The SOLID principles are a set of five design principles that help developers create more maintainable, flexible, and scalable software. Although these principles apply to object-oriented programming in general, you can easily apply them when writing Dart code.

1. S - Single Responsibility Principle (SRP)

Definition: A class should have only one reason to change, meaning it should have only one job or responsibility.

Example:

```
class Report {
  void generateReport() {
    // Code to generate the report
  }
}

class ReportSaver {
  void saveReport(Report report) {
    // Code to save the report
  }
}
```

Explanation: The `Report` class is responsible only for generating the report, while `ReportSaver` handles saving it. Each class has a single responsibility.

2. O - Open/Closed Principle (OCP)

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

Example:

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

class Circle extends Shape {
  double radius;
  Circle(this.radius);
```

```
  @override
  double area() => 3.14 * radius * radius;
}

class Square extends Shape {
  double side;
  Square(this.side);

  @override
  double area() => side * side;
}

double calculateArea(Shape shape) {
  return shape.area();
}
```

Explanation: The `Shape` class is open for extension (you can create new shapes like `Circle` and `Square`), but the `calculateArea` function doesn't need to be modified when a new shape is added.

3. L - Liskov Substitution Principle (LSP)

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

Example:

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

class Sparrow extends Bird {}

class Penguin extends Bird {
  @override
  void fly() {
    throw Exception('Penguins can\'t fly');
  }
}
```

Violation: Here, `Penguin` violates LSP because it can't substitute `Bird` without causing an error. A better approach is to refactor the design to avoid this issue, perhaps by introducing a `FlightlessBird` class.

4. I - Interface Segregation Principle (ISP)

Definition: Clients should not be forced to depend on interfaces they do not use.

Example:

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

abstract class Eater {
  void eat();
}

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

class Human implements Worker, Eater {
  @override
  void work() {
    print('Human working');
  }

  @override
  void eat() {
    print('Human eating');
  }
}
```

Explanation: The `Worker` and `Eater` interfaces are separated so that a `Robot` only implements what it needs (`work`), while a `Human` can implement both `work` and `eat` .

5. D - Dependency Inversion Principle (DIP)

Definition: High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

Example:

```
abstract class Keyboard {
  void type();
}

class MechanicalKeyboard implements Keyboard {
  @override
  void type() {
    print('Typing on a mechanical keyboard');
  }
}

class Computer {
  final Keyboard keyboard;

  Computer(this.keyboard);

  void start() {
    keyboard.type();
  }
}

void main() {
  var keyboard = MechanicalKeyboard();
  var computer = Computer(keyboard);
  computer.start(); // Outputs: Typing on a mechanical keyboard
}
```

Explanation: The `Computer` class depends on the `Keyboard` abstraction rather than a concrete implementation, making it flexible to use any type of keyboard without modifying the `Computer` class.

Summary:

- **SRP**: One class, one responsibility.
- **OCP**: Open for extension, closed for modification.
- **LSP**: Subtypes should be replaceable with their base types.
- **ISP**: Prefer smaller, more specific interfaces.

- **DIP**: Depend on abstractions, not concretions.

Applying these principles helps in building software that is easier to maintain, extend, and refactor.