SOLID:

- **S**ingle responsibility principle
- **O**pen/closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

Single responsibility principle:

- A class should have only 1 reason to change.
- A class should have a single responsibility, where a responsibility is nothing but a reason to change.
- Every class should have a single responsibility.
- Responsibility should be entirely encapsulated by the class.
- All class services should be aligned with that responsibility.
- This makes the class more robust and reusable.
- E.g. Consider the below piece of code:

```
public class Employee{
  private String employeeId;
  private String name;
  private string address;
  private Date dateOfJoining;
  public boolean isPromotionDueThisYear(){
    //promotion logic implementation
  }
  public Double calcIncomeTaxForCurrentYear(){
    //income tax logic implementation
  }
  //Getters & Setters for all the private attributes
}
```

Some problems with this are:

- The promotion logic is the responsibility of HR.
- When promotion policies change, the Employee class does not need to change.
- Similarly, tax computation is the finance department's responsibility.
- If the Employee class owns the income tax calculation responsibility then whenever the tax structure/calculations change then Employee class will need to be changed.
- Lastly, the Employee class should have the single responsibility of maintaining core attributes of an employee.

To fix it:

1. Let's move the promotion determination logic from the Employee class to the HRPromotions class, as shown below:

```
public class HRPromotions{
  public boolean
       isPromotionDueThisYear(Employee emp){
    // promotion logic implementation
    // using the employee information passed
  }
}
```

2. Similarly, let's move the income tax calculation logic from Employee class to FinITCalculations class, as shown below:

```
public class FinITCalculations{
  public Double
        calcIncomeTaxForCurrentYear(Employee emp){
    //income tax logic implementation
    //using the employee information passed
  }
}
```

Our Employee class now remains with a single responsibility of maintaining core employee attributes, as shown below:

```
public class Employee{
  private String employeeId;
  private String name;
  private string address;
  private Date dateOfJoining;
  //Getters & Setters for all the private attributes
}
```

Open/closed principle:

- A class should be open for extensibility but closed for modification.
- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.
- We should add new features not by modifying the original class, but rather by extending it and adding new behaviours.
- The derived class may or may not have the same interface as the original class.
- A module will be said to be **open** if it is still available for extension. For example, it should be possible to add new fields or new methods.
  I.e. If attributes or behavior can be added to a class it can be said to be **open**.
- A module will be said to be **closed** if it is available for use by other modules. This assumes that the module has been given a well-defined, stable description.
  I.e. If a class is re-usable or specifically available for extending as a base class then it is **closed**.
- A class adheres to the Open/Closed Principle when it is closed, since it may be compiled, stored in a library, baselined, and used by client classes, but it is also open, since any new class may use it as a parent, adding new features.
  I.e. A class can be open and closed at the same time.
- The general idea of this principle is to tell you to write your code so that you will be able to add new functionality without changing the existing code. That prevents situations in which a change to one of your classes also requires you to adapt all depending classes.

- E.g.

Let's say we need to calculate areas of various shapes. Say our first shape is Rectangle.

```
public class Rectangle{
 public double length;
 public double width;
}
```

Next we create a class to calculate the area of this Rectangle which has a method calculateRectangleArea().

```
public class AreaCalculator{
  public double calculateRectangleArea(Rectangle
    rectangle){
    return rectangle.length *rectangle.width;
  }
}
```

Let's create a new class Circle with a single attribute radius.

```
public class Circle{
 public double radius;
}
```

Then we modify the AreaCalculator class.

```
public class AreaCalculator{
  public double calculateRectangleArea(Rectangle
      rectangle){
    return rectangle.length *rectangle.width;
  }
  public double calculateCircleArea(Circle circle){
    return Math.PI*circle.radius*circle.radius;
  }
}
```

As the types of shapes grow this becomes messier as AreaCalculator keeps on changing and any consumers of this class will have to keep on updating their libraries which contain AreaCalculator.

As a result, AreaCalculator class will not be baselined(finalized) with surety as every time a new shape comes it will be modified.
So, this design is not closed for modification.
Also, note that this design is not extensible.
As we add more shapes, AreaCalculator will need to keep on adding their computation logic in newer methods.
We are not really expanding the scope of shapes; rather we are simply doing piece-meal(bit-by-bit) solution for every shape that is added.

