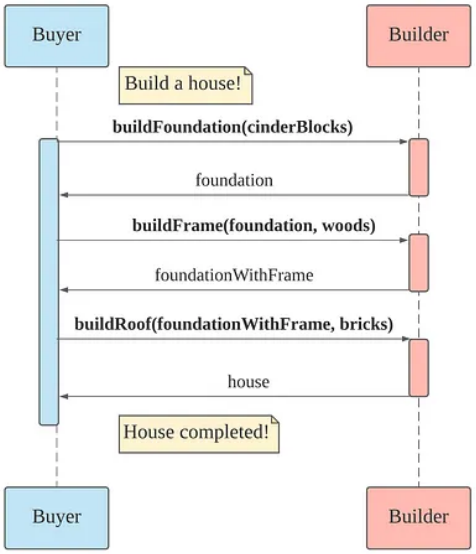
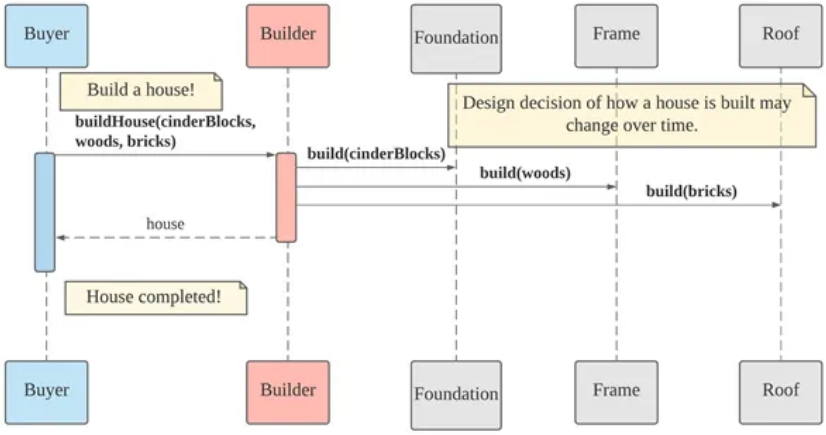
WRITING CLEAN CODE

https://github.com/PacktPublishing/Writing-Clean-Code---20-Common-Code-Smells-and-How-to-Avoid-Them

SOFTWARE DESIGN PRINCIPLE

Information hiding: Hides the internal details of a module or component that may change, so that other components that depend on it are not affected.



