**Lista 1 - MPOO**

### ****1) Explique:****

#### ****1.1) Os porquês de o paradigma da orientação a objetos (POO) ser uma alternativa para o antigo paradigma clássico (imperativo ou estruturado):****

O paradigma de orientação a objetos (POO) surgiu para resolver algumas limitações do paradigma estruturado, como a dificuldade de organizar e reutilizar o código em projetos grandes. No paradigma estruturado, o foco está em funções e fluxos de controle, o que pode levar a códigos extensos, com baixa modularidade e difícil manutenção.  
Já no paradigma POO, o foco está nos objetos, que encapsulam dados e comportamentos, promovendo:

* **Reutilização de código:** Classes e objetos podem ser usados em diferentes partes do sistema.
* **Modularidade:** A divisão do código em classes melhora a organização e a compreensão do sistema.
* **Facilidade de manutenção:** Alterações em uma classe impactam apenas os objetos relacionados, minimizando efeitos colaterais.

#### ****1.2) Como deu o surgimento do POO?****

O paradigma POO começou a ser desenvolvido na década de 1960 com a linguagem Simula, criada por Ole-Johan Dahl e Kristen Nygaard. Eles desenvolveram a ideia de classes e objetos como forma de modelar sistemas complexos, principalmente para simulações. Posteriormente, a linguagem Smalltalk refinou o conceito e introduziu características como herança e polimorfismo, consolidando a base da POO moderna.

**REFERÊNCIA:** Elaboração própria.

#### ****1.3) Quais as principais diferenças entre o paradigma clássico do POO?****

* **Paradigma Clássico (Imperativo):**
  + Foco no fluxo de controle (sequência de comandos).
  + Usa funções e procedimentos para manipular dados.
  + Baixa modularidade e reutilização.
* **POO:**
  + Foco em objetos que encapsulam dados e comportamentos.
  + Usa conceitos como herança, polimorfismo e encapsulamento.
  + Alta modularidade e reutilização de código.

### ****2) Analise a frase:****

"Quando um produto grande é formado por um único bloco monolítico de código, sua manutenção se torna um pesadelo. Mesmo para o autor de tal monstruosidade, tentar depurar o código é extremamente difícil; para outro programador, compreendê-lo é praticamente impossível” (Schach, 2010 p. 1771).

#### ****Como o paradigma OO dirime essa problemática?****

O paradigma de orientação a objetos (POO) resolve essa problemática ao dividir o sistema em componentes menores e mais manejáveis, que são as **classes**. Cada classe encapsula dados e comportamentos relacionados, permitindo que o código seja modular, compreensível e reutilizável.

* **Encapsulamento:** Esconde detalhes de implementação, expondo apenas o que é necessário, o que facilita a compreensão e uso da classe por outros programadores.
* **Modularidade:** Dividir o código em classes promove uma organização que torna mais simples localizar, corrigir ou atualizar partes específicas do sistema.
* **Reutilização de código:** Ao definir classes reutilizáveis, a duplicação de código é reduzida, melhorando a eficiência do desenvolvimento.
* **Polimorfismo:** Permite que diferentes partes do sistema interajam de forma flexível, reduzindo a necessidade de alterar várias partes do código ao adicionar novas funcionalidades.

No geral, a POO transforma o código monolítico em uma estrutura organizada, facilitando a manutenção e o trabalho colaborativo.

### ****3) No intuito de relacionarmos o paradigma clássico à disciplina MPOO, responda como fazemos na linguagem de programação C para criar uma solução de um problema de forma que possamos reutilizar o código em outras situações.****

Na linguagem de programação C, a reutilização de código no paradigma clássico é alcançada principalmente através das **funções** e do uso de **bibliotecas**.

#### ****Passos para Reutilização em C:****

1. Funções:
2. Encapsulam um bloco de código que pode ser chamado sempre que necessário.
3. São reutilizáveis, pois podem ser chamadas com diferentes parâmetros.

Exemplo:

int soma(int a, int b) {

return a + b;

}

A função soma pode ser usada em diferentes contextos para realizar a mesma operação.

**Structs:**

* 1. Permitem agrupar diferentes tipos de dados em uma única estrutura.
  2. Apesar de não possuir métodos, o uso de structs organiza os dados, possibilitando sua manipulação através de funções externas.

