SOLID Principles

The **SOLID** principles are a set of five guidelines that help developers design software that is easier to understand, maintain, and extend. The term was introduced by Robert C. Martin (Uncle Bob) in the early 2000s and became a **foundation for object-oriented design**.

They are not strict rules, but rather guidelines that encourage good software architecture.

SRP – Single Responsibility Principle

The Single Responsibility Principle (SRP) is the first of the SOLID principles and also the simplest to understand — yet it's one of the most commonly violated. The idea is that every class, function, or module should focus on **only one responsibility**.

Definition:
A class should have only one reason to change.
In other words, a class should do only one thing, and do it well.

Why it matters:

- **Simpler testing**: smaller units are easier to test.
- **Higher readability**: you instantly understand what the class does.
- **Safer changes**: modifying one responsibility won't break unrelated functionality.
- **Better teamwork**: multiple developers can work on different parts without conflicts.

Bad Example (violating SRP):
This class is doing too many things at once: saving to the database and sending an email.

```
class UserManager {
  saveToDB(user) {
    console.log("Saving user to database...");
  }

  sendEmail(user) {
    console.log("Sending email to user...");
  }
}
```

Problem: If tomorrow the email system changes (e.g., from SMTP to SendGrid), you'd have to modify this same class, even though database logic has nothing to do with sending emails.

Good Example (following SRP):
Separate responsibilities into two classes:

```
class UserRepository {
  saveToDB(user) {
    console.log("Saving user to database...");
  }
}

class UserNotifier {
  sendEmail(user) {
    console.log("Sending email to user...");
  }
}
```

Now each class has only one responsibility and one reason to change.

Illustration:

```
❌ One Person Does Everything
  +-----------------------+
  |   Employee            |
  |  - cookFood()         |
  |  - serveTable()       |
  |  - processPayment()   |
  +-----------------------+

✅ Clear Roles
  +---------+   +--------+   +------------+
  |  Chef   |   | Waiter |   |  Cashier   |
  | cook()  |   | serve()|   | pay()      |
  +---------+   +--------+   +------------+
```

OCP – Open/Closed Principle

The Open/Closed Principle (OCP) is the second principle in SOLID. It encourages us to design software components that are open for extension but closed for modification.

Definition:
Software entities (classes, modules, functions) should be open for extension, but closed for modification.

This means we should be able to add new features without changing the existing, working code, but easy to extend (by adding new classes or methods). This is usually achieved through abstraction and polymorphism.

Why it matters:

- Prevents breaking existing features when adding new ones.
- Makes your code more stable and predictable.
- Encourages the use of abstraction (interfaces, base classes, strategy patterns).
- Critical in growing projects where features are constantly added

Bad Example (violating OCP):
Here, we must modify the same class whenever a new shape is added:

```
class AreaCalculator {
  calculate(shape) {
    if (shape.type === "circle") {
      return Math.PI * shape.radius * shape.radius;
    } else if (shape.type === "square") {
      return shape.side * shape.side;
    }
    // Adding a new shape? We must edit this class again!
  }
}
```

Problem: Every time a new shape (`triangle`, `rectangle`, etc.) is introduced, we have to go back and modify this class — risking breaking existing logic.

Good Example (following OCP):
We use polymorphism: each shape knows how to calculate its own area. The calculator just calls the method.

```
class Circle {
  constructor(radius) {
    this.radius = radius;
  }
  area() {
    return Math.PI * this.radius * this.radius;
  }
}

class Square {
  constructor(side) {
    this.side = side;
  }
  area() {
    return this.side * this.side;
  }
}

class AreaCalculator {
  calculate(shape) {
    return shape.area();
  }
}

// Usage
const calculator = new AreaCalculator();
console.log(calculator.calculate(new Circle(5)));
console.log(calculator.calculate(new Square(4)));
```

Now, if we add a `Triangle class`, we don’t touch `AreaCalculator` — we just extend the system with a new class.