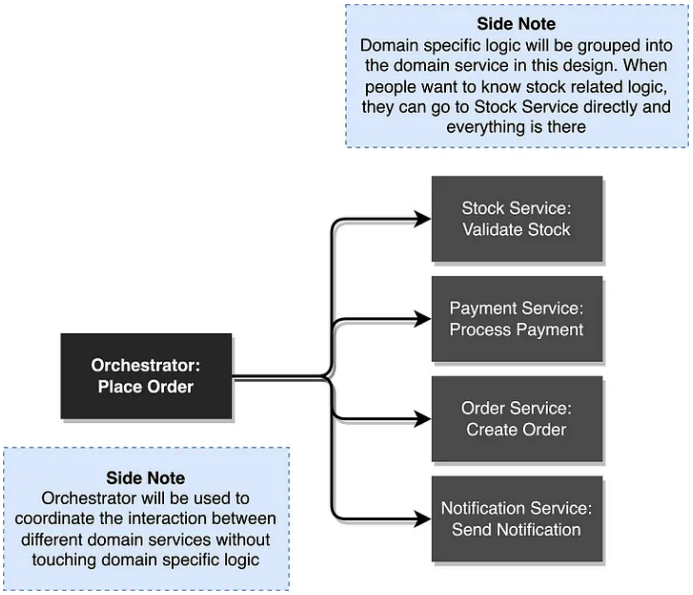


Before

After

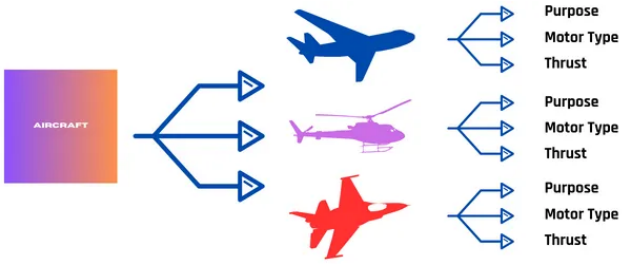
```tsx
// ProductService.ts
export class ProductService {
  async getAllProducts(): Promise<Product[]> {
    // Today we use db
    return await db.products.findMany();

    // In the future we can want to change it to:
    // return await axios.get("https://external.api/products");
  }
}


// productsController.ts
import { ProductService } from "./ProductService";

const service = new ProductService();

app.get("/products", async (req, res) => {
  const products = await service.getAllProducts();
  res.json(products);
});
```

Encapsulation: controlling the access to information by grouping or wrapping related items together to hide the complexity. For example, creating a Car component that wraps up all mechanical parts of a car in a single unit.




```
public void placeOrder(...) {
    validateStock(...);
    processPayment(...);
    createOrder(...);
    sendNotification(...);
}

prviate void validateStock(...) {
    // doSomething
}

prviate void processPayment(...) {
    // doSomething
}

prviate void createOrder(...) {
    // doSomething
}

prviate void sendNotification(...) {
    // doSomething
}
```

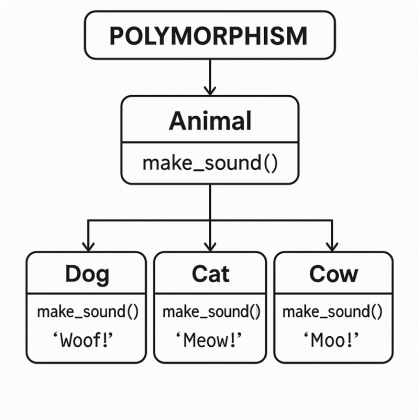
Abstraction: hiding unwanted details so that a programmer can focus on information of greater importance.



Counter.tsx

```
// Abstract class: defines the interface
abstract class PaymentProcessor {
  abstract pay(amount: number): void;
}

// Concrete class: implements the abstract behavior
class StripeProcessor extends PaymentProcessor {
  pay(amount: number): void {
    console.log(`Paying ${amount} with Stripe`);
    // implementation details hidden from the user
  }
}

class PayPalProcessor extends PaymentProcessor {
  pay(amount: number): void {
    console.log(`Paying ${amount} with PayPal`);
    // implementation details hidden from the user
  }
}

// Higher-level code using abstraction
function checkout(processor: PaymentProcessor, amount: number) {
  processor.pay(amount); // only cares about "pay", not how it's done
}

// Usage
const processor = new StripeProcessor();
checkout(processor, 100);  // Output: Paying $100 with Stripe

//What's abstracted?
// The details of how Stripe or PayPal processes payments.
// checkout() only cares that the processor can pay().
```

