

Being SOLID in Dart.

 [ankitgg66.medium.com/being-solid-in-dart-1933037168ae](https://medium.com/@ankitgg66/being-solid-in-dart-1933037168ae)

April 12, 2021



Dart

Ever since my journey to coding started my only goal was to become a better and better programmer day by day, But during the time I found myself becoming good at problem-solving but lacked the quality of Software Development

And the Day arrived and my mentor/friend introduced me to the concept of SOLID Principles of Software along with Design Patterns (Our only focus, for now, is SOLID Principles), and I thought let's give this a try and my search was over. I have found what I was really looking for.

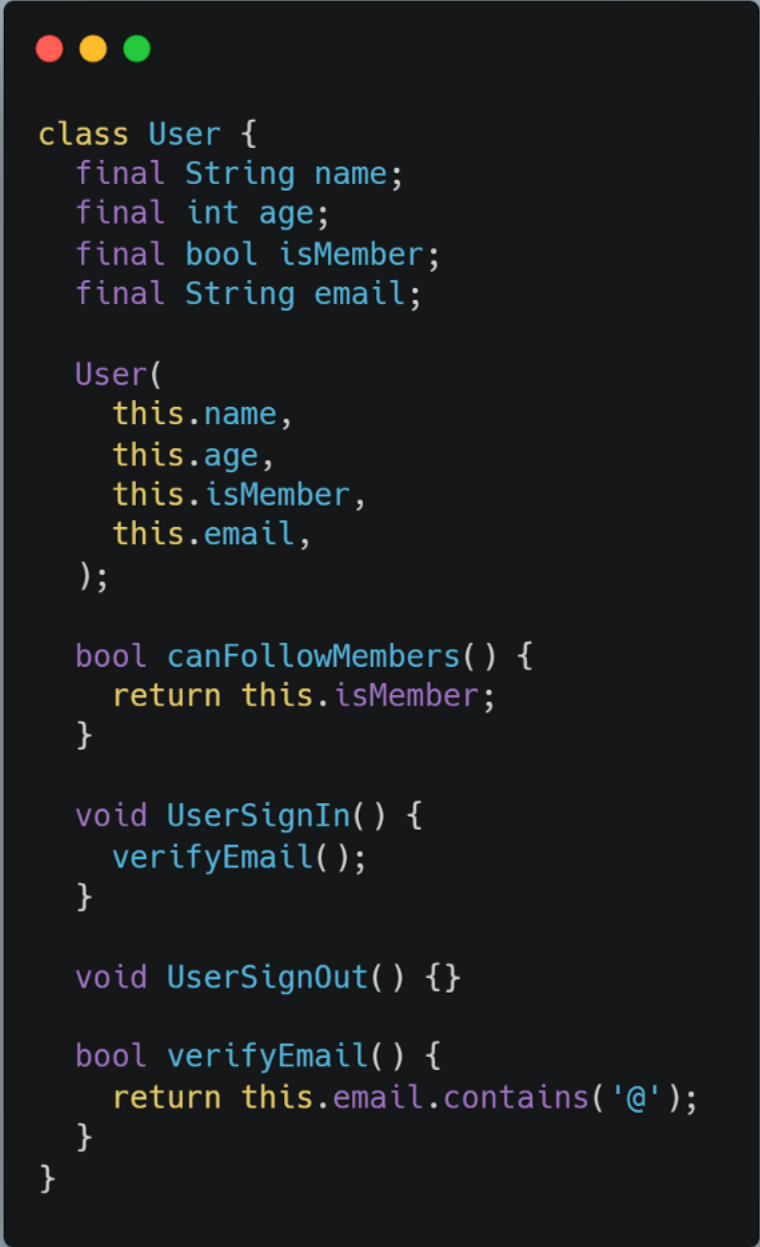
So why SOLID is really a big deal in Development well, in this case, I recall my early days of coding where many time even for small changes need to be made in code I need to face devastating consequences as I need to go to different usage to refactor my code to make it work, everything was ok for me at point of time but I really had an instinct as this is a bad practice and my code didn't scale and even was not maintainable at as a change needed to be made many parts of code turned to be scrap code.

