)
    # Consulta
    print(joao.is_ancestral(carlos))  # True
```

Portanto, desse modo, pode-se concluir que em OO, a resolução de busca e restrições depende do programador descrever passo a passo o algoritmo. Além disso, é possível estabelecer que no paradigma lógico, o programador apenas descreve relações e o mecanismo de inferência trata da busca usando unificação e backtracking.

8. Quando OO não é a melhor escolha? Dê três exemplos de problemas/domínios em que OO tende a ser inadequada ou excessiva e proponha alternativas (imperativo estruturado, lógico).

Orientação a objetos é ótima quando você tem uma variedade de coisas que se comportam de formas diferentes, mas compartilham uma estrutura comum. Ela passa a ser “ruim” nessas seguintes situações:

No primeiro exemplo, um processamento intensivo de dados científicos, em aplicações de computação científica (simulações físicas, álgebra linear, processamento de sinais), a eficiência computacional e o controle de memória são críticos. OO introduz overheads como indireção de vtable, alocações dinâmicas frequentes e pobre localidade de cache, impactando performance.

a. Alternativa em Paradigma Imperativo Estruturado (C/Fortran):

Cálculo das diferenças finitas em Fortran (exemplo prático)

```
subroutine finite_difference(u, n, dt, dx)
    real, dimension(n) :: u
    real :: dt, dx
    integer :: n, i

    do i = 2, n-1
u(i) = u(i) + dt * (u(i+1) - 2*u(i) + u(i-1)) / dx**2
    end do
end subroutine
```

Jack Dongarra, enfatiza que "em aplicações numéricas, a eficiência frequentemente requer operações diretas sobre arrays contíguos, minimizando abstrações" (Source: SIAM Review, 2020). O estilo procedural evita custos de OO e

permite otimizações agressivas.

Enquanto que, no segundo exemplo, um sistemas especialistas baseados em regras domínios como diagnósticos médicos ou configuração de sistemas complexos envolvem centenas de regras lógicas interdependentes. OO força a modelagem como objetos com comportamento, resultando em hierarquias de classes rígidas e código espalhado, dificultando a manutenção e adaptação das regras.

b. Alternativa em Paradigma Lógico (Prolog):

```
% Sistema de diagnóstico médico em Prolog

diagnostico(febre_reumatica) :-
    paciente_apresenta(febre),
    paciente_apresenta(artrite),
    paciente_apresenta(cardite),
    idade_paciente(Idade), Idade < 25.

diagnostico(artrite_idiopatica) :-
    paciente_apresenta(artrite),
    duracao_sintomas(Duracao), Duracao > 6_semanas,
    nao(diagnostico(febre_reumatica)).
```

Robert Kowalski, no artigo *"Algorithm = Logic + Control"* (1979), argumenta que "a programação em lógica separa o conhecimento lógico (as regras) do controle (a execução), permitindo que especialistas do domínio expressem conhecimento diretamente". Isso é mais natural que forçar regras em objetos.

O terceiro exemplo, o desenvolvimento de scripts e utilitários de sistema, para tarefas como processamento de texto, automação de sistema ou prototipagem rápida, a cerimônia de OO (definição de classes, hierarquias, encapsulamento) é excessiva. O overhead de desenvolvimento não compensa para problemas essencialmente sequenciais.

c. Alternativa em Paradigma Imperativo Estruturado (Python/Shell):

```
# Exemplo: Processamento de logs em Shell
grep "ERROR" system.log | \
awk '{print $1, $2, $5}' | \
sort | \
uniq -c | \
sort -nr > error_summary.txt

# Versão em Python estruturado
import sys
from collections import Counter
errors = Counter()
for line in sys.stdin:
    if "ERROR" in line:
        parts = line.split()
        error_code = parts[4]
        errors[error_code] += 1
for error, count in errors.most_common():
    print(f"{error}: {count}")
```

Fundamentado por Eric S. Raymond, em *The Art of Unix Programming*, defende que "para problemas de manipulação de texto e filtragem de dados, ferramentas especializadas e scripts lineares frequentemente superam abordagens OO

em simplicidade e eficiência de desenvolvimento". Logo, conclui-se que, a escolha do paradigma deve alinhar-se à natureza do problema. OO é poderosa para modelagem de domínios complexos com estado e comportamento, mas torna-se inadequada quando a eficiência computacional, a expressividade lógica ou a simplicidade sequencial são prioritárias.