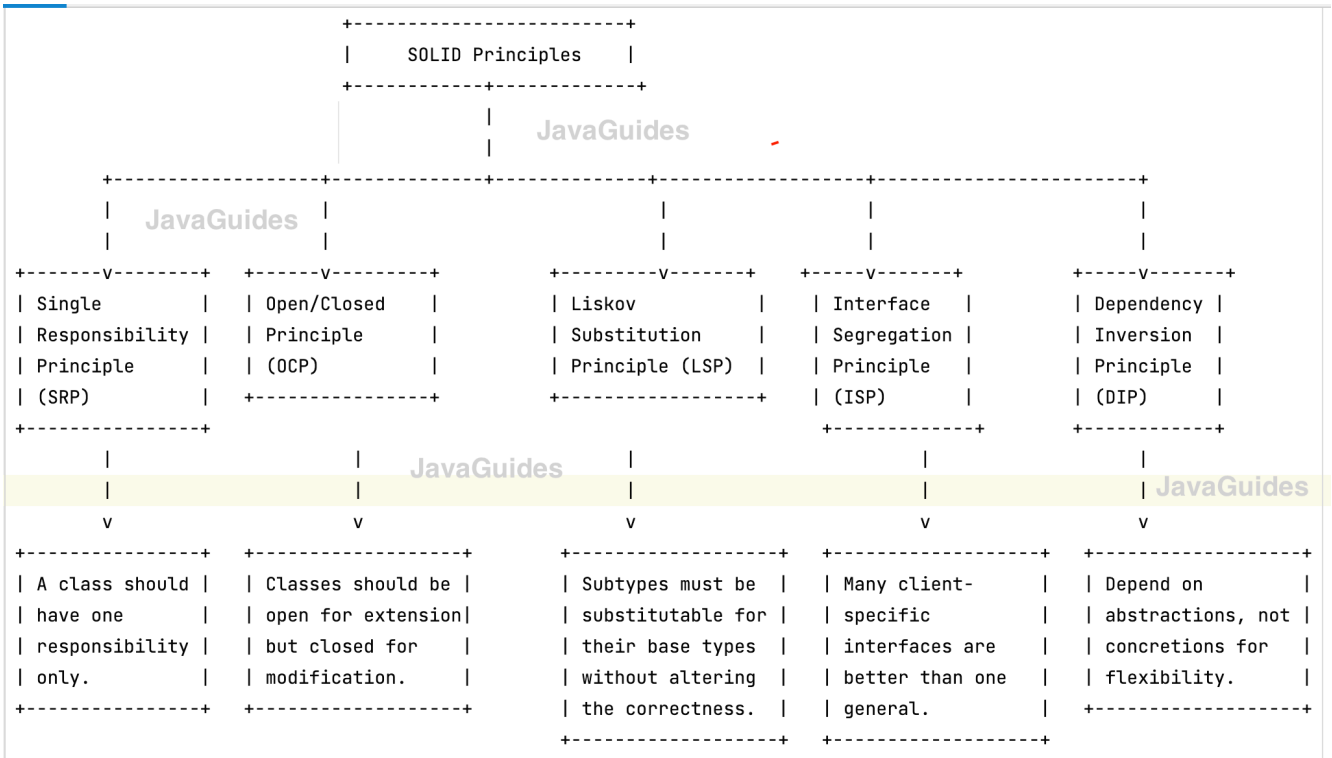
Polymorphism: concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It means “many forms”.