Illustration:

```
❌ Bad: Rewrite the menu each time       ✅ Good: Extend menu without modifying

+----------------------------+          +-----------------------------+
| Menu                       |          | AbstractDish (base recipe)  |
| - servePizza()             |          | - prepare()                 |
| - servePasta()             |          +-----------------------------+
| - serveBurger()            |                     ^
| // add new dish → modify   |                     |
+----------------------------+            +---------------+   +---------------+
                                          | PizzaDish     |   | PastaDish     |
                                          | prepare()     |   | prepare()     |
                                          +---------------+   +---------------+

                                        +-----------------------------+
                                        | Menu                        |
                                        | - serve(dish.prepare())     |
                                        +-----------------------------+
```

LSP – Liskov Substitution Principle

The Liskov Substitution Principle (LSP), formulated by Barbara Liskov, is the third SOLID principle. It states that subtypes must be substitutable for their base types without altering the correctness of the program.

Definition:
Objects of a superclass should be replaceable with objects of a subclass without affecting the program.

In other words, you should be able to replace a parent class with a child class without breaking the program.

Why it matters:

- Ensures **reliable polymorphism**.
- Makes **inheritance safer** and predictable.
- Prevents unexpected behavior when extending classes.
- Helps create more **robust, reusable code**.

Bad Example (violating LSP):

```
class Bird {
  fly() {
    console.log("Flying high!");
  }
}

class Penguin extends Bird {
  fly() {
    throw new Error("I can't fly!");
  }
}

function makeBirdFly(bird) {
  bird.fly(); // expects any bird to fly
}

makeBirdFly(new Bird());     // ✅ Flying high!
makeBirdFly(new Penguin());  // ❌ Error: breaks expectations
```

Problem: The `Penguin class` **breaks the contract** of the `Bird class`. Code expecting a “Bird” that flies now crashes.

Good Example (following LSP):

We separate birds into `FlyingBird` and `NonFlyingBird`, or use a `canFly` method to respect behavior:

```
class Bird {
  move() {
    console.log("Moving...");
  }
}

class FlyingBird extends Bird {
  fly() {
    console.log("Flying high!");
  }
}

class Penguin extends Bird {
  swim() {
    console.log("Swimming in water!");
  }
}

// Usage
const birds = [new FlyingBird(), new Penguin()];
birds.forEach(bird => bird.move()); // works for all birds
```

Illustration:

```
❌ Bad Example: Violates LSP
+------------------+
|      Bird        |
|------------------|
| fly()            |
+------------------+
         ^
         |
+------------------+
|    Penguin       |
| fly() throws err |
+------------------+
  makeBirdFly(new Penguin());  // ❌ breaks code

✅ Good Example: Follows LSP
  +------------------+
  |      Bird        |
  |------------------|
  | move()           |
  +------------------+
           ^
           |
  +------------------+     +-------------------+
  | FlyingBird       |     | Penguin           |
  | move() = fly()   |     | move() = swim()   |
  +------------------+     +-------------------+

birds.forEach(bird => bird.move()); // ✅ works for all birds
```

ISP – Interface Segregation Principle

The Interface Segregation Principle (ISP) is the fourth SOLID principle. It emphasizes **that no client should be forced to depend on methods it does not use**.

Definition:
Many specific interfaces are better than one general-purpose interface.

In other words, instead of creating one huge interface with many methods, break it into smaller, more focused interfaces so classes only implement what they actually need.

Why it matters:

- Avoids forcing classes to implement **unused methods**.
- Reduces **unnecessary dependencies**.
- Makes code **cleaner, easier to maintain, and more flexible**.
- Improves **readability** — each class only sees what it actually uses.

Bad Example (violating ISP):
A general-purpose `Worker` interface forces all workers to implement methods they may not need.

```
  class Worker {
    work() {}
    eat() {}
```

```
}

class Robot extends Worker {
  work() {
    console.log("Robot is working...");
  }

  eat() {
    throw new Error("Robots do not eat!");
  }
}
```

Problem: `Robot` doesn't need `eat()`, but is forced to implement it, violating ISP.