Well SOLID makes out code flexible, maintainable, reusable this helps to avoid a large part of our code suddenly turned into scrap code.

This SOLID actually comprises 5 individual principles defined by each letter of the word SOLID.

1. SRP — Single Responsibility principle

Ok Now, what this really means. Let's have a look.



```
class User {
    final String name;
    final int age;
    final bool isMember;
    final String email;

    User(
        this.name,
        this.age,
        this.isMember,
        this.email,
    );

    bool canFollowMembers() {
        return this.isMember;
    }

    void UserSignIn() {
        verifyEmail();
    }

    void UserSignOut() {}

    bool verifyEmail() {
        return this.email.contains('@');
    }
}
```

A Generic User Class with all responsibility packed in a single class.

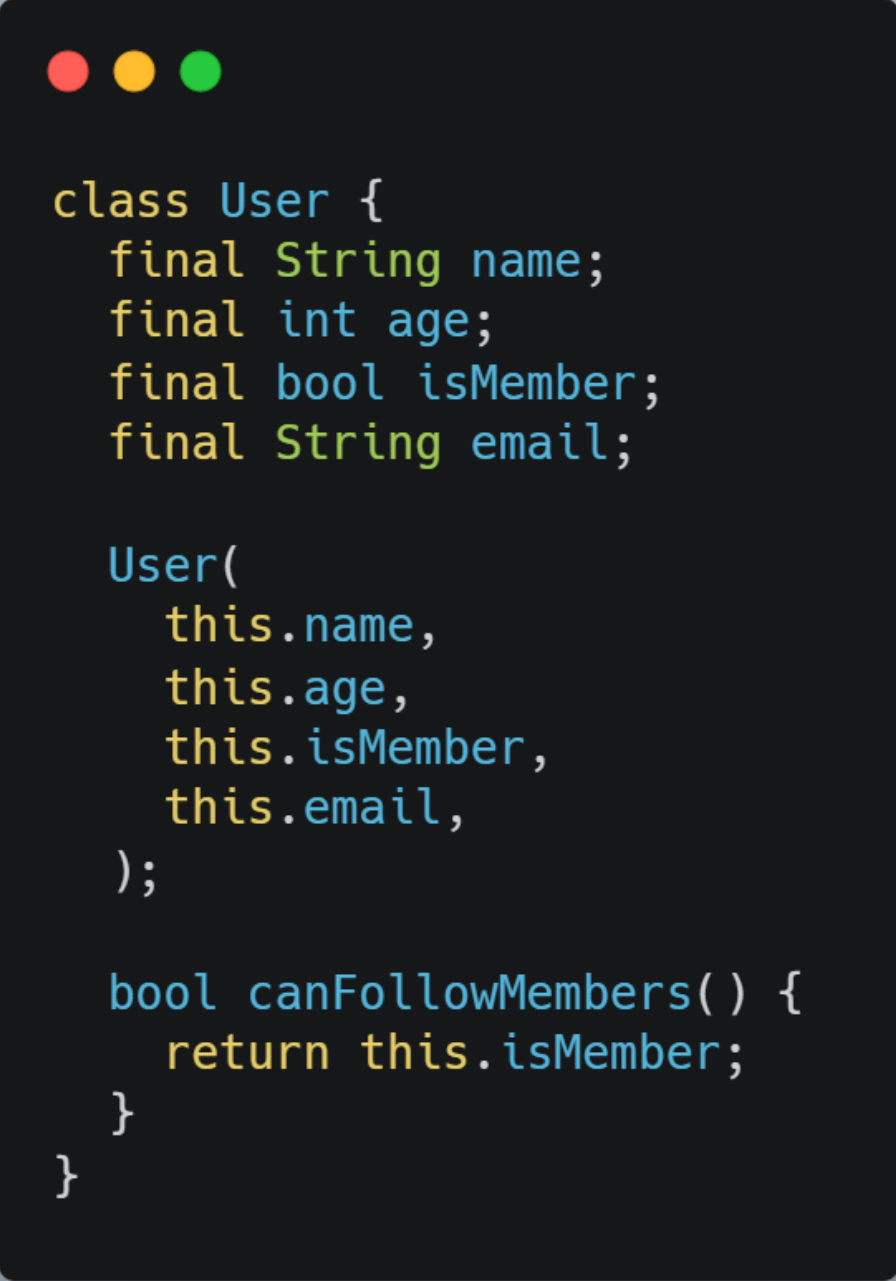
Seeing our User class we know that the User class should only be responsible for handling user details but here it does a lot more than that, we also have imposed the user authentication with validator for email.