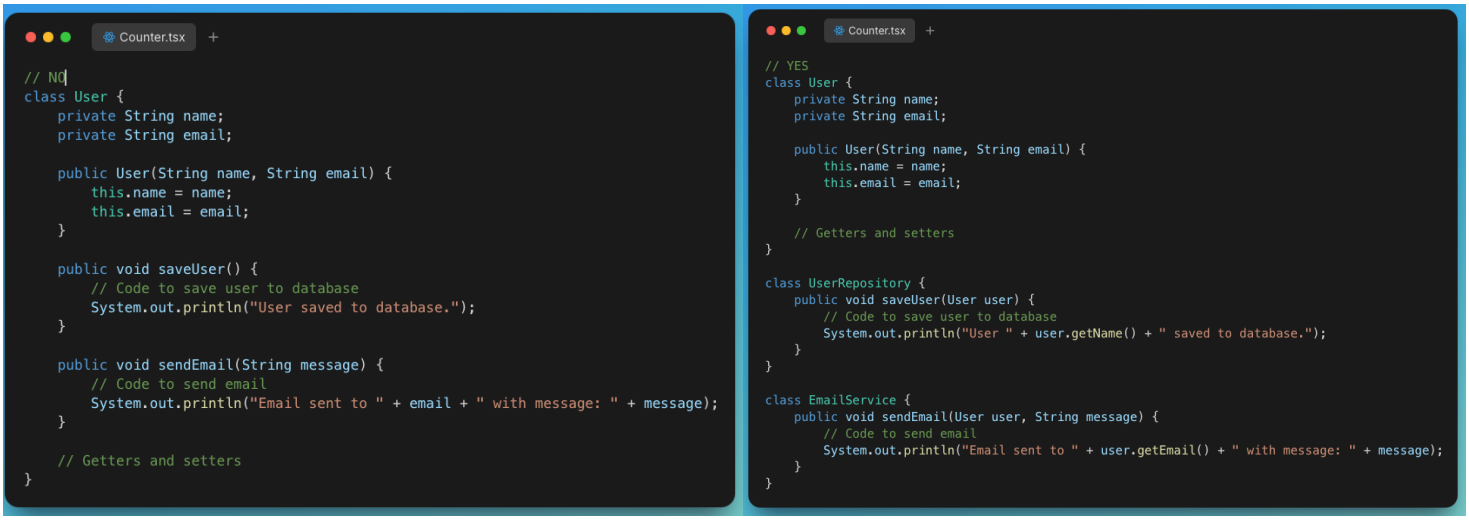


Counter.tsx

```python
class Animal:
    def make_sound(self):
        print("Some generic animal sound")

class Dog(Animal):
    def make_sound(self):
        print("Woof!")

class Cat(Animal):
    def make_sound(self):
        print("Meow!")

# Polymorphism in action
animals = [Dog(), Cat()]

for animal in animals:
    animal.make_sound()  # Each object behaves according to its class
```

SOLID PRINCIPLES

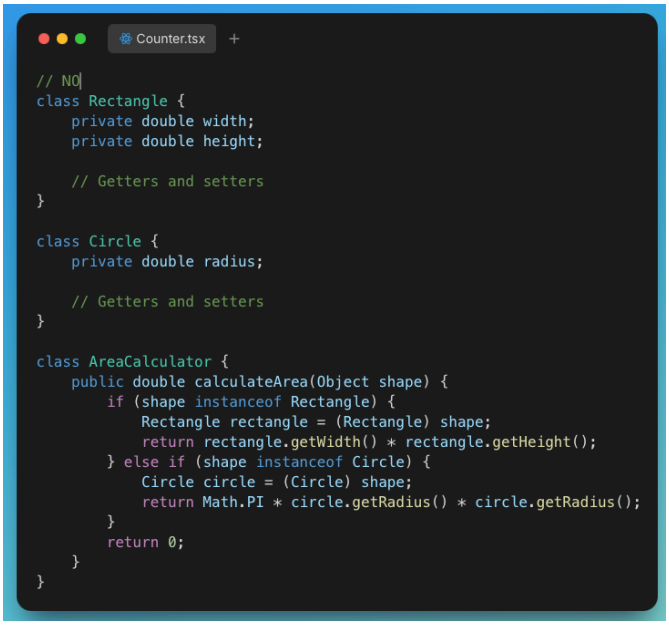




```tsx
// NO
class User {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public void saveUser() {
        // Code to save user to database
        System.out.println("User saved to database.");
    }

    public void sendEmail(String message) {
        // Code to send email
        System.out.println("Email sent to " + email + " with message: " + message);
    }

    // Getters and setters
}
```