Instead, we should do this:
For this we need to first define a base type Shape.

```
public interface Shape{
  public double calculateArea();
}
```

Next, have Circle & Rectangle implement the Shape interface.

```
public class Rectangle implements Shape{
  double length;
  double width;
  public double calculateArea(){
    return length * width;
  }
}

public class Circle implements Shape{
  public double radius;
  public double calculateArea(){
    return Math.PI*radius*radius;
  }
}
```

Now, the AreaCalculator class looks like this.

```
public class AreaCalculator{
  public double calculateShapeArea(Shape shape){
    return shape.calculateArea();
  }
}
```

This AreaCalculator class now fully removes our design flaws noted above and gives a clean solution which adheres to the Open-Closed Principle.

Liskov substitution principle:

- If S is a subtype of T, then objects of type S may be substituted for objects of type T, without altering any of the desired properties of the program.
  Note: "S is a subtype of T" means S is a child class of T, or S implements interface T in Java.
  I.e. The Liskov substitution principle is saying "If C is a child class of P, then we should be able to substitute C for P in our code without breaking it."

- E.g. Consider a square and a rectangle. In math, a square is a rectangle. The "is a" makes you want to model this with inheritance. However if in code you made Square derive from Rectangle, then a Square should be usable anywhere you expect a Rectangle. This makes for some strange behavior. Imagine you had SetWidth and SetHeight methods on your Rectangle base class. If your Rectangle reference pointed to a Square, then SetWidth and SetHeight doesn't make sense because setting one would change the other to match it. In this case Square fails the Liskov Substitution Test with Rectangle and the abstraction of having Square inherit from Rectangle is a bad one. Hence, in OO programming and design, unlike in math, it is not the case that a Square is a Rectangle. This is because a Rectangle has more behaviours than a Square, not less.
- The LSP is related to the Open/Close principle. The sub classes should only extend (add behaviours), not modify or remove them.
- The LSP is applicable when there's a supertype-subtype inheritance relationship by either extending a class or implementing an interface. We can think of the methods defined in the supertype as defining a contract. Every subtype is expected to stick to this contract. If a subclass does not adhere to the superclass's contract, it's violating the LSP.
- This makes sense intuitively. A class's contract tells its clients what to expect. If a subclass extends or overrides the behavior of the superclass in unintended ways, it would break the clients.
- Some ways a method in a subclass can break a superclass method's contract are:
    1. Returning an object that's incompatible with the object returned by the superclass method.
    2. Throwing a new exception that's not thrown by the superclass method.
    3. Changing the semantics or introducing side effects that are not part of the superclass's contract.

Interface segregation principle:

- Clients should not be forced to depend on methods they do not use. Declaring methods in an interface that the client doesn't need pollutes the interface and leads to a "bulky" or "fat" interface.
- It is better to have lots of small, specific interfaces than fewer larger ones. This way, it will be easier to extend and modify the design.
- Similar to the Single Responsibility Principle, the goal of the Interface Segregation Principle is to reduce the side effects and frequency of required changes by splitting the software into multiple, independent parts.
- E.g. We'll create some code for a burger place where a customer can order a burger, fries or a combo of both.

```
interface OrderService {
    void orderBurger(int quantity);
    void orderFries(int fries);
    void orderCombo(int quantity, int fries);
}
```

Since a customer can order fries, or a burger, or both, we decided to put all order methods in a single interface.
Now, to implement a burger-only order, we are forced to throw an exception in the orderFries() method and the orderCombo() method.

```
class BurgerOrderService implements OrderService {
    @Override
    public void orderBurger(int quantity) {
        System.out.println("Received order of "+quantity+" burgers");
    }

    @Override
    public void orderFries(int fries) {
        throw new UnsupportedOperationException("No fries in burger only order"
    }

    @Override
    public void orderCombo(int quantity, int fries) {
        throw new UnsupportedOperationException("No combo in burger only order"
    }
}
```