9. Interoperabilidade de paradigmas. Mostre um caso realista de paradigmas mistos (ex.: núcleo algorítmico imperativo, regras em Prolog, orquestração OO). Explique benefícios e dificuldades de integração (tipagem, depuração).

Um caso seria um sistema de análise de risco de crédito, onde um banco que precisa automatizar a decisão de aprovar ou negar um pedido de empréstimo. O sistema deve ser rápido, capaz de processar grandes volumes de dados e aplicar regras de negócio que mudam constantemente. Para construir o sistema, dividimos as responsabilidades entre dois componentes, cada um baseado em um paradigma diferente: Orquestração Orientada a Objetos (O Maestro), O Motor de Regras Lógico (O Especialista/Prolog).

Na Orquestração Orientada a Objetos (OO), é a camada principal da aplicação, responsável por gerenciar o fluxo de dados e modelar o domínio do problema, uma das suas responsabilidades:

1. Controlar a sequência de operações: recebe a requisição, coordena a chamada aos outros componentes e formula a resposta final.
2. Criar entidades do mundo real como *Cliente*, *PedidoEmprestimo*, e *ContaBancaria* são representadas como classes. Essas classes encapsulam dados (os atributos) e comportamentos (os métodos), tornando o sistema modular e fácil de manter.
3. Interagir com bancos de dados e outras APIs.

Já o Motor de Regras Lógico, após o cálculo do score, o sistema precisa aplicar as políticas de crédito do banco. Essas políticas são frequentemente complexas, cheias de exceções e sujeitas a mudanças frequentes por parte da equipe de negócios e uma das suas responsabilidades:

4. Determinar o resultado final (*aprovado, negado, análiseManual*) com base em um conjunto de fatos (o score do cliente, sua renda, idade, etc.) e regras predefinidas.
5. Permitir que as regras de negócio sejam expressas de forma clara, concisa e, crucialmente, separada do código principal da aplicação.

Dessa forma, um dos benefícios é que cada paradigma é usado em sua área de maior força, resultando em um código mais limpo, eficiente e expressivo para cada parte do sistema. O uso do paradigma imperativo para as tarefas computacionalmente "pesadas" garante que o sistema atenda aos requisitos de velocidade, enquanto o resto da aplicação pode ser desenvolvido com linguagens de maior produtividade.

Entretanto, uma das maiores dificuldades são a comunicação entre componentes de paradigmas diferentes é um desafio. Por exemplo, um objeto Java (tipagem estática e forte) precisa ser "traduzido" para um conjunto de fatos em Prolog (tipagem dinâmica). Esse processo de serialização e desserialização de dados (muitas vezes via JSON ou formatos similares) cria uma "ponte" que precisa ser mantida. Uma alteração na classe Java não é automaticamente refletida no lado do Prolog, o que pode introduzir erros.

10. Paradigmas e manutenção de software. A evolução de requisitos ocorre quando o sistema precisa ser modificado para atender novas demandas de negócio, corrigir falhas, adaptar-se a novas tecnologias ou regulamentações. Explique como o paradigma orientado a objetos aborda a evolução de requisitos em sistemas grandes, destacando vantagens e limitações no contexto de facilidade de manutenção, impacto no código existente e esforço de adaptação.

O paradigma orientado a objetos (OO) aborda a evolução em sistemas de software de grande porte atuando como um eficaz amortecedor de impacto. Sua principal virtude não reside em prevenir a ocorrência de mudanças, mas sim em sua capacidade de canalizá-las para pontos específicos da arquitetura, minimizando o risco de um colapso sistêmico. O mecanismo central para gerenciar essa evolução é o isolamento, obtido pela imposição de fronteiras rígidas através do encapsulamento e pela criação de interfaces flexíveis e intercambiáveis via polimorfismo. Com isso, quando um novo requisito surge, a pergunta central deixa de ser "o que esta mudança pode quebrar?" e passa a ser "em qual compartimento encapsulado esta nova lógica se encaixa?".

