Table of Contents

# Bab 14: Best Practices dan Design Patterns di TypeScript

## Penjelasan Materi

Best Practices dan Design Patterns adalah panduan dan pola desain yang telah terbukti efektif dalam pengembangan software. Dalam TypeScript, kita dapat menerapkan pola-pola ini dengan memanfaatkan fitur-fitur type system untuk membuat kode yang lebih maintainable, scalable, dan type-safe.

## Analogi yang Mudah Dipahami

Bayangkan Design Patterns seperti resep masakan yang sudah teruji: - Singleton Pattern seperti resep untuk membuat satu porsi khusus - Factory Pattern seperti resep untuk membuat berbagai variasi dari menu dasar - Observer Pattern seperti sistem notifikasi di restoran - Strategy Pattern seperti berbagai metode memasak untuk bahan yang sama - Decorator Pattern seperti menambahkan topping pada makanan dasar

## Point Penting

1. **SOLID Principles**
   * Single Responsibility
   * Open/Closed
   * Liskov Substitution
   * Interface Segregation
   * Dependency Inversion
2. **Creational Patterns**
   * Singleton
   * Factory Method
   * Abstract Factory
   * Builder
   * Prototype
3. **Structural Patterns**
   * Adapter
   * Bridge
   * Composite
   * Decorator
   * Facade
4. **Behavioral Patterns**
   * Observer
   * Strategy
   * Command
   * State
   * Template Method
5. **TypeScript Best Practices**
   * Type Safety
   * Code Organization
   * Error Handling
   * Performance
   * Testing

