References:

<https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/>

<https://www.geeksforgeeks.org/solid-principle-in-programming-understand-with-real-life-examples/>

 The SOLID principle was introduced by ***Robert C. Martin***, also known as *Uncle Bob* and it is a coding standard in programming. This principle is an acronym of the five principles which is given below…

1. Single Responsibility Principle (SRP)
2. Open/Closed Principle
3. Liskov’s Substitution Principle (LSP)
4. Interface Segregation Principle (ISP)
5. Dependency Inversion Principle (DIP)

The SOLID principle helps in reducing tight coupling. Tight coupling means a group of classes are highly dependent on one another which you should avoid in your code. Opposite of tight coupling is loose coupling and your code is considered as a good code when it has loosely-coupled classes. Loosely coupled classes minimize changes in your code, helps in making code more reusable, maintainable, flexible and stable. Now let’s discuss one by one these principles…

1. **Single Responsibility Principle:**This principle states that “a class should have only one reason to change” which means every class should have a single responsibility or single job or single purpose

**Common Pitfalls and Anti-patterns**

In this section we will look at some common mistakes that violate the Single Responsibility Principle. Then we will talk about some ways to fix them.

We will look at the code for a simple bookstore invoice program as an example. Let's start by defining a book class to use in our invoice.

class Book {

String name;

String authorName;

int year;

int price;

String isbn;

public Book(String name, String authorName, int year, int price, String isbn) {

this.name = name;

this.authorName = authorName;

this.year = year;

this.price = price;

this.isbn = isbn;

}

}

This is a simple book class with some fields. Nothing fancy. I am not making fields private so that we don't need to deal with getters and setters and can focus on the logic instead.

Now let's create the invoice class which will contain the logic for creating the invoice and calculating the total price. For now, assume that our bookstore only sells books and nothing else.

public class Invoice {

private Book book;

private int quantity;

private double discountRate;

private double taxRate;

private double total;

public Invoice(Book book, int quantity, double discountRate, double taxRate) {

this.book = book;

this.quantity = quantity;

this.discountRate = discountRate;

this.taxRate = taxRate;

this.total = this.calculateTotal();

}

public double calculateTotal() {

double price = ((book.price - book.price \* discountRate) \* this.quantity);

double priceWithTaxes = price \* (1 + taxRate);

return priceWithTaxes;

}

public void printInvoice() {

System.out.println(quantity + "x " + book.name + " " + book.price + "$");

System.out.println("Discount Rate: " + discountRate);

System.out.println("Tax Rate: " + taxRate);

System.out.println("Total: " + total);

}

public void saveToFile(String filename) {

// Creates a file with given name and writes the invoice

}

}

Here is our invoice class. It also contains some fields about invoicing and 3 methods:

* **calculateTotal**method, which calculates the total price,
* **printInvoice** method, that should print the invoice to console, and
* **saveToFile**method, responsible for writing the invoice to a file.

You should give yourself a second to think about what is wrong with this class design before reading the next paragraph.

Ok so what's going on here? Our class violates the Single Responsibility Principle in multiple ways.

The first violation is the **printInvoice**method, which contains our printing logic. The SRP states that our class should only have a single reason to change, and that reason should be a change in the invoice calculation for our class.

But in this architecture, if we wanted to change the printing format, we would need to change the class. This is why we should not have printing logic mixed with business logic in the same class.

There is another method that violates the SRP in our class: the **saveToFile**method. It is also an extremely common mistake to mix persistence logic with business logic.

Don't just think in terms of writing to a file – it could be saving to a database, making an API call, or other stuff related to persistence.

So how can we fix this print function, you may ask.

We can create new classes for our printing and persistence logic so we will no longer need to modify the invoice class for those purposes.

We create 2 classes, **InvoicePrinter**and **InvoicePersistence,** and move the methods.

public class InvoicePrinter {

private Invoice invoice;

public InvoicePrinter(Invoice invoice) {

this.invoice = invoice;

}

public void print() {

System.out.println(invoice.quantity + "x " + invoice.book.name + " " + invoice.book.price + " $");

System.out.println("Discount Rate: " + invoice.discountRate);

System.out.println("Tax Rate: " + invoice.taxRate);

System.out.println("Total: " + invoice.total + " $");

}

}

public class InvoicePersistence {

Invoice invoice;

public InvoicePersistence(Invoice invoice) {

this.invoice = invoice;

}

public void saveToFile(String filename) {

// Creates a file with given name and writes the invoice

}

}

Now our class structure obeys the Single Responsibility Principle and every class is responsible for one aspect of our application. Great!

1. **Open Closed Principle**

This principle states that “software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification” which means you should be able to extend a class behavior, without modifying it.