Na prática, essa abordagem se traduz em vantagens significativas para o ciclo de vida do software. A manutenção é simplificada, pois o código tende a espelhar o domínio do negócio com entidades como *Cliente* ou *Pedido*; alterar uma regra de negócio se torna uma tarefa de localizar a classe correspondente, em vez de vasculhar um código espaguete desestruturado. Além disso, o baixo acoplamento, alcançado pelo uso de dependências em interfaces em vez de implementações concretas, contém o impacto das modificações. Uma alteração fica contida em seu módulo de origem, impedindo o temido efeito cascata. Por fim, o esforço de adaptação é reduzido, pois novas funcionalidades são frequentemente adicionadas pela criação de novas classes (seja por herança ou composição), em vez da alteração de código existente e já testado.

No entanto, a aplicação prática da orientação a objetos revela limitações críticas que exigem disciplina. A herança, por exemplo, pode se tornar uma forma de acoplamento patológico, criando um vínculo tão íntimo entre a classe-pai e suas subclasses que qualquer modificação na base pode quebrar implementações dependentes de forma imprevisível, travando a evolução que deveria facilitar. Ademais, a ilusão do encapsulamento perfeito é desfeita por requisitos transversais, como a necessidade de auditoria ou logging, que "furam" o modelo OO e exigem a modificação de múltiplas classes, violando o princípio do isolamento. Soma-se a isso a complexidade acidental, fruto da tentação de criar abstrações "futuristas" que, ao tentar antecipar todas as mudanças possíveis, geram um labirinto de *over-engineering* que dificulta a manutenção.

Essa visão pragmática é bem capturada por Steve McConnell em sua obra "Code Complete", onde ele expõe a mecânica do "encapsulamento da variação" — o ato consciente de identificar o que é volátil em um sistema e isolá-lo em suas próprias classes. Conclui-se, portanto, que a orientação a objetos é uma ferramenta poderosa para a evolução de software, mas seu sucesso depende de usar seu poder de contenção com discrição. A arquitetura destinada a facilitar a mudança não pode, por excesso de rigidez ou complexidade, se tornar o seu maior obstáculo.

11. Paradigmas e reutilização de código. Discuta como cada paradigma (imperativo, lógico, OO) promove ou dificulta o reuso de código. Apresente exemplos práticos,

como funções reutilizáveis no paradigma imperativo, regras reutilizáveis no paradigma lógico e classes ou bibliotecas no paradigma orientado a objetos.

A reutilização de código é um dos princípios centrais no desenvolvimento de software moderno, visto que permite reduzir redundâncias, aumentar a eficiência e facilitar a manutenção de sistemas. No entanto, a forma como o reuso é promovido ou dificultado varia de acordo com o paradigma de programação adotado. Os paradigmas imperativo, lógico e orientado a objetos apresentam estratégias próprias para estruturar e reaproveitar componentes, cada um com vantagens e limitações específicas.

No paradigma imperativo, a reutilização de código se manifesta principalmente por meio de funções e procedimentos. Esse paradigma, que se baseia em comandos sequenciais para alterar o estado do programa, possibilita encapsular trechos de lógica em funções que podem ser chamadas em diferentes pontos do sistema. Por exemplo, uma função para calcular a média de valores pode ser utilizada em múltiplos contextos, desde sistemas de gestão acadêmica até programas financeiros. Entretanto, o imperativo tende a gerar maior acoplamento entre funções, exigindo cuidados extras para que o reuso seja realmente eficiente.

Já no paradigma lógico, a reutilização ocorre de maneira distinta, pois o foco está na definição de fatos e regras que descrevem relações lógicas. Nesse modelo, em vez de detalhar como executar uma tarefa, o programador especifica o que deve ser alcançado, e o sistema de inferência se encarrega de encontrar a solução. Regras reutilizáveis, como as que definem vínculos hierárquicos ou relações de parentesco, podem ser aplicadas em diferentes contextos sem necessidade de reescrever código. Um exemplo clássico é a definição de uma regra de "ancestral" em Prolog, que pode ser usada em genealogia, em sistemas de organização empresarial ou mesmo em ontologias mais complexas. Todavia, a curva de aprendizado e a menor popularidade desse paradigma podem dificultar sua ampla adoção prática.

