SOLID Principles

SOLID Principles is an acronym for five famous design principles, Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion.

1- **Single Responsibility Principle:**
Abbreviated with SRP, this principle states (very intuitively) that a class should only have a single reason to change should create classes with a single responsibility so that they're easier to maintain and harder to break.
example:

```
// Calculations and logic
abstract class Shape {
  double area();
}

class Square extends Shape {}
class Circle extends Shape {}
class Rectangle extends Shape {}

// UI painting
class ShapePainter {
  void paintSquare(Canvas c) {
    /* ... */ }
  void paintCircle(Canvas c) {
    /* ... */ }
  void paintTriangle(Canvas c) {
    /* ... */ }
}

// Networking
class ShapesOnline {
```

```
  String wikiArticle(String figure) {
    /* ... */ }
  void _cacheElements(String text) {
    /* ... */ }
}
```

2- Open-Closed Principle:

states that in good architecture the developer should add new behaviors without changing the existing source code. This concept is notoriously described as classes should be open for extension and closed for changes.

example:

```
abstract class Area {
  const Area();

  double computeArea();
}

class Rectangle extends Area {
  final double width;
  final double height;
  const Rectangle(this.width, this.height);

  @override
  double computeArea() => width * height;
}

class Circle extends Area {
  final double radius;
  const Circle(this.radius);

  @override
  double computeArea() => radius * radius * 3.1415;
}
```

```
class AreaCalculator {
  double calculate(Area shape) => shape.computeArea();
}
```

3- **Liskov Substitution Principle:**
   The LSP states that subclasses should be replaced with superclasses without changing the logical correctness of the program. In simpler terms, it means that a subtype must guarantee the usage conditions of its supertype plus some more behaviors.

4- **Interface Segregation Principle:**
   This principle states that clients don't have to implement a behavior they don't need The gist of this principle is you should create small interfaces with minimal methods.

5- **Dependency Inversion Principle:**
   The DIP states that we should favor abstractions over implementations Extending an abstract class or implementing an interface is good but descending from a concrete class is bad.

Reference [https://www.topcoder.com/thrive/articles/solid-principles-in-dart].