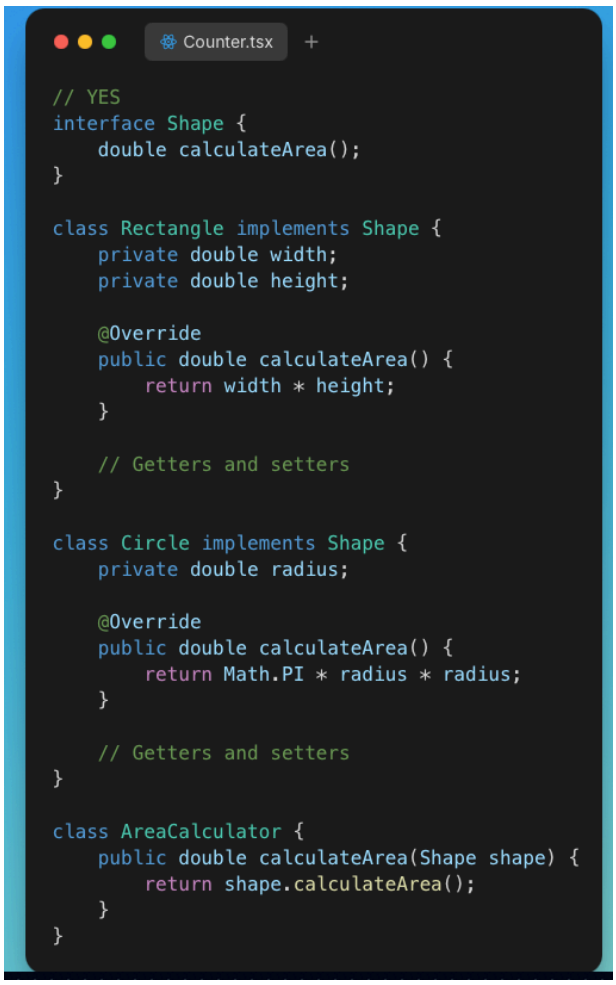
```tsx
// YES
class User {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // Getters and setters
}

class UserRepository {
    public void saveUser(User user) {
        // Code to save user to database
        System.out.println("User " + user.getName() + " saved to database.");
    }
}

class EmailService {
    public void sendEmail(User user, String message) {
        // Code to send email
        System.out.println("Email sent to " + user.getEmail() + " with message: " + message);
    }
}
```

- Single responsability



```tsx
// NO
class Rectangle {
    private double width;
    private double height;

    // Getters and setters
}

class Circle {
    private double radius;

    // Getters and setters
}

class AreaCalculator {
    public double calculateArea(Object shape) {
        if (shape instanceof Rectangle) {
            Rectangle rectangle = (Rectangle) shape;
            return rectangle.getWidth() * rectangle.getHeight();
        } else if (shape instanceof Circle) {
            Circle circle = (Circle) shape;
            return Math.PI * circle.getRadius() * circle.getRadius();
        }
        return 0;
    }
}
```

- Open / Closed



```tsx
// YES
interface Shape {
    double calculateArea();
}

class Rectangle implements Shape {
    private double width;
    private double height;

    @Override
    public double calculateArea() {
        return width * height;
    }

    // Getters and setters
}

class Circle implements Shape {
    private double radius;

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }

    // Getters and setters
}

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

Counter.tsx +

```java
// NO
class Bird {
    public void fly() {
        System.out.println("Bird is flying");
    }
}

class Sparrow extends Bird {
    @Override
    public void fly() {
        System.out.println("Sparrow is flying");
    }
}

class Penguin extends Bird {
    @Override
    public void fly() {
        throw new UnsupportedOperationException("Penguins can't fly");
    }
}
```

-Liskov substitution

Counter.tsx +

```java
// YES
abstract class Bird {
    public abstract void move();
}

class Sparrow extends Bird {
    @Override
    public void move() {
        System.out.println("Sparrow is flying");
    }
}

class Penguin extends Bird {
    @Override
    public void move() {
        System.out.println("Penguin is swimming");
    }
}

public class Main {
    public static void main(String[] args) {
        Bird sparrow = new Sparrow();
        Bird penguin = new Penguin();

