HOW TO BUILD YOUR CODE USING SOLID PRINCIPLES

Single Responsibility Principle

Each class should have one responsibility

BAD ▸ Mixing responsibilities in a class

```
public class User {
    void saveUser(User user) {
        // Saving user data to the database
    }
    void processPayment(double amount) {
        // Process payment logic
    }
}
```

GOOD ▸ Separating responsibilities

```
class UserService {
    void saveUser(User user) {
        // Saving user data to the database
    }
}

class PaymentProcessor {
    void processPayment(double amount) {
        // Process payment logic
    }
}
```

Open/Closed Principle

Software entities should be open for extension but closed for modification

BAD ▸ Modifying existing class for new methods

```
class PaymentProcessor {
    void processPayment(double amount, String method) {
        if (method.equals("CreditCard")) {
            // Credit card payment logic
        } else if (method.equals("PayPal")) {
            // PayPal payment logic
        }
        // More payment methods...
    }
}
```

GOOD ▸ Extending without modification

```
interface PaymentMethod {
    void processPayment(double amount);
}

class CreditCardPayment implements PaymentMethod {
    void processPayment(double amount) {
        // Credit card payment logic
    }
}

class PayPalPayment implements PaymentMethod {
    void processPayment(double amount) {
        // PayPal payment logic
    }
}
```

Liskov Substitution Principle

Subtypes must be substitutable for their base types

BAD ▸ Modifying existing class for new methods

```
class Rectangle {
    int width;
    int height;
    void setWidth(int width) { /* ... */ }
    void setHeight(int height) { /* ... */ }
    int area() { /* ... */ }
}

class Square extends Rectangle { /* ... */ }
```

GOOD ▸ Extending without modification

```
abstract class Shape {
    abstract int area();
}

class Rectangle extends Shape { /* ... */ }

class Square extends Shape { /* ... */ }
```

Interface Segregation Principle

A class should not be forced to implement interfaces it doesn't use. keep interfaces focused on specific tasks.

BAD ▸ Expanded interface with unnecessary methods

```
interface Machine {
    void print();
    void scan();
    void fax();
}

class AllInOnePrinter implements Machine { /* ... */ }
```

GOOD ▸ Separated interfaces for specific responsibilities

```
interface Printer {
    void print();
}

interface Scanner {
    void scan();
}

interface Faxer {
    void fax();
}

class AllInOnePrinter implements Printer, Scanner { /* ... */ }
```

Dependency Inversion Principle

Decoupling high-level modules from low-level modules by introducing an abstraction layer

BAD ▸ Direct Dependency

```
class LightBulb {
    void turnOn() {
        // Code to turn on the light bulb
    }
}

class Switch {
    private LightBulb bulb = new LightBulb();

    void press() {
        bulb.turnOn();
    }
}
```

GOOD ▸ Both depend on abstractions

```
interface Switchable {
    void turnOn();
}

class LightBulb implements Switchable {
    void turnOn() { /** Code to turn on the light bulb **/ }
}

class Switch {
    private Switchable device;

    Switch(Switchable device) {
        this.device = device;
    }
}
```