Vitor de Melo Soares - Ulbra Torres
Texto acadêmico referente à AP2 da cadeira Laboratório de Programação.

Introdução

A programação orientada a objetos (POO) é um paradigma fundamental no desenvolvimento de software, reconhecida por suas contribuições significativas à modularidade, reutilização de código, manutenção e abstração. Ao encapsular dados e comportamentos em objetos, a POO permite que sistemas complexos sejam divididos em partes menores e mais gerenciáveis, facilitando a compreensão e a colaboração em equipes. Um dos principais benefícios desse paradigma é a reutilização de código por meio da herança, que possibilita a criação de classes base que podem ser estendidas, reduzindo a duplicação e acelerando o desenvolvimento. Além disso, a POO simplifica a manutenção de software, pois o encapsulamento protege o estado interno dos objetos, minimizando o impacto de alterações. A estrutura hierárquica das classes permite que mudanças em uma classe base sejam automaticamente refletidas nas classes derivadas. A abstração, um princípio essencial da POO, permite que desenvolvedores se concentrem em aspectos relevantes de um problema, melhorando a legibilidade do código e facilitando a comunicação. Assim, a POO se estabelece como uma abordagem poderosa para lidar com a complexidade dos sistemas modernos, resultando em software mais robusto e flexível.

Encapsulamento

O encapsulamento refere-se à prática de restringir o acesso direto aos dados internos de um objeto, permitindo que apenas métodos específicos manipulem esses dados. Essa abordagem promove a segurança e a integridade das informações, essencial em sistemas complexos.

Ao encapsular os dados, os desenvolvedores definem quais atributos e métodos de um objeto são acessíveis externamente e quais permanecem ocultos. Normalmente, os atributos são declarados como privados, enquanto os métodos que permitem a interação com esses atributos são públicos. Essa separação cria uma interface clara e controlada, onde as operações no objeto são realizadas apenas através de

métodos definidos, conhecidos como "getters" e "setters". Os getters permitem a leitura dos valores dos atributos, enquanto os setters possibilitam a modificação desses valores, frequentemente incluindo validações para garantir que os dados permaneçam em um estado consistente.

Uma das principais vantagens do encapsulamento é a proteção dos dados. Ao limitar o acesso direto aos atributos, os desenvolvedores podem prevenir alterações indevidas que poderiam levar a erros ou comportamentos inesperados. Por exemplo, em um sistema bancário, um atributo que representa o saldo de uma conta não deve ser modificado diretamente, mas apenas através de métodos que implementem regras de negócio, como depósitos e retiradas, garantindo que o saldo nunca se torne negativo. Exemplo:

```
public class Agencia

{

    private List<Destino> Destinos { get; set; }

    private List<Cliente> Clientes { get; set; }

    private List<PacoteTuristico> Pacotes { get; set; }

    private List<Reserva> Reservas { get; set; }



       public Agencia(List<Destino> destinos, List<Cliente> clientes,
List<PacoteTuristico> pacotes, List<Reserva> reservas)

    {

        Destinos = destinos;

        Clientes = clientes;

        Pacotes = pacotes;

        Reservas = reservas;

      }
```

Abstração

A abstração permite que os desenvolvedores simplifiquem a complexidade dos sistemas ao focar nos aspectos mais relevantes e ocultar detalhes desnecessários. Essa prática é fundamental para modelar conceitos do mundo real de forma que sejam compreensíveis e gerenciáveis no contexto de software.

O funcionamento da abstração se dá principalmente através da definição de classes e interfaces. Uma classe é uma estrutura que encapsula dados e comportamentos relacionados, enquanto uma interface define um contrato que as classes devem seguir, especificando quais métodos devem ser implementados. Ao criar uma classe, os desenvolvedores podem identificar e isolar os atributos e métodos que são essenciais para representar uma entidade específica, ignorando detalhes que não são relevantes para a sua utilização. Exemplo:

```
public interface IPesquisavel

{

    public void PesquisarReserva(int codigo);
```

Método utilizado na classe Agencia:

```
public void PesquisarReserva(int codigo)

    {

        string? status = null;

        foreach(var item in Reservas)

        {

            var cod = item.CodigoReserva;

            if(cod == codigo)

            {

                    Console.WriteLine($"\nReserva encontrada\nDestino:
{item.Destino}\nData: {item.Datas}\nDescrição: {item.Descricao}");
```

```
                status = "Concluido";
                break;
            }else{
                continue;
            }
        }
        switch(status)
        {
            case null:
            Console.WriteLine("Reserva não encontada.");
            break;

            case "Concluido":
            break;
        }
```

Herança

A herança permite que uma classe herde propriedades e comportamentos de outra classe. Esse mecanismo é essencial para a reutilização de código e a criação de hierarquias de classes, proporcionando uma forma de organizar e estruturar o software de maneira eficiente.