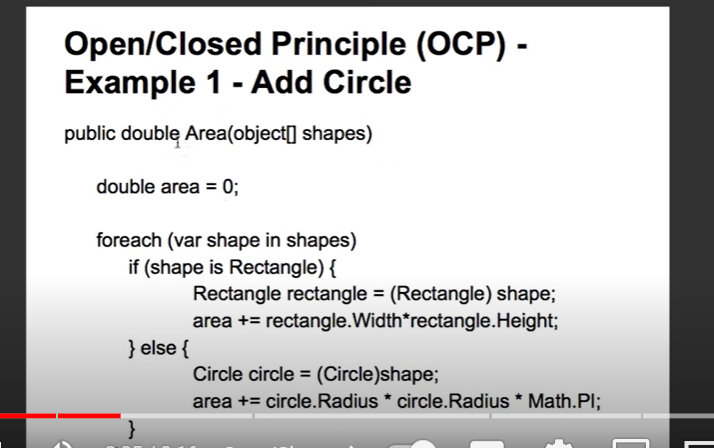
Example:

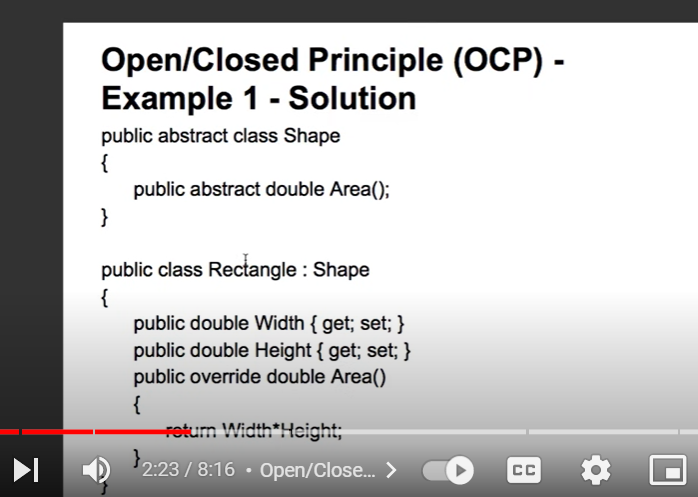
Initially we have an area method that receives an array of shapes and calculates the sum of their areas.

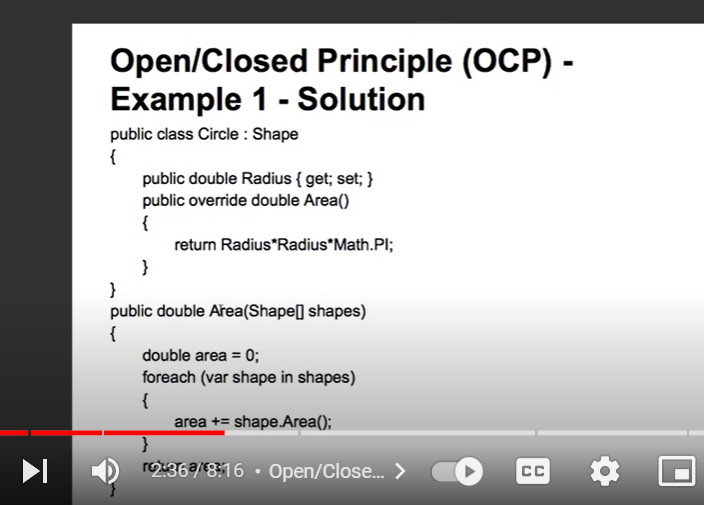
If we want to add more shapes we would need to modify that method for each shape, which is not good.

So what we do is, we create an interface Shape or an abstract class with method name area. Now each shape (circle, rectangle etc) implement their own area method. This way addition of new shapes will also not create a problem.

Now in the main method where we are calculating the sum of areas, we just call the area method of each shape and add the area. When new shape types are added, then this method does not require any change.







1. **Liskov’s Substitution Principle**

The principle was introduced by Barbara Liskov in 1987 and according to this principle “Derived or child classes must be substitutable for their base or parent classes“. This principle ensures that any class that is the child of a parent class should be usable in place of its parent without any unexpected behavior.

Liskov's principle is easy to understand but hard to detect in code. So let's look at an example.

class Rectangle {

protected int width, height;

public Rectangle() {

}

public Rectangle(int width, int height) {

this.width = width;

this.height = height;

}

public int getWidth() {

return width;

}

public void setWidth(int width) {

this.width = width;

}

public int getHeight() {

return height;

}

public void setHeight(int height) {

this.height = height;

}

public int getArea() {

return width \* height;

}

}

We have a simple Rectangle class, and a **getArea**function which returns the area of the rectangle.

Now we decide to create another class for Squares. As you might know, a square is just a special type of rectangle where the width is equal to the height.

class Square extends Rectangle {

public Square() {}

public Square(int size) {

width = height = size;

}

@Override

public void setWidth(int width) {

super.setWidth(width);

super.setHeight(width);

}

@Override

public void setHeight(int height) {

super.setHeight(height);

super.setWidth(height);

}

}

Our Square class extends the Rectangle class. We set height and width to the same value in the constructor, but we do not want any client (someone who uses our class in their code) to change height or weight in a way that can violate the square property.

Therefore we override the setters to set both properties whenever one of them is changed. But by doing that we have just violated the Liskov substitution principle.

