The SOLID Principles

As object oriented programs grow large in size, it gets more essential to split that system apart into little pieces.

But what does ‘split apart’ mean? And what is a ‘little piece’, exactly?

The five design principles known as SOLID came about as solutions to this. They have some pretty indecipherable names but are easy enough - once you understand what problems they solve.

The five SOLID principles

Let’s just list them out, so you know what the acronym means:

- **SRP** Single Responsibility Principle
- **OCP** Open/Closed Principle
- **LSP** Liskov Substitution Principle
- **ISP** Interface Segregation Principle
- **DIP** Dependency Inversion Principle

You can see the acronym in the first letters.

The titles of each principle are jargon in the proper sense. Once you know what each one means, the short titles make sense. But until then, they won’t mean much.

It’s important to note that the order of these is ‘what sounds best’ - not ‘what is most useful’. So, unlike every other book on SOLID, I’m going to explain the principles in order of usefulness - “SDLOI”. I’ll give you that it doesn’t roll off the tongue as easily.

SRP Single Responsibility - do one thing well

Code that does many things is hard to understand.

If we have a class that opens files, reads lines of text, does a spelling check, blanks out passwords, then writes an HTML page - it's a headache.

It's doing more than one thing.

The solution is to split this up into separate classes where *each one does one thing*.

That is the SRP. Give a Class one single responsibility.

It is both the simplest and most widely used of the SOLID principles. Unfortunately, it is also the most difficult to pin down.

What is 'one thing', anyway?

The problem with it is in deciding what 'one thing' is.

I'm writing this on a web-based Word Processor. To me, it is 'just one thing' - the thing I type on.

But inside, it will have many different moving parts: Text management, storage management, web input and output, to name a few.

It's hard to create any rules around this. When we look at 'Hexagonal Architecture', we'll see some things that help us figure out single pieces, by splitting out anything concerned with an external system.

For most of our code, though, a good rule is to look at the language we use in our test names.

Do they talk about 'the same thing'?

As an example, look at our test class for our Bill object from earlier, the BillTest.

It has methods relating to adding up different total values for different items on a bill.

That's doing one thing.

If it started having tests for `createsHtmlWithUnderlinedTotal()` or `savesTotalToDatabase()` then we can see that our Bill class has grown some extra responsibilities.

The solution is simply to refactor those extra responsibilities into their own classes. Go back to basics and redesign your object collaborations.

But before we can do that, we're going to have to look at our next SOLID principle: Dependency Inversion.

DIP Dependency Inversion: Bring out the Big Picture

This is the most important and most useful idea of SOLID.

By Inverting Dependencies, you create code by assembling swappable 'plugin' style components.

This gets you two big benefits:

- Changes in one component do not ripple through to any others
- You can replace components with ones designed to help TDD tests

What is an 'inverted dependency'?

The easiest way to understand this is to take some code without it, see what problems it has, then refactor it.

Let's start with a simple method that lives in a class somewhere. Its responsibility is to take some keyboard input, make it all upper case, then display it on the console:

```
public void showInputInUpperCase() {

    // Fetch Keyboard Input
    Scanner scanner = new Scanner(System.in);
    String inputText = scanner.nextLine();

    // Convert
    String upperCaseText = inputText.toUpperCase();

    // Display
    System.out.println(upperCaseText);
}
```

There's nothing remarkable about it, at first sight. We see the three sections to fetch input, convert, display.

But notice that new keyword. That's a problem.

Because of this new, the method has a *direct dependency* on how the keyboard is read and how the output is displayed.

It also calls System.out.println() for output, which has the same problem. We are tightly bound to a global method. But let's just consider input for now.

Why is 'new' such a problem?

The new keyword itself is not a problem. It is *where it is in the code.*

Dependencies like this are hard-coded. They cannot be changed.

If we wanted a different input method, we would have to go into this method and change the code. We would delete the two lines relating to Scanner, and replace them with something else.

This doesn't sound too bad for a four line method. But as a design principle, it is exactly what we don't want. We don't want it so that unrelated areas of code have to change.

Think about it this way. What is the main purpose of our little function? It is to change something into upper case. Yes, it needs some input from somewhere to do that. But the function does not need to know anything at all about how that input is obtained.

This is where TDD really forces a good decision early.

How could we write a unit test for our function? Spoiler: We couldn't.