Por sua vez, o paradigma orientado a objetos (OO) eleva o reuso a um novo patamar, ao organizar o código em classes e objetos que representam entidades do mundo real. A herança, o polimorfismo e a composição são mecanismos que permitem estender e adaptar funcionalidades já existentes. Um exemplo recorrente é o uso de bibliotecas gráficas, nas quais classes podem ser herdadas e customizadas para criar interfaces específicas, sem que seja necessário reescrever componentes básicos como botões, menus ou janelas. Além disso, bibliotecas e frameworks amplamente compartilhados consolidaram a orientação a objetos como uma das abordagens mais eficazes no estímulo à reutilização.

Diante do exposto, percebe-se que cada paradigma oferece caminhos particulares para o reuso de código. O imperativo se apoia em funções reutilizáveis, o lógico em regras que podem ser aplicadas em diferentes contextos, e o orientado a objetos em estruturas robustas de classes e bibliotecas. Portanto, compreender as especificidades de cada modelo é essencial para selecionar a abordagem mais adequada ao problema a ser resolvido, garantindo soluções mais consistentes, escaláveis e sustentáveis no desenvolvimento de software.

12. Educação: por onde começar? Defenda, com argumentos, uma trilha pedagógica (imperativo → OO → lógico, ou outra). Considere curva de aprendizado, erros comuns e motivação.

A escolha de uma trilha pedagógica adequada para o ensino de programação é um desafio recorrente na educação em ciência da computação. A ordem em que os paradigmas são apresentados influencia diretamente a curva de aprendizado, a motivação dos estudantes e até mesmo os erros que estes tendem a cometer. Dessa forma, definir uma sequência lógica e coerente pode significar a diferença entre o engajamento e a frustração dos iniciantes.

Em primeiro lugar, iniciar pelo paradigma imperativo parece ser a opção mais natural. Nele, os estudantes aprendem a lógica básica da computação por meio de comandos sequenciais, variáveis e estruturas de controle, que são conceitos fundamentais para qualquer linguagem de programação. Além disso, esse paradigma fornece resultados visíveis rapidamente, como programas simples de cálculo ou manipulação de listas, o que contribui para a motivação dos alunos. O erro mais comum nesse estágio está no controle do fluxo e na manipulação incorreta de variáveis, mas tais obstáculos fazem parte do processo de internalização da lógica computacional.

Após esse primeiro contato, a progressão para o paradigma orientado a objetos (OO) é recomendada. Nessa etapa, os estudantes já têm familiaridade com os conceitos básicos e podem avançar para abstrações mais sofisticadas, como classes, herança e polimorfismo. A orientação a objetos, além de ser amplamente utilizada na indústria, favorece a criação de projetos mais próximos de situações reais, como sistemas de cadastro ou jogos simples, o que aumenta a motivação dos aprendizes. Erros comuns incluem dificuldades em diferenciar atributos de instâncias e de classes, além de confusões com a herança múltipla, quando aplicável. Ainda assim, esses problemas podem ser contornados com exemplos práticos e metodologias ativas de aprendizagem.

Por fim, o paradigma lógico deve ser introduzido em um momento posterior. Embora extremamente útil para áreas específicas, como inteligência artificial e sistemas especialistas, sua forma declarativa e baseada em inferências exige um nível maior de abstração e maturidade intelectual. Nesse ponto, os estudantes já terão desenvolvido a capacidade de raciocinar sobre diferentes modelos de programação e estarão mais preparados para compreender a lógica por trás da definição de regras e fatos. A principal dificuldade nesse estágio reside no distanciamento da prática cotidiana da programação, o que pode desmotivar alguns aprendizes caso não haja contextualização adequada.