No contexto da POO, uma classe que herda de outra é chamada de classe derivada ou subclasse, enquanto a classe da qual ela herda é chamada de classe base ou superclasse. A subclasse não apenas possui todos os atributos e métodos da superclasse, mas também pode adicionar novos atributos e métodos ou até mesmo sobrescrever métodos existentes. Isso promove a reutilização, pois a subclasse

pode aproveitar a implementação já existente na superclasse, evitando a duplicação de código. Exemplo:

```
public class Reserva : PacoteTuristico

{

    private string NumeroIdentificacaoCliente { get; set; }

    public int CodigoReserva { get; set; }



     public Reserva(string numeroIdentificacaoCliente, int codReserva,
string destino, DateOnly datas, decimal preco, int vagasDisponiveis,
string codigo, string descricao):base(destino, datas, preco,
vagasDisponiveis, codigo, descricao)

    {

        CodigoReserva = codReserva;

        NumeroIdentificacaoCliente = numeroIdentificacaoCliente;

    }

}
```

Polimorfismo

O polimorfismo refere-se à capacidade de um objeto de assumir muitas formas. Essa característica permite que uma única interface seja utilizada para interagir com diferentes tipos de objetos, facilitando a flexibilidade e a extensibilidade do código. O polimorfismo se manifesta principalmente em duas formas: o polimorfismo de tempo de compilação (ou sobrecarga) e o polimorfismo de tempo de execução (ou sobrescrita).

No polimorfismo de tempo de compilação, várias versões de um método ou função podem coexistir, diferenciadas pelo número ou tipo de parâmetros. Por exemplo, se temos um método chamado *somar*, podemos ter diferentes implementações que aceitam diferentes tipos de argumentos, como dois números inteiros, dois números

de ponto flutuante ou até mesmo uma lista de números. Esse tipo de polimorfismo permite que o compilador decida qual método invocar com base na assinatura do método chamado, facilitando a reutilização de código e a clareza. Exemplo:

```
public abstract class ServicoViagem

{

    public string Codigo { get; set; }

    public string Descricao { get; set; }



    public ServicoViagem(string codigo, string descricao)

    {

        Codigo = codigo;

        Descricao = descricao;

    }



    public abstract void Reservar();

    public abstract void Cancelar();

}
```

```
public class PacoteTuristico : ServicoViagem, IResevavel, IPesquisavel

{

    public string Destino { get; set; }

    public DateOnly Datas { get; set; }

    public decimal Preco { get; set; }

    public int VagasDisponiveis { get; set; }
```

```
      public PacoteTuristico(string destino, DateOnly datas, decimal
preco,     int     vagasDisponiveis,     string     codigo,     string
descricao):base(codigo,descricao)

    {

        Destino = destino;

        Datas = datas;

        Preco = preco;

        VagasDisponiveis = vagasDisponiveis;

    }



    public override void Reservar()

    {

        if(VagasDisponiveis > 0)

        {

            VagasDisponiveis =- 1;

        }else

        {

        }

    }




    public override void Cancelar()

    {

        VagasDisponiveis =+1;

    }

}
```

Conclusão do aprendizado obtido com o desenvolvimento do projeto

Durante uma semana dedicada ao desenvolvimento de um projeto pessoal com Programação Orientada a Objetos (POO), aprendi lições valiosas. No início, a estruturação das classes e objetos foi um desafio, mas entender os princípios de encapsulamento, herança e polimorfismo me ajudou a criar um código mais modular e reutilizável.

Dediquei tempo ao planejamento, mapeando requisitos e desenhando diagramas de classes. Essa etapa foi crucial para evitar retrabalhos e facilitar a execução do projeto. Ao longo do desenvolvimento, enfrentei vários problemas, desde bugs até dificuldades na integração de componentes, o que exigiu raciocínio lógico e persistência.

Embora tenha trabalhado principalmente sozinho, compartilhar meu progresso com colegas trouxe novas perspectivas e melhorou a qualidade do trabalho. Essa experiência me mostrou que, além das técnicas de POO, ter uma mentalidade orientada a problemas e colaborar com os outros é fundamental. Estou animado para aplicar essas lições em futuros projetos!