## Contoh Kode dan Penjelasan

```typescript // 1. Singleton Pattern class Database { private static instance: Database; private constructor() { // Private constructor }

static getInstance(): Database {  
 if (!Database.instance) {  
 Database.instance = new Database();  
 }  
 return Database.instance;  
}  
  
query(sql: string): Promise<any> {  
 // Implementation  
 return Promise.resolve();  
}

}

// 2. Factory Method Pattern interface Product { name: string; price: number; }

interface ProductFactory { createProduct(): Product; }

class PhysicalProductFactory implements ProductFactory { createProduct(): Product { return { name: “Physical Product”, price: 100 }; } }

class DigitalProductFactory implements ProductFactory { createProduct(): Product { return { name: “Digital Product”, price: 50 }; } }

// 3. Observer Pattern interface Observer { update(data: any): void; }

class Subject { private observers: Observer[] = [];

attach(observer: Observer): void {  
 this.observers.push(observer);  
}  
  
detach(observer: Observer): void {  
 const index = this.observers.indexOf(observer);  
 if (index !== -1) {  
 this.observers.splice(index, 1);  
 }  
}  
  
notify(data: any): void {  
 this.observers.forEach(observer => observer.update(data));  
}

}

// 4. Strategy Pattern interface PaymentStrategy { pay(amount: number): void; }

class CreditCardPayment implements PaymentStrategy { pay(amount: number): void { console.log(`Paying ${amount} using Credit Card`); } }

class PayPalPayment implements PaymentStrategy { pay(amount: number): void { console.log(`Paying ${amount} using PayPal`); } }

class PaymentProcessor { constructor(private strategy: PaymentStrategy) {}

processPayment(amount: number): void {  
 this.strategy.pay(amount);  
}

}

// 5. Decorator Pattern interface Coffee { cost(): number; description(): string; }

class SimpleCoffee implements Coffee { cost(): number { return 10; }

description(): string {  
 return "Simple Coffee";  
}

}

abstract class CoffeeDecorator implements Coffee { constructor(protected coffee: Coffee) {}

cost(): number {  
 return this.coffee.cost();  
}  
  
description(): string {  
 return this.coffee.description();  
}

}

class MilkDecorator extends CoffeeDecorator { cost(): number { return this.coffee.cost() + 2; }

description(): string {  
 return \`\${this.coffee.description()} + Milk\`;  
}

}

// 6. Repository Pattern with Generic Types interface Repository { find(id: string): Promise<T | null>; findAll(): Promise<T[]>; create(item: T): Promise; update(id: string, item: T): Promise; delete(id: string): Promise; }

class UserRepository implements Repository { async find(id: string): Promise<User | null> { // Implementation return null; }

async findAll(): Promise<User[]> {  
 // Implementation  
 return [];  
}  
  
async create(user: User): Promise<User> {  
 // Implementation  
 return user;  
}  
  
async update(id: string, user: User): Promise<User> {  
 // Implementation  
 return user;  
}  
  
async delete(id: string): Promise<void> {  
 // Implementation  
}

}

// 7. Service Layer Pattern interface UserService { register(email: string, password: string): Promise; login(email: string, password: string): Promise; resetPassword(email: string): Promise; }

class UserServiceImpl implements UserService { constructor( private userRepository: Repository, private authService: AuthService ) {}

async register(email: string, password: string): Promise<User> {  
 // Implementation  
 return {} as User;  
}  
  
async login(email: string, password: string): Promise<string> {  
 // Implementation  
 return "";  
}  
  
async resetPassword(email: string): Promise<void> {  
 // Implementation  
}

}

// 8. Unit of Work Pattern class UnitOfWork { private readonly userRepository: Repository; private readonly orderRepository: Repository; private transactions: (() => Promise)[] = [];

constructor() {  
 this.userRepository = new UserRepository();  
 this.orderRepository = new OrderRepository();  
}  
  
getUserRepository(): Repository<User> {  
 return this.userRepository;  
}  
  
getOrderRepository(): Repository<Order> {  
 return this.orderRepository;  
}  
  
addTransaction(transaction: () => Promise<void>): void {  
 this.transactions.push(transaction);  
}  
  
async commit(): Promise<void> {  
 for (const transaction of this.transactions) {  
 await transaction();  
 }  
 this.transactions = [];  
}  
  
async rollback(): Promise<void> {  
 this.transactions = [];  
}

}

// 9. Dependency Injection Pattern interface Logger { log(message: string): void; }

class ConsoleLogger implements Logger { log(message: string): void { console.log(message); } }

class UserController { constructor( private userService: UserService, private logger: Logger ) {}

async createUser(email: string, password: string): Promise<User> {  
 this.logger.log(\`Creating user with email: \${email}\`);  
 return this.userService.register(email, password);  
}

}

// 10. Builder Pattern with Fluent Interface class EmailBuilder { private email: Email = { to: ““, from:”“, subject:”“, body:”“, attachments: [] };

setTo(to: string): EmailBuilder {  
 this.email.to = to;  
 return this;  
}  
  
setFrom(from: string): EmailBuilder {  
 this.email.from = from;  
 return this;  
}  
  
setSubject(subject: string): EmailBuilder {  
 this.email.subject = subject;  
 return this;  
}  
  
setBody(body: string): EmailBuilder {  
 this.email.body = body;  
 return this;  
}  
  
addAttachment(attachment: string): EmailBuilder {  
 this.email.attachments.push(attachment);  
 return this;  
}  
  
build(): Email {  
 return { ...this.email };  
}

} ```

## Cara Kerja Design Patterns

1. **Creational Patterns**:
   * Object creation
   * Instance management
   * Flexibility
   * Reusability
2. **Structural Patterns**:
   * Object composition
   * Interface adaptation
   * Functionality extension
   * Decoupling
3. **Behavioral Patterns**:
   * Object communication
   * Responsibility distribution
   * Algorithm encapsulation
   * State management

## Tips dan Trik

1. **SOLID Principles Implementation**

* // ✅ Single Responsibility Principle  
  class UserService {  
   constructor(  
   private userRepository: UserRepository,  
   private emailService: EmailService,  
   private logger: Logger  
   ) {}  
    
   async createUser(userData: UserDTO): Promise<User> {  
   this.logger.log('Creating new user');  
   const user = await this.userRepository.create(userData);  
   await this.emailService.sendWelcomeEmail(user.email);  
   return user;  
   }  
  }

1. **Type Safety**

* // ✅ Gunakan type guards dan generics  
  function isError<T>(value: T | Error): value is Error {  
   return value instanceof Error;  
  }  
    
  async function processData<T>(data: T): Promise<Result<T>> {  
   try {  
   // Process data  
   return { success: true, data };  
   } catch (error) {  
   if (isError(error)) {  
   return { success: false, error: error.message };  
   }  
   return { success: false, error: 'Unknown error' };  
   }  
  }

1. **Error Handling**

* // ✅ Gunakan custom error classes  
  class ValidationError extends Error {  
   constructor(  
   message: string,  
   public readonly field: string  
   ) {  
   super(message);  
   this.name = 'ValidationError';  
   }  
  }  
    
  function validate(data: unknown): void {  
   if (!isValidData(data)) {  
   throw new ValidationError('Invalid data', 'data');  
   }  
  }

## Kesalahan yang Sering Dilakukan Pemula

1. **Tight Coupling**

* // ❌ Buruk: Tight coupling  
  class UserService {  
   private database = new Database();  
   private logger = new ConsoleLogger();  
  }  
    
  // ✅ Baik: Dependency injection  
  class UserService {  
   constructor(  
   private database: Database,  
   private logger: Logger  
   ) {}  
  }

1. **Not Using Interfaces**

* // ❌ Buruk: Concrete implementation  
  class PaymentProcessor {  
   processPayment(payment: CreditCardPayment) {  
   // Implementation  
   }  
  }  
    
  // ✅ Baik: Interface-based  
  interface Payment {  
   process(): Promise<void>;  
  }  
    
  class PaymentProcessor {  
   processPayment(payment: Payment) {  
   return payment.process();  
   }  
  }

1. **Violating SOLID Principles**

* // ❌ Buruk: Violating Single Responsibility  
  class User {  
   saveToDatabase() {}  
   sendEmail() {}  
   validateData() {}  
  }  
    
  // ✅ Baik: Separate responsibilities  
  class User {  
   getData() {}  
  }  
    
  class UserRepository {  
   save(user: User) {}  
  }  
    
  class UserMailer {  
   sendEmail(user: User) {}  
  }

### Solusi:

1. Ikuti SOLID principles
2. Gunakan dependency injection
3. Buat interface yang jelas
4. Pisahkan concerns
5. Implementasi proper error handling