We would have to admit defeat. We don't have a way for the test to make keyboard input at this level. We are locked-in to using a Scanner object.

But let's re-phrase that question and not give up: How *could* we write a unit test? What would we have to change?

Inverting the input Dependency

What we need is some way for this code to fetch input, without knowing anything about how that is done.

We've seen how to do that before. Our polymorphic Shape.draw() knew how to ask an object to do something without knowing anything about how.

Let's use the same technique here.

We'll introduce an interface called Input, which we can ask to give us some text:

```
1  interface Input {
2      String fetch();
3  }
```

This is an abstraction. It says 'you can ask me for an input string, but leave it to me to figure out where it will come from'.

It replaces the two lines of code relating to the scanner in our example code.

```
public void showInputInUpperCase() {
    String inputText = input.fetch();

    // Convert
    String upperCaseText = inputText.toUpperCase();

    // Display
    System.out.println(upperCaseText);
}
```

I've also removed the comment. The interface makes it clear enough without.

As it stands, this would not compile.

We need to make two more changes:

- Create an implementation of this interface
- Add a parameter or field 'input' - so our method can use the interface

Making a concrete KeyboardInput class

We can easily make our first implementation of this interface:

```
class KeyboardInput implements Input {
    @Override
    public String fetch() {
          Scanner scanner = new Scanner(System.in);
          String inputText = scanner.nextLine();

          return inputText;
    }
}
```

We've copied-and-pasted the Scanner code into a new class that implements the interface. It is called 'KeyboardInput' because this class is allowed to know exactly where the input comes from.

This class is what is meant by inverting a dependency.

The old method newed up the Scanner inside the method. It depended on Scanner.

The new method code depends only on the interface. The new KeyboardInput class wraps up Scanner and also depends on the interface. That's the inversion part.

We've changed the dependency from "method depends on Scanner" to "class containing Scanner depends on interface".

The dependency is precisely backwards to before. It is inverted.

This thought process is known as the **DIP** Dependency Inversion Principle.

Dependency Injection - using our inverted dependency

Our original method needs a way to use this inverted dependency.

The most typical way is to inject it into a constructor:

```
1  class TextConversion {
2      private Input input ;
3
4      // This is where we inject the dependency
5      public TextConversion(final Input input) {
6          this.input = input;
7      }
8
9      public void showInputInUpperCase() {
10         String inputText = input.fetch();
11
12         // Convert
13         String upperCaseText = inputText.toUpperCase();
14
15         // Display
16         System.out.println(upperCaseText);
17     }
18 }
```

and we would build a little application to wire it up and run it:

```
public class TextUtility {
    public static void main(String[] commandLineArgs) {
        Input input = new KeyboardInput();

        new TextConversion(input).showInputInUpperCase();
    }
}
```

Running this, we get our original behaviour. This has been a refactoring exercise, not a feature change.

Can you see how we have *externalised* the details of keyboard input from the actual text conversion?

Swappable input sources

Here is the most remarkable thing about this. This design isolates changes to the input source.

If we decided to change from using the keyboard to using a database, we would not change the TextConversion class in any way. Nor would we change the keyboard input class. Which means we wouldn't break anything we'd already written.

We would write a new concrete class called DatabaseInput that implemented the Input interface. We would wire that up in our application instead of the keyboard input:

```
public class TextUtility {
    public static void main(String[] commandLineArgs) {
        Input input = new DatabaseInput();

        new TextConversion(input).showInputInUpperCase();
    }
}
```

You can see nothing has changed except for the class name we create with new.

Inverting the output to display

In the same way as abstracting out the input source, we can do the same thing for display output:

Create the interface:

```
interface Output {
  void display( String toDisplay );
}
```

Create a concrete class to display to System out:

```
class ConsoleOutput implements Output {
  @Override
  public void display( String message ) {
    System.out.println( message );
  }
}
```

Inject both Input and Output dependencies in the constructor:

```
class TextConversion {
    private Input input ;
    private Output output ;

    public TextConversion(final Input input, final Output output){
        this.input = input;
        this.output = output;
    }

    public void showInputInUpperCase() {
        String inputText = input.fetch();
        String upperCaseText = inputText.toUpperCase();
        output.display(upperCaseText);
    }
}
```

And create the concrete classes in the application:

```
public class TextUtility {
    public static void main(String[] commandLineArgs) {
        Input input = new KeyboardInput();
        Output output = new ConsoleOutput();

        new TextConversion(input, output).showInputInUpperCase();
    }
}
```

That allows us to swap to any input or display source. We only change the objects that get created with new. Nothing else.

