S-SOLID:

A refatoração realizada na classe **ProcessadorEncomendas** foi a seguinte:

1. Separei em classes distintas cada método que desempenhava uma função específica dentro da classe **ProcessadorEncomendas** (**ProcessadorEncomendas** e **SaveArquivo**).
2. Criei uma classe **Encomenda** para que seja possível instanciá-la e passá-la como parâmetro para o método `processar()`, eliminando assim a necessidade de capturar o ID e o peso via scanner.
3. Na classe `main`, instanciei as classes **ProcessadorEncomendas** e **Encomenda** para que fosse possível utilizar o método `processadorEncomendas.processar(encomenda)`.

Essa refatoração foi realizada para seguir o **Single Responsibility Principle** do **SOLID** .

O-SOLID:

A refatoração realizada na classe **SistemaPagamento** foi a seguinte:

1. Separei em classes distintas cada `if` e `else` contido no método **realizarPagamento** (**PagamentoBoleto**, **PagamentoCartao**, **PagamentoPix**).
2. Criei uma interface para implementar a regra de negócio em cada uma das classes mencionadas. Modifiquei as classes necessárias para adequá-las ao modelo de negócio definido pela interface.
3. Alterei a classe **SistemaPagamento** e o método `realizarPagamento()` para o formato `double pagar()`, que passa os seguintes parâmetros: `(double valor, IConcluirPagamento concluirPagamento)`. Assim, para executar o método, é necessário fornecer essas informações.
4. Na criação da `Main`, instanciei a classe **SistemaPagamento** para que fosse possível acessar o método `pagar()` e forneci as informações necessárias: o valor e a classe que implementa a interface referente à forma de pagamento.

Essa refatoração foi realizada para aderir ao **Open Closed Principle** do **SOLID**.

L-SOLID:

A refatoração realizada nas classes `ContaBancaria` e `ContaPoupanca` foi a seguinte:

1. Criei uma interface `IConta` com os métodos `depositar(double valor)` e `getSaldo()`, para aderir às contas (ContaCorrente e ContaPoupanca);

2. Criei uma interface separada `IContaSaque`, que adiciona o método `sacar(double valor)`, assim só a conta que pode efetuar saques realiza esse método;

3. Criei a classe `ContaCorrente` para implementar `IContaSaque`, pois ela pode realizar a funcionalidade de saque;

4. Ajustei a classe `ContaPoupanca` para implementar apenas `IConta`, pois as contas poupança não podem realizar saques direto. Assim a classe deixou de sobrescrever o método `sacar()` com uma exceção, assim deixou de descumprir o Princípio de Substituição de Liskov;

5. Essa separação garante que qualquer código que utilize objetos do tipo `IConta` ou `IContaSaque` possa tratá-los corretamente sem causar erros inesperados, como exceções de operação não suportada;

Essa refatoração foi realizada para aderir ao **Liskov Substitution Principle** do **SOLID**.

I-SOLID:

A refatoração realizada na interface **Veiculo** foi a seguinte:

1. Criei duas novas interfaces: “VeiculoNavegador” e “VeiculoVoador”.
2. Deletei o método “ voar();” de “**Veiculo**” e coloquei na interface “**VeiculoVoador**”; e deletei o método “navegar();” e coloquei na interface “**VeiculoNavegador**”.
3. Dentro da interface “**Veiculo**” ficou somente o método “dirigir();”.

A refatoração realizada na classe **Carro** foi a seguinte:

1. Como a interface “**Veiculo**” só possui o método “dirigir()” e “**Carro**” implementa da interface “Veiculo”, a classe “Carro” possui somente o método “dirigir()”. Com isso, os outros métodos, “voar()” e “navegar()”, foram deletados da classe “**Carro**”, pois, obviamente, um carro não voa nem navega.
2. Foi feita a criação das classes: “**Avião**”, “**Caravela**” e “**CarroAnfíbio**”, como exemplo para que seja possível a implementação das outras duas interfaces e para que haja uma classe que implementa duas interfaces.
3. A classe “**Avião**” implementa “**VeiculoVoador**”, fazendo com que a classe “**Avião**” possua o método “voar()”; pois um avião, obviamente, voa.
4. A classe “**Caravela**” implementa “**VeiculoNavegador**”, fazendo com que a classe “**Caravela**” possua o método “navegar()”; pois uma caravela, obviamente, navega sobre o mar.
5. A classe “**CarroAnfíbio**” implementa tanto “**Veiculo**” quanto “**VeiculoNavegador**”, tendo os métodos “dirigir()” e “navegar()”.

(Bônus) D-SOLID - Exemplo 1:

A refatoração realizada na classe **MySQLConnection**:

1. Foi feita a criação da interface “**SQLConnection**”, o qual possui o método “connect()”. Pois, dentro da classe **UsuarioDAO**, há um método que faz uma conexão em um banco de dados. Nesse caso, foi o SQL, porém, vamos supor que o administrador do projeto queira utilizar outro banco de dados. Seria necessário a mudança da classe “**UsuarioDAO**” e da criação de outra classe.
2. Agora, a classe “**MySQLConnection**” implementa a interface “**SQLConnection**” e faz o “@Override” da classe “connect()”.
3. Foi feita a criação da classe “**PostgreSQLConnection**” que também implementa a interface “SQLConnection” e o “@Override” da classe “connect()”.

A refatoração realizada na classe **UsuarioDAO**:

4. Primeiramente, houve a mudança do atributo da classe que ficou “private SQLConnection sqlConnection” ao invés de “private MySQLConnection mySQLConnection”.
5. Houve a mudança do método construtor “**UsuarioDAO"** o qual foi adicionado um parâmetro, “**SQLConnection sqlConnection**” no método e, já que houve a mudança de atributo dentro da classe, mudou dentro do método também. É necessário o parâmetro, pois permite que o administrador do projeto consiga escolher o banco de dados desejado de forma flexível. Além disso, houve a mudança no método “salvarUsuario(String nome)” que, em vez de ser “mySQLConnection.connect()”, é “sqlConnection.connect()”.