1) Explique o que é o Princípio da Responsabilidade Única e como ele ajuda a manter o código mais organizado e fácil de dar manutenção. Cite um exemplo prático de uma classe que poderia violar este princípio e como você corrigiria a violação.

R= Uma classe deve ter apenas um motivo para mudar, significando que uma classe deve ter apenas uma tarefa ou responsabilidade. Deixando assim mais fácil o reuso.

Errado:

```
class ClassCliente {
  void calculaPagamento() {
    print("Pago");
  }

  void salvarDados() {
    print("Salvo");
  }

  void controleDePonto() {
    print("Ponto batido");
  }
}
```

Certo:

```
class CalculaPagamento {
  void calcular() {}
}

class SalvarDados {
  void salvar() {}
}

class ControleDePonto {
  void reportarHoras() {}
}
```

2) Descreva o Princípio Aberto/Fechado. Como ele pode ser aplicado para que um sistema seja fácil de expandir, sem a necessidade de modificar código existente? Ilustre sua resposta com um exemplo teórico.

R= Isso significa que o comportamento de uma classe ou módulo deve poder ser estendido sem alterar seu código-fonte original. O objetivo é evitar a necessidade de modificar código já existente, pois isso pode introduzir bugs ou problemas de manutenção. Em vez disso, deve-se focar em adicionar novas funcionalidades através de extensões, como herança ou composição, mantendo o código original intacto.

Errado:

```
class CalculadoraImpostos {
  double calcularImposto(String tipoProduto, double valor) {
    if (tipoProduto == "eletronico") {
      return valor * 0.2;
    } else if (tipoProduto == "alimento") {
      return valor * 0.1;
    } else {
      return valor * 0.15;
    }
  }
}
```

certo:

```
abstract class Imposto {
  double calcular(double valor);
}

class ImpostoEletronico implements Imposto {
  @override
  double calcular(double valor) {
    return valor * 0.2;
  }
}

class ImpostoAlimento implements Imposto {
  @override
  double calcular(double valor) {
    return valor * 0.1;
  }
}

class CalculadoraImpostos {
  double calcularImposto(Imposto imposto, double valor) {
    return imposto.calcular(valor);
  }
}
```

3) Explique o Princípio da Substituição de Liskov e por que é importante seguir este princípio ao trabalhar com herança em linguagens orientadas a objetos, como o Dart. Dê um exemplo de uma situação em que a violação desse princípio poderia causar problemas no código.

R= Formulado por Barbara Liskov, o princípio declara que os objetos de uma superclasse devem ser substituíveis por objetos de suas subclasses sem afetar a corretude do programa. O LSP é crucial para alcançar reusabilidade e modularidade no design de software. Ele garante que uma classe derivada possa estender uma classe base sem alterar seu comportamento esperado.

Errado:

```
abstract class Ave {
  void comer();
}

abstract class AveQueVoa extends Ave {
  void voar();
}

class Pardal implements AveQueVoa {
  @override
  void comer() {
    print("O Pardal está comendo.");
  }

  @override
  void voar() {
    print("O Pardal voa.");
  }
}

class Avestruz implements Ave {
  @override
  void comer() {
    print("Avestruz está comendo.");
  }
}

void main() {
  AveQueVoa pardal = Pardal();
  pardal.voar(); // Funciona como esperado
}
```

Certo:

```
abstract class Ave {
  void comer();
}

abstract class AveQueVoa extends Ave {
  void voar();
}

class Pardal implements AveQueVoa {
  @override
  void comer() {
    print("O Pardal está comendo.");
  }

  @override
  void voar() {
    print("O Pardal voa.");
  }
}

class Avestruz implements Ave {
  @override
  void comer() {
    print("Avestruz está comendo.");
  }
}

void main() {
  AveQueVoa pardal = Pardal();
  pardal.voar(); // Funciona como esperado
}
```

4) refatore o codigo e aplique os principios solid:

```
class Relatorio {
  void gerarRelatorio() {
    // Geração do relatório
  }

  void enviarRelatorioPorEmail() {
    // Enviar relatório por e-mail
  }

  void salvarRelatorioNoBanco() {
    // Salvar relatório no banco de dados
  }
}
```

R=

```
void main() {
  GeradorRelatorio geradorRelatorio = new GeradorRelatorio();
  geradorRelatorio.gerarRelatorio();
}

class GeradorRelatorio {
  void gerarRelatorio() {
    print('gerou o relatorio');
  }
}

class EnviarRelatorioPorEmail {
  void enviarRelatorioPorEmail() {
    print('relatorio enviado por email');
  }
}

class SalvarRelatorioNoBanco {
  void salvarRelatorioNoBanco() {
    print('relatorio salvo no banco');
  }
}
```

5) refatore o codigo abaixo e aplique os principios solid:

```
class Pagamento {
  void processarPagamento(String tipoPagamento) {
    if (tipoPagamento == "cartao") {
      // Lógica de pagamento com cartão
    } else if (tipoPagamento == "boleto") {
```

```
    // Lógica de pagamento com boleto
  }
 }
}
```

R=

```dart
abstract class Pagamento {
  void processarPagamento();
}

class PagamentoCartao implements Pagamento {
  @override
  void processarPagamento() {
    print('Processando pagamento com cartão.');
  }
}

class PagamentoBoleto implements Pagamento {
  @override
  void processarPagamento() {
    print('Processando pagamento com boleto.');
  }
}

class GerenciadorPagamento {
  void processar(Pagamento pagamento) {
    pagamento.processarPagamento();
  }
}

void main() {
  GerenciadorPagamento gerenciador = GerenciadorPagamento();

  Pagamento pagamentoCartao = PagamentoCartao();
  gerenciador.processar(pagamentoCartao);

  Pagamento pagamentoBoleto = PagamentoBoleto();
  gerenciador.processar(pagamentoBoleto);
}
```