And *that* helps us with our unit testing. We can swap the input for something we can control and swap the output for something that captures anything sent to it.

We'll see how this works in the chapter on 'TDD and Test Doubles'

Inversion - Injection: two sides of the same coin

It's quite common to have heard about DIP, SOLID and DI frameworks like Spring, without ever really connecting the dots.

Dependency Inversion is the *design* technique. The thought process by which we remove a hard-coded coupling from the software.

Dependency Injection is the *implementation* technique. We have inverted a dependency. Now we need some way of choosing a suitable concrete implementation and plugging it in.

DI frameworks are often useful. But don't forget - simply new-ing up classes and passing them into a constructor is all that DI is at heart.

LSP Liskov Substitution Principle - Making things swappable

Barbara Liskov is a computing legend. LSP is the name given to an important principle she discovered.

The principle helps us reason about a problem we have in polymorphism.

Let's go back to our Shapes example to see what it is.

When Shapes go Bad

```
interface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw(){
        System.out.println("I'm round");
    }
}
```

This is the 'good' case.

A Circle implements the draw() method of interface Shape and can be used anywhere a Shape is expected.
You'll recall the example calling code which could mix together Circle, Square and Triangle without code changes.

But can every class that implements the interface method be swapped in? How about this one:

```
class CardDealer implements Shape {
    private final Deck cards ;

    CardDealer( final Deck cards) {
        this.cards = cards;
    }

    public void draw(){
        Card next = cards.next();
        System.out.println(next.asText());
    }
}
```

This might be a class for a card game; Bridge or even Snap.

Class CardDealer implements the method Shape.draw() and so it is polymorphic with it.

We can add it into our list of shapes easily enough:

```
1   List<Shape> shapes = List.of( new Circle(), new CardDealer());
2   shapes.forEach( Shape::draw );
```

That will compile and run.

But will it work *as expected*?

Substitutability

Intuitively - No. Of course it won't work as expected. CardDealer does not fit in with our expectation of what Shape.draw() should do.

The whole setup of Shape.draw() and Circle and so on suggests that we expect our classes to draw geometric shapes. Not draw cards from a deck.

This was the insight Barbara Liskov had. She went on to formalise this intuition mathematically. It is that formalism which is known as the Liskov Substitution Principle (LSP).

LSP reminds us that classes implementing interfaces are only useful if they can be substituted for that interface in all circumstances.

Specifically, a class meeting this *must*:

- Accept all possible inputs
- Produce all expected outputs for those inputs

In short, you can swap out any of the implementing classes without causing the code to break.

Liskov gave a rigorous definition as:

> if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program

Practically, I always end up asking that simple question: "Are there any cases where this class would break the interface contract?"

LSP also applies to inheritance in general - classes and subclasses, abstract classes and so on. It is not restricted to interfaces and implementations.

The only reason I describe it in those terms is that I find interface/implementation to be the simplest, safest usage of inheritance in OOP. It is 'the happy path'.

OCP Open/Closed Principle - adding without change

The Open/Closed Principle describes code that can be made to behave differently without changing that code itself.

We've seen this before in our Shapes example. The code that draws Shapes can draw any shape without needing any change itself.

```
1  class Shapes {
2      private final List<Shape> shapes = new ArrayList<>();
3
4      public void add (Shape s) {
5          shapes.add(s);
6      }
7
8      public void draw() {
9          shapes.forEach(Shape::draw);
10     }
11 }
```

This aggregate class allows you to add shapes to an inner list then draw them.

Where OCP comes in is that you can add any LSP compatible Shape implementation using the add(shape) method. You can then change the behaviour of Shapes.draw() without changing anything inside that class. The change of behaviour is done outside of the Shapes class, not inside.

This is what is meant by Shapes is Open for extension (you make it do different things), but closed for modification.

OCP is essentially Dependency Inversion in disguise.

Strategy Pattern: Externalising behaviour

There is a general-purpose pattern for OCP.

First described in the book Design Patterns by the 'Gang of Four', the Strategy pattern is a general way of allowing behaviour to be injected into a class.

Let's think of our Shapes class as being part of a bigger graphics program, like Photoshop. We want to add a way to process each Shape to our Shapes class. Our first algorithm could be to add a blur effect to each shape.