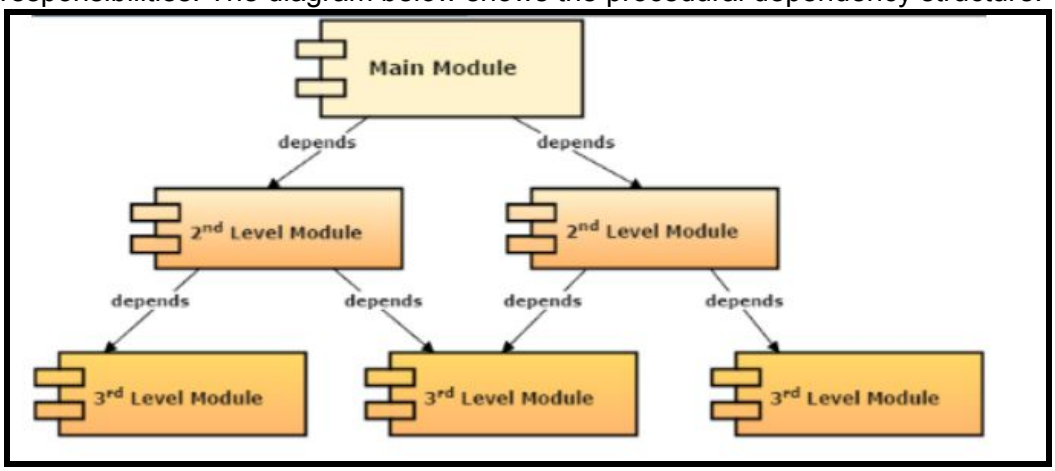
Similarly, for a fries-only order, we'd also need to throw an exception in orderBurger() method and the orderCombo() method.
And this is not the only downside of this design. The BurgerOrderService and FriesOrderService classes will also have unwanted side effects whenever we make changes to our abstraction. Let's say we decided to accept an order of fries in units such as pounds or grams. In that case, we most likely have to add a unit parameter in orderFries(). This change will also affect BurgerOrderService even though it's not implementing this method.

- By violating the ISP, we face the following problems in our code:
    - Client developers are confused by the methods they don't need.
    - Maintenance becomes harder because of side effects. A change in an interface forces us to change classes that don't implement the interface.

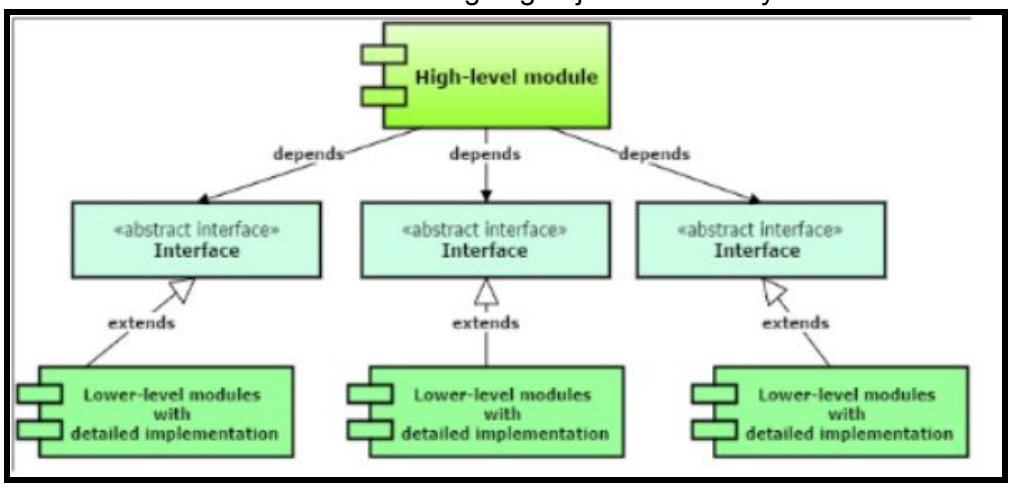
Dependency inversion principle:

- The dependency inversion principle states the following:
    1. Depend upon abstractions. Do not depend upon concretions.
    2. Abstractions should not depend upon details. Details should depend upon abstractions.
    3. High-level modules should not depend on low-level modules. Both should depend on abstractions.
- The general idea of this principle is that high-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features. To achieve that, you need to introduce an abstraction that decouples the high-level and low-level modules from each other.

- In procedural systems, higher level modules depend on lower level modules to fulfil their responsibilities. The diagram below shows the procedural dependency structure.



The Dependency Inversion Principle, however, advocates that the dependency structure should rather be inverted when designing object-oriented systems.



- If you take a relook at the diagram above showing modular dependencies in a procedural system, then one can clearly see the tight coupling that each module layer has with its sub-layer. Thus, any change in the sub-layer will have a ripple effect in the next higher layer and may propagate even further upwards. This tight coupling makes it extremely difficult and costly to maintain and extend the functionality of the layers.
- The Dependency Inversion Principle, does away with this tight-coupling between layers by introducing a layer of abstraction between them. So, the higher-level layers, rather than depending directly on the lower-level layers, instead depend on a common abstraction. The lower-level layer can then vary(be modified or extended) without the fear of disturbing higher-level layers depending on it, as long as it obeys the contract of the abstract interface. If, as shown in the object-oriented design diagram above, the lower layers literally extend the abstraction layer interfaces, then they will follow the contract.