Does this stand strong in favor of the Single Responsibility Principle?

Does this implementation have a single reason to change in the future?


No, Acc to this principle the best case must be to make separate classes for user details, user authentication, and validators.

So now let's refactor to fit the principle.



```
class User {  
    final String name;  
    final int age;  
    final bool isMember;  
    final String email;  
  
    User(  
        this.name,  
        this.age,  
        this.isMember,  
        this.email,  
    );  
  
    bool canFollowMembers() {  
        return this.isMember;  
    }  
}
```

User class



```
mixin Validators {  
  bool verifyEmail(String email) {  
    return email.contains('@');  
  }  
}
```



```
class UserAuth with Validators {  
  
  void UserSignIn(User user) {  
    verifyEmail(user.email);  
  }  
  
  void UserSignOut(User user) {}  
}
```

Validators and UserAuth.

After this, we have 2 classes with one mixin in the dart, Now as we can look each and every chunk of code is responsible for individual change and their responsibility is isolated so it becomes easy to maintain and extend User details, UserAuth, and

Validators in future if demand arises.

with this let's move to the next principle.

2. OCP — Open Closed Principle

This aims for extending its behavior but avoids any modification in source code.

Let's start with an example.

```
void main() {
    Rectangle rect = Rectangle(12.0, 18.0);
    Area a = Area(rect);
    a.calculateArea();
}

class Area {
    final Rectangle shape;
    Area(this.shape);

    void calculateArea() {
        print(shape.calculateArea());
    }
}

class Rectangle {
    final double width;
    final double height;

    Rectangle(this.width, this.height);

    double calculateArea() {
        print("I am Rectangle");
        return width * height;
    }
}
```

Here class rectangle represents a shape and the only responsibility of our area class is to print the calculated area by rectangle class.

Till now, everything looks cool but let's say there is a need to add a new shape to our code say Square and Area class should work with both shapes, well in this case our code gets modified too.

```

void main() {
    Rectangle rect = Rectangle(12.0, 18.0);
    Area a = Area(rect);
    a.calculateArea();
    Square sq = Square(12.0);
    Area area = Area(sq);
    area.calculateArea();
}

class Area {
    final Object shape;

    Area(this.shape);

    void calculateArea() {
        if (shape is Rectangle) {
            Rectangle rect = shape as Rectangle;
            print(rect.calculateArea());
        } else if (shape is Square) {
            Square sq = shape as Square;
            print(sq.calculateArea());
        } else {}
    }
}

class Square {
    final double side;
    Square(this.side);
    double calculateArea() {
        print("I am Square");
        return side * side;
    }
}

class Rectangle {
    final double width;
    final double height;

    Rectangle(this.width, this.height);

    double calculateArea() {
        print("I am Rectangle");
        return width * height;
    }
}

```

Adding a new shape.

Ok, this was some messy code as calculateArea() in Area class was implemented using an if-else_if-else and this if-else ladder would grow as every time a new shape is added to our source code and here we have violated the open-closed principle as we needed to modify our Area class.

So now, let's implement this by governing the principle of open-Closed.


```

void main() {
    Rectangle rect = Rectangle(12.0, 18.0);
    Area a = Area(rect);
    a.calculateArea();
    Shape sq = Square(12.0);
    Area area = Area(sq);
    area.calculateArea();
}

class Area {
    final Shape shape;

    Area(this.shape);

    void calculateArea() {
        print(shape.calculateArea());
    }
}

abstract class Shape {
    double calculateArea();
}

class Rectangle implements Shape {
    final double width;
    final double height;

    Rectangle(this.width, this.height);

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

class Square implements Shape {
    final double side;
    Square(this.side);

    @override
    double calculateArea() {
        return side * side;
    }
}

class Triangle implements Shape {
    final double base;
    final double height;

    Triangle(this.base, this.height);
    @override

```

```
double calculateArea() {  
    return 0.5 * base * height;  
}  
}
```

OCP

Ok now, we implemented our Shape as an abstract class, and every time we add a new Shape class to our code our Area class is not modified i.e we are open to adding new shapes by implementing Shape but closed to modify our Area class.