One way to do this might be to add a blur() method to Shapes:

```
1  interface Shape {
2      void draw();
3      void apply(Filter f);
4  }
5
6  class Shapes {
7      private final List<Shape> shapes = new ArrayList<>();
8
9      public void add (Shape s) {
10         shapes.add(s);
11     }
12
13     public void draw() {
14         shapes.forEach(Shape::draw);
15     }
```

```
16
17      public void blur() {
18          shapes.forEach(s -> s.apply(new BlurFilter()));
19      }
20  }
```

To support this, we have also added an apply(Filter f) method to Shape to give us a way to run the processing on each Shape. Filter is an interface. Each different kind of graphics processing must implement it. We pass in a concrete class of our choice, depending on what kind of graphics processing we want. In this example, it is a BlurFilter, which gives each shape a gentle Gaussian blur effect.

This works just fine. But we have had to modify class Shapes.

In general, we want to avoid modifying classes that we've already written. Or at least only do it for a good reason. We don't want to introduce bugs into code that already works. We certainly don't want to create giant classes with loads of methods. That's the motivation behind the Open/Closed Principle.

Adding a feature like graphics processing to Shapes is usually going to need a code change like this. Any other way of tackling it would be a stretch.

If we only need the one method blur(), this isn't a bad way to do it.

But if we needed more kinds of processing - sharpen, lighten, darken, posterise say - could we do better? Could we limit it to making only the one change and then injecting in those other behaviours?

This is what the Strategy pattern does well:

```
1   class Shapes {
2       private final List<Shape> shapes = new ArrayList<>();
3
4       public void add (Shape s) {
5           shapes.add(s);
6       }
7
8       public void draw() {
9           shapes.forEach(Shape::draw);
```

```
10      }
11
12      public void filter(Filter filter) {
13          shapes.forEach( s -> filter.applyTo(s) );
14      }
15  }
```

Here, the filter() method injects a class implementing the Filter interface into the method. It then runs it across each of the shapes.

Filter is a *Strategy Pattern*. It defines the strategy for how we filter a Shape. It externalises behaviour.

Our Shapes.filter() method has no knowledge of how exactly we process each Shape. It takes what it is given from outside the class and applies it to each Shape.

In the same way that Shapes.draw() does not know how to draw any shape, Shapes.filter() does not know how to filter any Shape. The Strategy pattern formed by our Filter interface pushes that variable behaviour outside the class, leaving it open for extension but closed for further modification.

Any future new processing will be done by creating a new class that implements Filter. This will be done outside of the Shapes class and will require no changes to it.

The Strategy pattern is a powerful way of parameterising behaviour. We use the power of polymorphism to leave our choice of behaviour open-ended.

ISP Interface Segregation Principle - honest interfaces

The Interface Segregation Principle states that calling code should not be forced to depend on methods it does not use. It's a good way of putting it.

I think of ISP as keeping interfaces as small and honest. It's about avoiding polluting an interface with methods that only sometimes make sense. The trick is to only add methods that *always* make sense. We move the 'sometimes needed' methods elsewhere to where they are always needed, then redesign to make use of them.

Bad Example: TV Controls

The hardware team proudly tell us that their latest low-cost digital TV product is ready for us to get some code into. Our first task is to implement the remote control code.

We keep it simple and write an interface exposing all there is to be controlled:

```
interface Control {
    void on();
    void off();
    void channelUp();
    void channelDown();
}
```

We can stub that out for testing our remote control adapter and create a production class that drives the TV hardware.

All good.

A couple of weeks later, the team tell us they have got some improved Smart TV hardware. We will be launching two products, one low-cost, entry-level version and our high-end Smart TV.

It features digital programme recording as well as YouTube and Netflix internet TV.

We might decide to add the extra features into our existing Control interface:

```
interface Control {
    // entry level TV features
    void on();
    void off();
    void channelUp();
    void channelDown();

    // Smart TV Only features
    void startRecording();
    void endRecording();
```

```
11      void launchYoutube();
12      void launchNetflix();
13  }
```

We start coding our class to drive the TV hardware, and we want to keep the same codebase, rather than fork it per-product.

Ah.

If we keep a single class for both entry-level and Smart TVs, we soon find a problem with the Smart TV methods on the entry-level product: they don't work.

