## **🔁 1. Red-Green-Refactor (TDD Workflow)**

This is a **Test-Driven Development** cycle:

✅ **Red**: Write a failing test →  
 ✅ **Green**: Write the code to pass the test →  
 ✅ **Refactor**: Improve the code without changing behavior.

### **🧠 Why it matters:**

It ensures **working, testable code** before optimization. You don’t write unnecessary code.

### **✅ Example (React + Jest):**

tsx

CopyEdit

// Red: Write test first

test('should calculate discounted price', () => {

expect(calculateDiscount(100, 20)).toBe(80); // 20% discount

});

// Green: Minimal code to pass

function calculateDiscount(price: number, discount: number) {

return price - (price \* discount / 100);

}

// Refactor: Add validation, types, naming

function calculateDiscount(price: number, discountPercent: number): number {

if (price < 0 || discountPercent < 0) throw new Error('Invalid input');

return price - (price \* discountPercent / 100);

}

## **🧼 2. DRY (Don’t Repeat Yourself)**

**Avoid repeating logic, UI, or data** in your codebase.

### **🧠 Why it matters:**

Copy-paste is dangerous. Any logic duplicated will eventually go out of sync or be forgotten.

### **🚫 Bad (repeated buttons):**

tsx

CopyEdit

<button onClick={() => addToCart(1)}>Add</button>

<button onClick={() => addToCart(2)}>Add</button>

### **✅ Good (Reusable component):**

tsx

CopyEdit

const AddToCartButton = ({ productId }: { productId: number }) => (

<button onClick={() => addToCart(productId)}>Add</button>

);

// Usage:

<AddToCartButton productId={1} />

<AddToCartButton productId={2} />

### **✅ Backend DRY:**

ts

CopyEdit

// Instead of repeating:

if (!email) throw new Error("Missing email");

if (!password) throw new Error("Missing password");

// Use:

validateFields({ email, password });

## **💡 3. KISS (Keep It Simple, Stupid)**

Code should be as **simple and obvious** as possible. Avoid overengineering.

### **🧠 Why it matters:**

Junior devs often overcomplicate. Senior devs solve problems with **minimal logic** and **maximum clarity**.

### **🚫 Bad (Too complex):**

tsx

CopyEdit

const Button = ({ children, isPrimary }) => {

return isPrimary ? (

<button style={{ backgroundColor: 'blue' }}>{children}</button>

) : (

<button style={{ backgroundColor: 'gray' }}>{children}</button>

);

};

### **✅ Good:**

tsx

CopyEdit

const Button = ({ children, variant = 'primary' }) => {

const color = variant === 'primary' ? 'blue' : 'gray';

return <button style={{ backgroundColor: color }}>{children}</button>;

};

## **🙅‍♂️ 4. YAGNI (You Aren’t Gonna Need It)**

Don’t build features or abstractions until you actually need them.

### **🧠 Why it matters:**

Many juniors add flexibility **too early** — future-proofing code that no one ever uses.

### **🚫 Bad (Premature abstraction):**

ts

CopyEdit

// Trying to be too flexible too early

class ProductManager {

constructor(productType: string) { ... }

}

### **✅ Good:**

ts

CopyEdit

// Just build what you need first

function fetchProducts() {

return fetch('/api/products');

}

Later, **when needed**, extract to a class or service.

## **🧱 5. SOLID Principles (OOP-inspired but applicable)**

These help make code **scalable, maintainable, and testable** — especially in services or business logic.

### **✅ S - Single Responsibility Principle**

One function/class should do one thing only.

### **🚫 Bad:**

ts

CopyEdit

function handleOrder(req, res) {

// Validate input

// Save order to DB

// Send email

// Respond to client

}

### **✅ Good:**

ts

CopyEdit

function handleOrder(req, res) {

const orderData = validateOrder(req.body);

const savedOrder = orderService.save(orderData);

emailService.sendConfirmation(savedOrder.userEmail);

res.status(200).json(savedOrder);

}

### **✅ O - Open/Closed Principle**

Code should be **open to extension**, but **closed to modification**.

### **✅ Example: Instead of modifying a payment service for new methods:**

ts

CopyEdit

interface PaymentStrategy {

pay(amount: number): void;

}

class PaypalPayment implements PaymentStrategy {

pay(amount) { console.log("Paid with PayPal"); }

}

class StripePayment implements PaymentStrategy {

pay(amount) { console.log("Paid with Stripe"); }

}

function processPayment(strategy: PaymentStrategy, amount: number) {

strategy.pay(amount);

}

Now you can add new methods without touching old code.

### **✅ L - Liskov Substitution Principle**

A child class should be replaceable without breaking parent logic.

ts

CopyEdit

class Bird {

fly() { ... }

}

class Penguin extends Bird {

fly() { throw new Error("Penguins can't fly") } // 🚫 violates LSP

}

✅ Solution: Split into proper hierarchy:

ts

CopyEdit

class Bird { ... }

class FlyingBird extends Bird { fly() {} }

class Penguin extends Bird { swim() {} }

### **✅ I - Interface Segregation Principle**

Don’t force one interface to handle **all things**.

### **🚫 Bad:**

ts

CopyEdit

interface Animal {

fly(): void;

swim(): void;

}

### **✅ Good:**

ts

CopyEdit

interface Flyer { fly(): void }

interface Swimmer { swim(): void }

### **✅ D - Dependency Inversion**

Depend on abstractions, not implementations.

### **✅ Example in Node.js:**

ts

CopyEdit

// Instead of using DB directly:

function registerUser(user) {

return db.insert(user); // tightly coupled

}

// Use abstraction:

function registerUser(user, db: DatabaseClient) {

return db.insert(user);

}

Now you can mock DatabaseClient in tests — easier to refactor and scale.

## **✅ Summary Table**

| **Principle** | **Summary** | **Code Pattern** |
| --- | --- | --- |
| Red-Green-Refactor | TDD Cycle | Jest/Vitest tests first, logic second |
| DRY | Eliminate repetition | Custom components, hooks, utils |
| KISS | Simplicity is key | Avoid clever or complex logic |
| YAGNI | Don’t overbuild | Add only when needed |
| SOLID | Design for scale | Use service layers, abstraction, composition |