Solid

- The SOLID principles are a set of design principles that aim to create more maintainable and scalable software. They were introduced by Robert C. Martin and are widely used in object-oriented programming.
- Here's a brief overview of each principle and how they can be applied in Dart:

1- Single Responsibility Principle (SRP):

- A class should have only one reason to change, meaning it should have only one responsibility.

Bad example (violating SRP):

```
class FileHandler {
  void saveToFile(String data) {
    // Save to file logic
  }

  void processUserData(String userData) {
    // Process user data logic
  }
}
```

- The FileHandler class has two responsibilities: saving data to a file and processing user data. This violates SRP as a class should have only one reason to change.
- This design could lead to issues when modifications are needed, as changes to one responsibility might affect the other.

Good example (following SRP):

```
class FileSaver {
  void saveToFile(String data) {
    // Save to file logic
  }
}

class UserDataProcessor {
  void processUserData(String userData) {
    // Process user data logic
  }
}
```

- The responsibilities are separated into two classes: FileSaver and UserDataProcessor.
- Each class now has a single responsibility, making the code more maintainable and easier to understand.

2- Open/Closed Principle (OCP):

- Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

Bad example (violating OCP):

```
class Rectangle {
  double width;
  double height;

  double area() {
    return width * height;
  }
}
```

- The Rectangle class is not closed for modification because adding support for new shapes requires modifying the existing class.
- This design makes it hard to extend the code without altering existing implementations.

Good example (following OCP):

```
abstract class Shape {
  double area();
}

class Rectangle implements Shape {
  double width;
  double height;

  @override
  double area() {
    return width * height;
  }
}
```

- The Shape interface is introduced, allowing for easy extension by creating new classes that implement the interface.
- The Rectangle class now follows OCP, as it can be extended without modifying existing code.

3- Liskov Substitution Principle (LSP):

- Subtypes must be substitutable for their base types without altering the correctness of the program.

Bad example (violating LSP):

```
class Bird {
  void fly() {
    print('Bird is flying');
  }
}

class Penguin extends Bird {
  @override
  void fly() {
    // Penguins can't fly
    print('Penguin cannot fly');
  }
}
```

- The Penguin class, a subtype of Bird, overrides the fly method to indicate that penguins cannot fly.
- This violates LSP as it changes the expected behavior of the base class.

Good example (following LSP):

```
abstract class Bird {
  void fly();
}

class Sparrow implements Bird {
  @override
  void fly() {
    print('Sparrow is flying');
  }
}
```

- The Sparrow class follows LSP by maintaining the expected behavior of the base class (Bird), which is that all birds can fly.

4- Interface Segregation Principle (ISP):

- A class should not be forced to implement interfaces it does not use.

Bad example (violating ISP):

```
abstract class Worker {
  void work();
  void eat();
}

class Robot implements Worker {
  @override
  void work() {
    print('Robot is working');
  }

  @override
  void eat() {
    // Robots don't eat
    print('Robot cannot eat');
  }
}
```

- The Worker interface forces implementing classes to have both work and eat methods.
- The Robot class, being a worker, is forced to implement the eat method, which is unnecessary.

Good example (following ISP):

```
abstract class Worker {
  void work();
}

abstract class Eater {
  void eat();
}

class Robot implements Worker {
  @override
  void work() {
    print('Robot is working');
  }
}
```

- The Worker and Eater interfaces are introduced, allowing classes to implement only the methods relevant to their responsibilities.
- The Robot class now only implements the work method, adhering to ISP.

5- Dependency Inversion Principle (DIP):

- High-level modules should not depend on low-level modules. Both should depend on abstractions.

Bad example (violating DIP):

```
class LightBulb {
  void turnOn() {
    print('LightBulb is on');
  }

  void turnOff() {
    print('LightBulb is off');
  }
}

class Switch {
  LightBulb bulb;

  Switch(this.bulb);

  void operate() {
    // Operate logic
    bulb.turnOn();
  }
}
```

- The Switch class depends on the concrete implementation (LightBulb) rather than an abstraction.
- This makes the code more rigid and harder to change, violating DIP.

Good example (following DIP):

```
abstract class Switchable {
  void turnOn();
  void turnOff();
}

class LightBulb implements Switchable {
  @override
  void turnOn() {
    print('LightBulb is on');
  }

  @override
  void turnOff() {
    print('LightBulb is off');
  }
}

class Switch {
  Switchable device;

  Switch(this.device);

  void operate() {
    // Operate logic
    device.turnOn();
  }
}
```

- The Switchable interface is introduced as an abstraction for devices that can be switched on and off.
- The Switch class now depends on the abstraction (Switchable), adhering to DIP and making it more flexible to changes in devices.

Applying these SOLID principles helps in creating code that is more modular, maintainable, and adaptable to changes in requirements without affecting existing functionality.