Let's create a main class to perform tests on the **getArea**function.

class Test {

static void getAreaTest(Rectangle r) {

int width = r.getWidth();

r.setHeight(10);

System.out.println("Expected area of " + (width \* 10) + ", got " + r.getArea());

}

public static void main(String[] args) {

Rectangle rc = new Rectangle(2, 3);

getAreaTest(rc);

Rectangle sq = new Square();

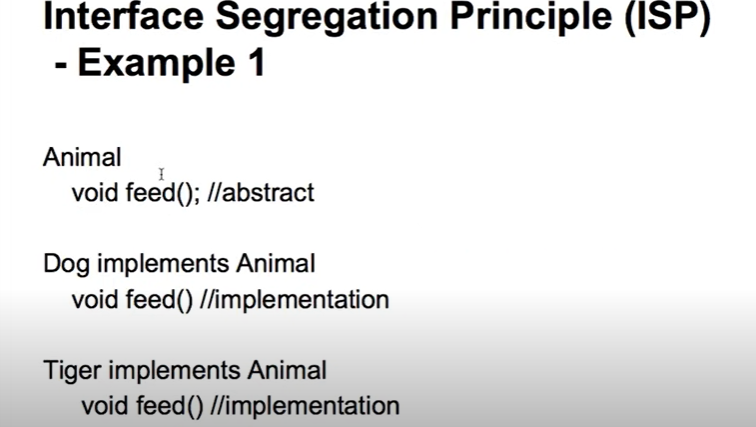
sq.setWidth(5);

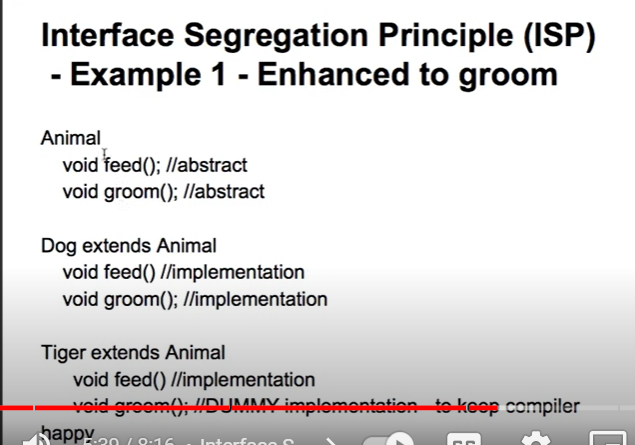
getAreaTest(sq);

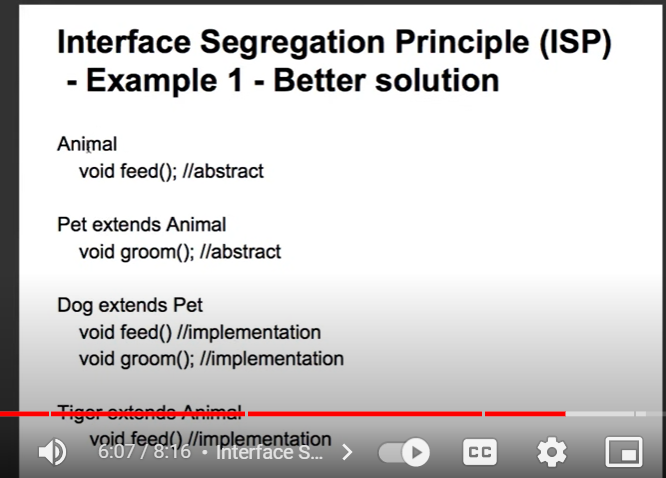
}

}

1. **Interface Segregation Principle:**This principle is the first principle that applies to Interfaces instead of classes in SOLID and it is similar to the single responsibility principle. It states that “do not force any client to implement an interface which is irrelevant to them“. Here your main goal is to focus on avoiding fat interface and give preference to many small client-specific interfaces. You should prefer many client interfaces rather than one general interface and each interface should have a specific responsibility.

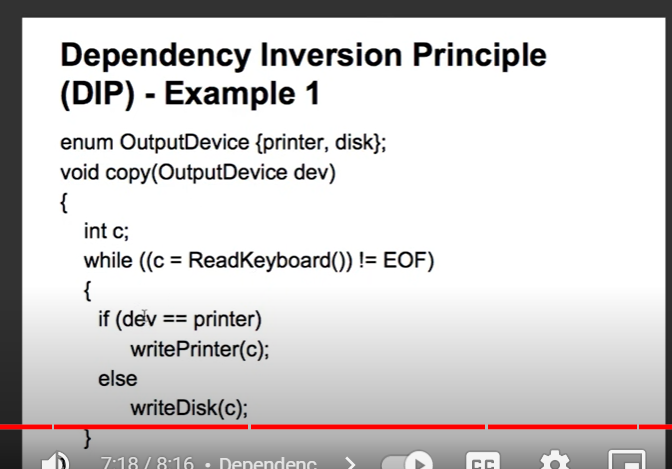






1. **Dependency Injection principle**

The Dependency Inversion principle states that our classes should depend upon interfaces or abstract classes instead of concrete classes and functions.



If the number of output devices change, we need to change this method.

Better create an interface Reader and Writer, and this method will read from any reader and write to any reader. It does not need to change.