Initial ideas are to add `if (isSmartTV)` statements everywhere needed. That itself is an OOP code smell. We should probably replace that with polymorphism, using two separate classes that implement that Control interface. Call them BasicControl and SmartTvControl.

This gets rid of the if statements. More specifically, it pushes them out to the edge of the system where we create the appropriate one of those two classes. But it still leaves us with a problem. What should BasicControl do with all the Smart-TV only methods?

Typically, these will either be no-operation (no code in there) or throw an UnsupportedOperationException.

The root cause of the problem is that we have violated ISP. We have forced the BasicControl implementer of the interface into using a bunch of methods that make absolutely no sense to it.

Fixing our ISP violation

This is where I wish I could say it was easy to fix. But stuff like this gets pretty involved or pretty hacky.

The first suggestion is to simply split the interface into two:

```
interface BasicFeatures {
    void on();
    void off();
    void channelUp();
    void channelDown();
}

interface AdvancedFeatures {
    void startRecording();
    void endRecording();
    void launchYoutube();
    void launchNetflix();
}
```

One approach would be to make AdvancedFeatures extend BasicFeatures. Above, I've chosen not to, because I don't think it helps us much.

By doing this split, we have fixed ISP in a technical sense. We can create a class (possibly two classes) that will drive the BasicFeatures on both entry-level and Smart TVs. We can create a separate class to drive the AdvancedFeatures on the Smart TV only. Each class only gets methods it cares about.

Redesigning to Command objects

In a practical system, with one codebase being used for two products with different feature sets, fixing ISP isn't enough.

The issue is that we want to expose one set of features to an entry-level TV user and a different superset of features to a Smart TV user. Java interfaces do not express that model.

A Java (and C# and C++) interface is statically bound. We can't merely add methods to it based on our TV type. It is static binding that leads us to write smaller interfaces and use ISP in the first place.

Our application ideally needs some way of modelling that an entry-level TV has three methods - on, off, selectChannel - and our Smart TV has more.

The way to do this elegantly is with a design pattern known as 'Command'.

```
interface Command {
    void execute();
}
```

You can see that a Command interface is a level of abstraction higher than we were using. It knows nothing at all about our specific application. It only knows that any objects that implement Command can be told to 'execute()' - to run their code that makes that command happen.

So we would see classes like

```
class ChannelUp implements Command {
    void execute() {
        // code to drive the hardware to show whatever
        // is on the next TV channel 'up' in the list

        // This is the same code as we would have in our
        // class that implemented BasicFeatures - it is
        // just a refactor into a different code structure
    }
}
```

We would have one class for each of our original methods in our 'big interface'.

Then, we would map each command object against something we got from the TV Remote control. Suppose our hardware team gave us a remote control driver that delivered us a number between 0 and 15 inclusive to represent each command. We would write a class that knew this mapping and could route the command number accordingly:

```
class Commands {
    private final Map<Integer, Command> commandsByNumber =
        new HashMap<Integer, Command>();

    public initialise( boolean isSmartTv ) {
        register(0, new On());
        register(1, new Off());
        register(2, new ChannelUp());
        register(3, new ChannelDown());

        if (isSmartTv) {
            register(4, new StartRecording());
            // ... register the others
        }
    }

    public void execute(int commandNumber) {
        Command c = commandsByNumber.get( commandNumber );

        if ( c != null ) {
            c.execute();
        }
    }

    private void register( int commandNumber, final Command c ){
        commandsByNumber.put( commandNumber, c );
    }
}
```

Using this, we get our 'dynamic interface' effect. The Smart TV has more commands available to it. We don't have any ISP violations, because we have designed that out. ISP is not possible here.

We initialise this class knowing what kind of TV product we have. Then we can pass our commandNumber to execute() and have this class do the right thing, Tell-Don't-Ask style.

But it seems a bit complicated …

Pragmatics: I would choose to do it wrong

Given all that, I would probably choose the solution that violates ISP.

We've only got two products. Most probably, any unsupported commands should have no effect. Given these factors, I would probably go with one big interface and no-operation methods for the Smart TV features.

I love the idea of dynamically mapped Commands at the granularity of one Command per user-visible feature. In this case, it just strikes me as over-engineering.

We're Agile, after all. So if we get yet another TV product out, then I might be tempted to refactor it into Commands then.

Just not now: YAGNI and KISS. You ain't gonna need it. Keep it simple.

Which way would you jump with this?