Good Example (following ISP):

Split the interface into smaller, more focused interfaces.

```
class Workable {
  work() {}
}

class Eatable {
  eat() {}
}

class HumanWorker extends Workable {
  work() {
    console.log("Human is working...");
  }
  eat() {
    console.log("Human is eating...");
  }
}

class RobotWorker extends Workable {
  work() {
    console.log("Robot is working...");
  }
}

// Usage
const workers = [new HumanWorker(), new RobotWorker()];
workers.forEach(worker => worker.work());
```

Illustration:

```
❌ Bad: One general interface forces unrelated responsibilities
 +-----------------------+
 |       Worker          |
 |-----------------------|
 | work()                |
 | eat()                 |  <-- Robot doesn't need this
 +-----------------------+
          ^
          |
 +-----------------------+
 | Robot                 |
 | work()                |  <-- works fine
 | eat() throws error    |  <-- unnecessary! violates ISP
 +-----------------------+

 +-----------------------+
 | Human                 |
 | work()                |
 | eat()                 |  <-- fine
 +-----------------------+

✅ Good: Split interfaces so classes only implement what they need
 +-------------+       +-------------+
 | Workable    |       | Eatable     |
 | work()      |       | eat()       |
 +-------------+       +-------------+
      ^                     ^
      |                     |
 +-------------+       +-------------+
 | Robot       |       | Human       |
 | work()      |       | eat()       |
 +-------------+       +-------------+
```

DIP – Dependency Inversion Principle

The Dependency Inversion Principle (DIP) is the fifth SOLID principle. It emphasizes that **high-level modules should not depend on low-level modules directly; both should depend on abstractions.**

Definition:
Depend on abstractions, not on concrete implementations..

This means your code should depend on interfaces or abstract classes, not specific classes, making it easier to change or extend functionality without modifying high-level code.

Why it matters:

- Reduces **tight coupling** between modules.
- Makes code **flexible, maintainable, and testable**.
- Allows swapping implementations **without changing high-level logic**.

Bad Example (violating DIP):
A Car class directly depends on GasEngine, so changing to ElectricEngine requires modifying Car.

```
class GasEngine {
  start() {
    console.log("Gas engine started");
  }
}

class Car {
  constructor() {
     this.engine = new GasEngine(); // directly depends on concrete class
  }
  start() {
    this.engine.start();
  }
}

// Usage
const myCar = new Car();
myCar.start();
```

Problem: high-level module `Car` depends on a low-level module `GasEngine`, switching to `ElectricEngine` means editing `Car`, violating DIP.

Good Example (following DIP):
Introduce an Engine interface that Car depends on instead of a concrete engine:.

```
class Engine {
  start() {} // abstraction
}

class GasEngine extends Engine {
  start() {
    console.log("Gas engine started");
  }
}

class ElectricEngine extends Engine {
  start() {
    console.log("Electric engine started");
  }
}

class Car {
  constructor(engine) {
    this.engine = engine; // depends on abstraction
  }
  start() {
    this.engine.start();
  }
}

// Usage
const gasCar = new Car(new GasEngine());
const electricCar = new Car(new ElectricEngine());

gasCar.start();        // Gas engine started
electricCar.start();   // Electric engine started
```

Illustration:

```
✕ Bad: High-level depends on low-level
+---------+      +-----------+
|   Car   | ---> | GasEngine |
| engine  |      | start()   |
+---------+      +-----------+

✅ Good: High-level depends on abstraction
+---------+
|   Car   |
| engine: |----------------+
+---------+                |
                           v
                 +-----------+   +---------------+
                 | GasEngine |   | ElectricEngine|
                 | start()   |   | start()       |
                 +-----------+   +---------------+
```