So moving to our next principle.

3. LSP — Liskov's Substitution Principle

This principle stands for the statement that .

This means, If is the subclass of then should be able to replace the object of in any use-case e.g passed as a parameter in a method so even if we pass the which is A's child classthe method and its behavior should not break giving a weird output or throw errors).

So let's walk through the code.

```

void main() {
    Rectangle rect = Rectangle(8, 5);
    print(rect.calculateArea()); // OP = 40
    Square square = Square(5, 5);
    print(square.calculateArea()); // OP = 25
    Rectangle sqr = Square(8, 5);
    print(sqr.calculateArea()); // OP = 64
}

class Rectangle {
    final int width;
    final int height;

    Rectangle(this.width, this.height);

    int calculateArea() {
        return width * height;
    }
}

class Square extends Rectangle {
    final int width;
    final int height;

    Square(this.width, this.height) : super(width, height);

    @override
    int calculateArea() {
        return width*width;
    }
}

```

We know a Square is a rectangle with width = height, But as we extend our Square class to a rectangle we need to change the definition of calculateArea() for square. According to Liskov's substitution principle child class should be able to replace parent class without breaking the behavior of the implementation but here when we replace the Rectangle object with the child Square object in the third assignment and we see a weird result.

```

Rectangle sqr = Square(8, 5); print(sqr.calculateArea()); -- OUTPUT = 64    [
Expected OUTPUT -- 40 ]

```

Ok, we have broken the principle because as we expect a Rectangle with a result of 40 and also Square class should be accepting a single parameter of side.

```

void main() {
    Shape rect = Rectangle(8, 5);
    print(rect.calculateArea()); // OP = 40
    Shape square = Square(5);
    print(square.calculateArea()); // OP = 25
    Shape squre = Square(8);
    print(squre.calculateArea()); // OP = 64
}

abstract class Shape {
    int calculateArea();
}

class Rectangle implements Shape {
    final int width;
    final int height;

    Rectangle(this.width, this.height);

    @override
    int calculateArea() {
        return width * height;
    }
}

class Square implements Shape {
    final int side;

    Square(this.side);

    @override
    int calculateArea() {
        return side * side;
    }
}

```

Applying Liskov's substitution principle

No matter what we assign to Shape be Rectangle or Square we can see that it maintains its behavior.

Now moving to the next principle.

4. ISP — Interface Segregation Principle

simple and straight If something is useless for me I should not be forced to keep it.

So let's see what the problem is?

```

void main() {
    Shape cube = Cube(3);
    print(cube.calculateArea());
    print(cube.calculateSurfaceArea());
    print(cube.calculateVolume());

    Shape rectangle = Rectangle(5, 8);
    print(rectangle.calculateArea());
    print(rectangle.calculateSurfaceArea());
    print(rectangle.calculateVolume());
}

abstract class Shape {
    double calculateArea();
    double calculateSurfaceArea();
    double calculateVolume();
}

class Rectangle implements Shape {
    final double width;
    final double height;

    Rectangle(this.width, this.height);

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

    @override
    double calculateSurfaceArea() {
        throw Exception('I am a 2 Dimensional object. No Surface Area');
    }

    @override
    double calculateVolume() {
        throw Exception('I am a 2 Dimensional object. Has No Volume');
    }
}

class Cube implements Shape {
    final double side;

    Cube(this.side);

    @override
    double calculateArea() {
        throw Exception('I am a 3 Dimensional object. I have Surface Area');
    }

    @override
    double calculateSurfaceArea() {
        return 6 * (side * side);
    }

    @override
    double calculateVolume() {
        return side * side * side;
    }
}

```



Forcing unnecessary methods.

