Aluno: Rodrigo Secundo Araújo

Explicação sobre a minha aplicação dos princípios SOLID

Estrutura do repositório:

\Aplicação SOLID\código_original\locadora.py - código sem alterações feitas;
\Aplicação SOLID\código_refatorado\locadora.py - código refatorado;
\Aplicação SOLID\código_refatorado\veiculo.py - classes e interfaces para diferentes tipos de veículos;
\Aplicação SOLID\código_refatorado\main.py - exemplo de uso de um sistema refatorado.

Link para os códigos no github:

-

Identificação dos problemas e das alterações feitas no código original:

1. Single Responsibility Principle (SRP):

Problema:
No código original, a classe Locadora é responsável por múltiplas funções: gerenciamento de inventário de veículos, cálculo de preços e geração de relatórios. Isso vai contra o SRP, que recomenda que cada classe ou módulo deve ter apenas uma razão para mudar, ou seja, uma única responsabilidade.

Refatoração:

Para atender ao SRP, foram feitas as seguintes alterações:

- A lógica de cálculo de preços foi movida para uma nova classe chamada CalculoPrecoAluguel, que é responsável exclusivamente por calcular os preços de aluguel dos veículos.
- A classe Locadora foi simplificada para focar apenas no gerenciamento do inventário de veículos e na geração de relatórios.

Justificativa:
Ao separar as responsabilidades, agora se tem classes menores e mais focadas, cada uma com uma única função clara. Isso torna o código mais fácil de ler e entender, além de facilitar a manutenção, pois qualquer modificação no cálculo de preços pode ser feita na classe CalculoPrecoAluguel sem impactar o gerenciamento de veículos na classe Locadora.

2. Open/Closed Principle (OCP):

Problema:
No código original, o método calcular_preco dentro da classe Locadora utiliza uma estrutura condicional (if/elif) para determinar o preço com base no tipo de veículo. Para adicionar um novo tipo de veículo, seria necessário modificar esse método, o que viola o OCP, que recomenda que as classes devem estar abertas para extensão, mas fechadas para modificação.

Refatoração:
Foi criada uma estrutura de herança com uma classe base abstrata Veiculo e subclasses específicas para cada tipo de veículo: Carro, Moto, e Caminhao. Cada uma dessas subclasses implementa seu próprio método calcular_preco, que permite adicionar novos tipos de veículos estendendo a classe Veiculo sem modificar a lógica na Locadora.

Justificativa:
Com essa refatoração, o sistema pode ser facilmente estendido para novos tipos de veículos, como bicicletas ou vans, sem alterar o código existente na Locadora. Isso torna o código mais robusto e adaptável a mudanças futuras, melhorando a modularidade e o encapsulamento das regras de negócios.

3. Liskov Substitution Principle (LSP):

Problema:
No código original, a lógica de veículos está embutida dentro da classe Locadora, o que impede o uso de tipos de veículos de forma intercambiável. Isso viola o LSP, que afirma que uma classe derivada deve poder substituir sua classe base sem alterar o comportamento esperado do programa.

Refatoração:
Foi criada a classe abstrata Veiculo com métodos abstratos que devem ser implementados por todas as subclasses (Carro, Moto, e Caminhao). Agora, cada veículo segue a mesma interface e pode ser tratado de forma intercambiável pela Locadora e pela classe CalculoPrecoAluguel.

Justificativa:
Através dessa mudança, agora é possível tratar qualquer instância de veículo como um Veiculo genérico. A Locadora e o CalculoPrecoAluguel podem operar com qualquer tipo de veículo sem precisar saber detalhes sobre as subclasses específicas. Isso permite que novas classes de veículos sejam criadas e integradas de forma mais flexível e mantém o comportamento esperado para o sistema.

4. Interface Segregation Principle (ISP):

Problema:
No código original, a Locadora centraliza múltiplas responsabilidades, como o cálculo de preços e a geração de relatórios, forçando todas as operações a

dependerem de uma única interface. Isso viola o ISP, que recomenda que uma classe ou cliente não deve ser forçado a depender de interfaces que ele não utiliza.

Refatoração:
Para atender ao ISP, foi definida uma interface específica Veiculo com métodos necessários (calcular_preco, tipo e marca). A classe Locadora agora lida apenas com o gerenciamento de inventário, enquanto a lógica de preços é isolada na CalculoPrecoAluguel.

Justificativa:
Agora, as classes são mais focadas em uma única responsabilidade, e cada classe ou módulo utiliza apenas as interfaces necessárias para sua função. Isso torna o sistema mais modular e minimiza o acoplamento, o que facilita a expansão e manutenção do código.

5. Dependency Inversion Principle (DIP):

Problema:
No código original, Locadora depende diretamente da implementação concreta da lógica de cálculo de preços, acoplando o gerenciamento de veículos à lógica de negócio dos preços. Isso viola o DIP, que sugere que classes de alto nível não devem depender de classes de baixo nível, mas de abstrações.

Refatoração:
Para aplicar o DIP, Locadora e CalculoPrecoAluguel foram modificadas para interagir com a abstração Veiculo em vez de lidar com detalhes de classes concretas. Agora, Locadora delega o cálculo de preço para uma instância de CalculoPrecoAluguel, que recebe um Veiculo como parâmetro.

Justificativa:
Com essa estrutura, Locadora não está mais acoplada a classes concretas de veículos ou à lógica específica de cálculo de preços. Isso facilita a substituição de implementações (por exemplo, podemos adicionar novos tipos de veículos ou alterar a forma de cálculo de preços) sem precisar modificar a Locadora. Dessa forma, o sistema se torna mais flexível e resiliente a mudanças.