Exemplo:

typedef struct {

char nome[50];

int idade;

} Pessoa;

**Modularização:**

1. Criando **arquivos header (.h)** e **arquivos fonte (.c)** para dividir o código em módulos reutilizáveis.
2. Isso permite organizar funções e structs em bibliotecas para uso em diferentes programas.
3. Exemplo:
   * 1. Arquivo matematica.h:

int soma(int a, int b);

* + 1. Arquivo matematica.c:

#include "matematica.h"

int soma(int a, int b) {

return a + b;

}

Essas práticas promovem a reutilização de código em C, aproximando-se do conceito de **modularidade**, que é amplamente explorado no paradigma da orientação a objetos.

### ****4) Qual a relação entre**** struct ****da linguagem de programação C e o conceito de classe?****

#### ****Semelhanças:****

**Agrupamento de Dados:**  
Tanto a struct em C quanto a classe em linguagens orientadas a objetos agrupam diferentes tipos de dados em uma única entidade. Por exemplo:

* 1. Uma struct pode armazenar atributos como nome e idade.
  2. Uma classe também armazena esses atributos como propriedades.

**Exemplo de struct em C:**

typedef struct {

char nome[50];

int idade;

} Pessoa;

**Exemplo de classe em Java:**

public class Pessoa {

String nome;

int idade;

}

**Reutilização:**  
Ambas podem ser usadas para organizar dados e facilitar a reutilização do código.

#### ****Diferenças:****

**Comportamento (Métodos):**

* 1. Na POO, uma classe encapsula tanto **dados** quanto **comportamentos** (métodos).
  2. Em C, uma struct só pode armazenar **dados**; comportamentos devem ser implementados separadamente por funções externas.

**Encapsulamento:**

* 1. As classes possuem controle de acesso através de modificadores (public, private, protected), que limitam a visibilidade de atributos e métodos.
  2. As structs não possuem encapsulamento nativo; todos os membros são acessíveis diretamente.

**Herança e Polimorfismo:**

* 1. Classes suportam herança e polimorfismo, permitindo a criação de hierarquias de classes e maior reutilização.
  2. Structs não possuem suporte para herança ou polimorfismo.

#### ****Conclusão:****

A struct em C pode ser vista como uma versão simplificada de uma classe, pois organiza dados, mas não implementa os conceitos mais avançados de orientação a objetos, como encapsulamento, herança e métodos associados.

### ****5) Em uma demanda de desenvolvimento de software, uma das principais dúvidas surge na modelagem para diferenciar o que será código ou que apenas trate de uma interação do usuário com o sistema. Sendo assim:****

#### ****5.1) Apresente uma estratégia para diferenciar o que será código e o que não será código, ou seja, apenas uma interação com o sistema.****

Uma estratégia eficiente é utilizar **diagramas de caso de uso (use case)** e **documentos de requisitos** para distinguir:

* **Interação com o sistema:** Representa ações do usuário (ex.: clicar em botões, inserir dados). Essas ações geralmente não são diretamente codificadas, mas servem para definir como o sistema será utilizado.
* **Código:** Representa funcionalidades implementadas no sistema, como cálculos, validações e manipulação de dados. São codificadas a partir de requisitos funcionais e diagramas de classes.

Exemplo:

* **Interação:** "Usuário clica no botão para gerar um relatório."
* **Código:** Lógica para coletar os dados, processá-los e formatar o relatório.

#### ****5.2) Como uma demanda no desenvolvimento de software que será transformada em código pode ser apresentada para uma equipe de desenvolvimento de software?****

Uma demanda pode ser apresentada por meio de:

**Documentos de Requisitos:**

* + Descrevem o que o sistema deve fazer de forma detalhada.
  + Exemplo: "O sistema deve calcular o imposto sobre o valor da compra baseado na alíquota definida."

**Diagramas de Classes:**

* + Representam as entidades do sistema, seus atributos e métodos.

**Histórias de Usuário (User Stories):**

* + Uma abordagem ágil que descreve as funcionalidades do ponto de vista do usuário.
  + Exemplo: "Como cliente, quero visualizar o resumo do pedido para confirmar os itens antes de finalizar a compra."

#### ****5.3) Apresente uma situação em que há pelo menos duas demandas: uma relacionada à interação com o sistema e outra que se trata de uma funcionalidade codificada. Utilize trechos de um documento de requisitos, diagrama de classes e use case.****

##### ****Situação: Sistema de Devolução de Produtos****

**Demanda de interação:**

* + **Requisito:** O cliente deve clicar em "Solicitar Devolução" e fornecer o número do pedido.
  + **Descrição em Use Case:**
    - Ator: Cliente.
    - Ação: Selecionar produto para devolução.