        sparrow.move(); // Output: Sparrow is flying
        penguin.move(); // Output: Penguin is swimming
    }
}
```

- Interface segregations

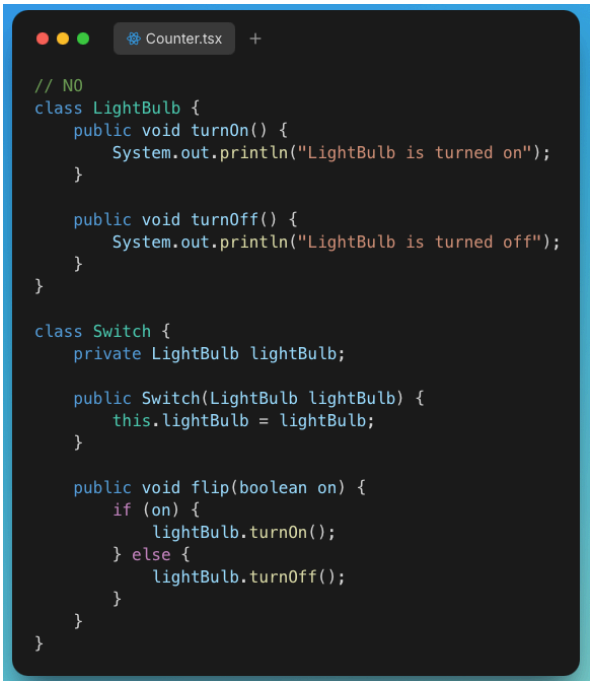
Counter.tsx +

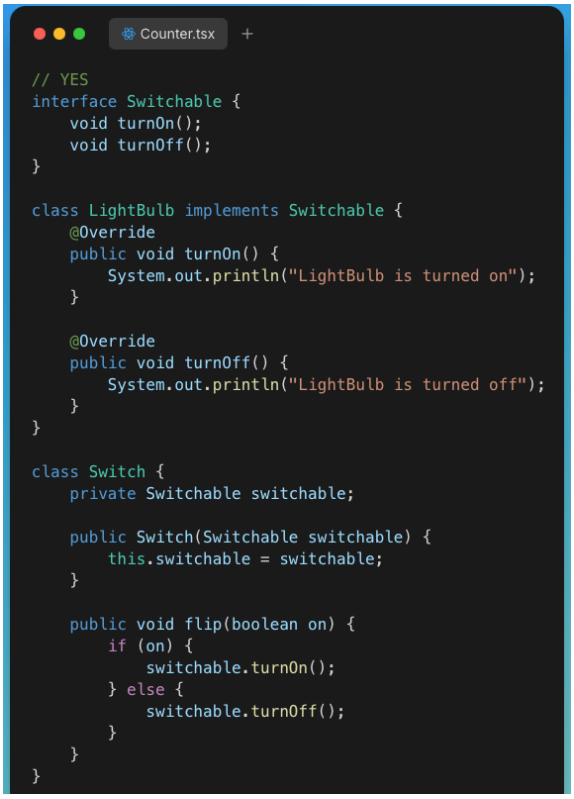
```java
// NO
interface Worker {
    void work();
    void eat();
}

class Developer implements Worker {
    @Override
    public void work() {
        System.out.println("Developer is coding.");
    }

    @Override
    public void eat() {
        System.out.println("Developer is eating.");
    }
}

class Robot implements Worker {
    @Override
    public void work() {
        System.out.println("Robot is working.");
    }

    @Override
    public void eat() {
        throw new UnsupportedOperationException("Robot does not eat.");
    }
}
```

Counter.tsx +

```java
// YES
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

class Developer implements Workable, Eatable {
    @Override
    public void work() {
        System.out.println("Developer is coding.");
    }

    @Override
    public void eat() {
        System.out.println("Developer is eating.");
    }
}

class Robot implements Workable {
    @Override
    public void work() {
        System.out.println("Robot is working.");
    }
}
```

- Dependency inversion





REPEATED SWITCHES

We have the same switch statements happiness in different parts of the code

How to fix it?
- Encapsulate the conditional logic
- Use polymorphism

PRIMITIVE OBSESSION

Avoid creating their own types, rely too much on primitives. Use custom types

Inefficient Loops

To many loops make it confusing to read the code and hard to maintain. Use pipeline structures (.map, .filer, .reduce, …)

LONG PARAMETER LIST

Long list of parameters, example flag parameters (for flags: use polymorphism to divide the operations)

- substitute data parameters by a single data object parameter
- Only pass necessary parameters

KNOWLEDGE DUPLICATION

The same codified knowledge is found in multiple places
Extract the duplicate knowledge into helper functions or classes

———-

Refused Bequest
ocurre cuando una subclase hereda métodos o propiedades de su superclase que no necesita o no usa, lo que rompe el principio de sustitución de Liskov (LSP).

Usar interfaces en lugar de superclases permite definir solo el comportamiento necesario, sin forzar herencia innecesaria. Esto:

- Reduce el acoplamiento (loose coupling)
- Mejora la reutilización
- Favorece composición sobre herencia