Dessa maneira, a trilha pedagógica que segue a ordem imperativo → orientado a objetos → lógico mostra-se a mais equilibrada. Ela respeita a curva de aprendizado, minimiza frustrações com erros comuns e, sobretudo, mantém os estudantes motivados ao apresentar gradativamente desafios condizentes com seu nível de experiência. Portanto, estruturar o ensino de paradigmas dessa forma contribui não

apenas para a formação técnica, mas também para a consolidação de um pensamento computacional crítico e adaptável a diferentes contextos.

13. Síntese crítica. Depois de estudar, o paradigma OO é "universalmente melhor" ou "adequado a certos contextos"? Elabore uma posição crítica, citando evidências dos itens anteriores.

O paradigma orientado a objetos (OO) consolidou-se como um dos mais utilizados no desenvolvimento de software contemporâneo. Sua popularidade deriva da capacidade de organizar o código em estruturas próximas ao mundo real, promovendo reuso, modularidade e manutenção mais simples. No entanto, ao analisar criticamente sua aplicação, é possível afirmar que o paradigma OO não é "universalmente melhor", mas sim adequado a certos contextos, especialmente quando comparado a outros paradigmas.

Em primeiro lugar, é inegável que a OO favorece a reutilização de código por meio de herança, polimorfismo e composição. Como discutido anteriormente, bibliotecas gráficas ou frameworks web demonstram o potencial desse paradigma em evitar retrabalho e acelerar o desenvolvimento. Ademais, sua abordagem promove uma curva de aprendizado natural após o domínio do paradigma imperativo, o que contribui para a motivação dos estudantes e para o preparo profissional, considerando que grande parte do mercado adota linguagens como Java, C# e Python.

Por outro lado, há limitações que impedem a OO de ser considerada universalmente superior. O paradigma imperativo, por exemplo, ainda é mais acessível para iniciantes, oferecendo resultados imediatos e tangíveis. Já o paradigma lógico, apesar de menos popular, é indispensável em áreas específicas, como sistemas de inferência, inteligência artificial simbólica e resolução de problemas complexos baseados em regras. Isso evidencia que a OO não substitui totalmente outras formas de pensar a programação, mas complementa um ecossistema de paradigmas com objetivos distintos.

Além disso, a aplicação indiscriminada da orientação a objetos pode levar a projetos excessivamente complexos, com hierarquias profundas e sobrecarga conceitual. Esse risco, associado à tendência de forçar problemas em moldes "orientados a objetos" mesmo quando soluções funcionais ou lógicas seriam mais simples, demonstra que a eficácia da OO está diretamente relacionada ao contexto em que é empregada.

Portanto, pode-se concluir que a posição crítica mais equilibrada é reconhecer a orientação a objetos como um paradigma adequado a muitos contextos práticos, mas não uma solução universal. Ela se destaca pela flexibilidade e pelo amplo suporte no mercado, mas convive com outros paradigmas que, em situações específicas, podem oferecer maior clareza, eficiência ou simplicidade. A verdadeira competência em programação não está em adotar a OO como padrão absoluto, mas em compreender a natureza do problema e escolher o paradigma mais apropriado para solucioná-lo.

14. REFERÊNCIAS

1. ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR 6023: Informação e documentação – Referências – Elaboração**. Rio de Janeiro: ABNT, 2018.(UNIP.br)
2. **TANENBAUM, A. S.; VAN STEEN, M.** Sistemas Distribuídos: Princípios e Paradigmas. 3ª ed. São Paulo: Pearson Prentice Hall, 2017.
3. **SOMMERVILLE, I.** Engenharia de Software. 10ª ed. São Paulo: Pearson Education do Brasil, 2019.
4. **GAMMA, E. et al.** Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos. Bookman, 2000.
5. **SEBESTA, R. W.** Conceitos de Linguagens de Programação. 11ª ed. Bookman, 2018.
6. **DONGARRA, J.; HEROUX, M. A.; LUSZCZEK, P.** A Report on the Evolution of High-Performance Computing. SIAM Review, v. 63, n. 3, p. 439-465, 2021.
7. **RAYMOND, Eric S.** The Art of Unix Programming. Addison-Wesley Professional, 2003.