**Demanda de funcionalidade:**

* + **Requisito:** O sistema deve verificar se o pedido existe e se o produto está elegível para devolução.
  + **Descrição em Diagrama de Classes:**  
    Classe: Pedido
    - Atributos: id, dataCompra, status.
    - Métodos: verificarElegibilidade().

**Use Case Simplificado:**

* **Título:** Devolver Produto.
* **Ator Principal:** Cliente.
* **Fluxo Principal:**
  1. O cliente solicita a devolução de um produto.
  2. O sistema verifica se o produto é elegível.

**Diagrama de Classes: (fazer)**

+------------------+

| Pedido |

+------------------+

| - id: int |

| - dataCompra: Date|

| - status: String |

+------------------+

| + verificarElegibilidade(): boolean |

+------------------+

### ****6) A partir da descrição do problema abaixo, apresente o devido diagrama de use case:****

#### ****[RF01] - Devolver Produto****

* **Ator Principal:** Cliente
* **Atores Secundários:** Funcionário, Caixa
* **Resumo:** Este caso de uso descreve as etapas necessárias para que um cliente devolva um produto comprado.
* **Pré-condições:** É necessário existir um produto.
* **Pós-condições:** Escolher a opção de devolução (troca ou ressarcimento).

**Diagrama de Use Case:**  
Vou descrever o diagrama e, se precisar, posso gerar uma representação visual.

**Casos de Uso:**

* + **Devolver Produto:**
    - Fluxo principal:
      1. Solicitar devolução apresentando o produto.
      2. O sistema registra a devolução e atualiza o estoque.
      3. O sistema permite escolher entre troca ou ressarcimento.
  + **Atualizar Estoque:**
    - Fluxo principal:
      1. Atualizar a base de dados com o produto devolvido.
  + **Efetuar Troca:**
    - Fluxo principal:
      1. Trocar o produto por outro.
  + **Ressarcir Cliente:**
    - Fluxo principal:
      1. Ressarcir o cliente pelo valor pago.

**Atores:**

* + Cliente (ator principal).
  + Funcionário (para efetuar troca).
  + Caixa (para realizar ressarcimentos).

### ****7) A partir da codificação abaixo:****

// arquivo Classe.java

public class Classe {

int atributo;

}

// arquivo App.java

public class App {

public static void main(String[] args) {

int variavel;

Classe objeto = new Classe();

}

}

#### ****7.1) Podemos dizer que o tamanho da área de memória de**** variavel ****é o mesmo de**** objeto****, uma vez que**** objeto ****possui apenas um atributo do tipo inteiro?****

Não, o tamanho da área de memória de variavel e objeto não é o mesmo.

variavel**:**  
Como é uma variável de tipo primitivo (no caso, um tipo int), ela ocupa 4 bytes (em sistemas de 32 bits ou 64 bits, dependendo da arquitetura da máquina).

objeto**:**  
Quando criamos um objeto, a área de memória necessária é maior, porque o objeto contém tanto a **referência** (que ocupa 4 ou 8 bytes, dependendo da arquitetura) quanto os atributos definidos na classe.  
No caso de Classe, o atributo int atributo ocupa 4 bytes, mas o objeto Classe em si ocupa mais memória para armazenar a referência e a sobrecarga da instância do objeto.

Portanto, a memória alocada para o **objeto** é maior que a memória alocada para a **variável**.

#### ****7.2) Explane sobre como é realizada a alocação de memória em JAVA para uma variável do tipo primitiva e para uma instância (objeto).****

Em **Java**, a alocação de memória para variáveis primitivistas e objetos ocorre de formas diferentes:

**Para variáveis primitivas:**

* + As variáveis primitivas (como int, char, boolean) são armazenadas **na pilha (stack)**. A memória para uma variável primitiva é alocada de forma direta e é **liberada automaticamente** quando a variável sai do escopo.

Exemplo:

int numero = 10; // A variável numero é armazenada na pilha.

* + A memória para essa variável é liberada assim que o método ou bloco de código termina.

**Para objetos (instâncias de classes):**

* + Objetos são armazenados **na heap** (memória dinâmica). A memória para o objeto é alocada quando o new é chamado e é liberada automaticamente pela **garbage collection** (coletor de lixo) quando o objeto não é mais referenciado.

Exemplo:

Classe objeto = new Classe(); // O objeto é armazenado na heap.

* + A memória para o objeto permanece até que não haja mais referências a ele.