A Rectangle is a 2 Dimensional Shape and Cube is 3 Dimensional Shape Rectangle that has Area but as it is a 2-dimensional shape surface area and volume is useless for its implementation similarly Cube is 3 dimensional so the area is useless for its implementation and since both of the class implements from the shape abstract class, just to make compiler happy we have imposed these methods which are *irrelevant to them*.

extract of Interface Segregation Principle.

So let's refactor our code by keeping **ISP** in mind.

```

void main() {
    ThreeDimensionalShape cube = Cube(3);
    print(cube.calculateSurfaceArea());
    print(cube.calculateVolume());

    TwoDimensionalShape rectangle = Rectangle(5, 8);
    print(rectangle.calculateArea());
}

abstract class ThreeDimensionalShape {
    double calculateSurfaceArea();
    double calculateVolume();
}

abstract class TwoDimensionalShape {
    double calculateArea();
}

class Rectangle implements TwoDimensionalShape {
    final double width;
    final double height;

    Rectangle(this.width, this.height);

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

class Cube implements ThreeDimensionalShape {
    final double side;

    Cube(this.side);

    @override
    double calculateSurfaceArea() {
        return 6 * (side * side);
    }

    @override
    double calculateVolume() {
        return side * side * side;
    }
}

```

Implementing ISP

Removing the single generic class we have separated this into two client-specific abstract classes and freed the clients from implementing usable methods.

Now finally moving to our last principle.

5. DIP — Dependency Inversion Principle

-
-

This principle is all about the loose coupling of your classes making them more localized and connecting them with abstract class. It's better to connect the high-level class to the low level via abstract class. Due to this, we will encounter less work to make changes in either of the classes when required, affecting fewer code blocks.

and further for the second part.

This principle really helps to maintain our code for future modification at a lower hectic.

Ok now let's see what actually it helps in.

```

void main() {
    UserBloc userBloc = UserBloc();
    userBloc.getUserByID();
    userBloc.getTopTenUserPosts();
}

class UserBloc {
    final NetworkCalls _networkCalls = NetworkCalls();

    void getUserByID() {
        _networkCalls.getUserByID(1);
    }

    void getTopTenUserPosts() {
        _networkCalls.getTopTenUserPosts();
    }
}


class NetworkCalls {
    Future getUserByID(int id) async {
        print('$id');
    }

    Future getTopTenUserPosts() async {
        print("One to Ten");
    }
}

```

Okay, what's wrong with this implementation actually nothing but the only thing is that we have declared using a concrete class and both the classes are tightly coupled with each other that is our UserBloc higher class is depends on the lower class of NetworkCalls which violets the DIP.

Now let's implement this using DIP in mind.



```

void main() {

    // Here we initialize NetworkTask as NetworkCall
    // and pass as a param to UserBloc
    NetworkTask networkTask = NetworkCalls();
    UserBloc userBloc = UserBloc(networkTask);

    userBloc.getUserByID();
    userBloc.getTopTenUserPosts();
}

class UserBloc {
    // Here UserBloc depends on abstraction rather
    //than concrete class for its implementation

    final NetworkTask networkTask;
    UserBloc(this.networkTask);

    void getUserByID() {
        networkTask.getUserByID(1);
    }

    void getTopTenUserPosts() {
        networkTask.getTopTenUserPosts();
    }
}

abstract class NetworkTask {
    Future getUserByID(int id);
    Future getTopTenUserPosts();
}

class NetworkCalls implements NetworkTask {
    Future getUserByID(int id) async {
        print('$id');
    }

    Future getTopTenUserPosts() async {
        print("One to Ten");
    }
}

```

DIP implementation

Now we have loosely coupled our UserBloc higher level class to lower-level class NetworkCalls via NetworkTask abstract class and the dependency of Network Task is passed as a parameter while instantiating the UserBloc in the first place.

Here UserBloc depends on abstraction rather than concrete class for its implementation.

Start implementing things in your personal projects to get a real hand practice because it takes some time to take a good grasp on this topic.

So, at last, there is no strong and concrete rule to apply this pattern the only thing this does is make our life easy when the point comes to maintain and reuse our code and gives us a feeling of being a developer So start your first step towards being a Pro Dev and if you are already, do share this with the passionate beings who want to take their first step in this journey.