TypeScript SOLID Principles and Design Patterns for Learning

June 25, 2025

Created by Sarthak Bhatnagar

Contents

1 SOLID Principles in TypeScript

SOLID principles provide a foundation for writing maintainable, scalable, and robust code. They ensure systems are modular, testable, and adaptable to change. Below, each principle is explained with detailed TypeScript examples, use cases, and benefits.

1.1 Single Responsibility Principle (SRP)

Type: Design Principle
The Single Responsibility Principle states that a class should have only one reason to change, meaning it should handle a single responsibility. This reduces complexity and enhances maintainability.

Use Case: Separating user data management from notification services to isolate changes in each domain.

```
class User {
    constructor(public name: string, public email: string) {}
    save(): void {
        console.log(”Saving user to database”);
        // Database save logic
    }
}

class NotificationService {
    sendEmail(email: string, message: string): void {
        console.log(‘Sending email to ${email}: ${message}‘);
        // Email sending logic
    }
}
```


Benefits: Changes to user data management do not affect notification logic, improving maintainability and testability.

1.2 Open/Closed Principle (OCP)

Type: Design Principle
The Open/Closed Principle states that classes should be open for extension but closed for modification, allowing new functionality without altering existing code.

Use Case: Adding new payment methods to a payment processor.

```
interface Payment {
    execute(amount: number): void;
}

class CreditCardPayment implements Payment {
    execute(amount: number): void {
        console.log(‘Processing credit card payment of $${amount}‘);
        // Credit card logic
```

```
    }
}

class CryptoPayment implements Payment {
    execute(amount: number): void {
        console.log(‘Processing crypto payment of $${amount}‘);
        // Crypto logic
    }
}

class PaymentProcessor {
    constructor(private payment: Payment) {}
    process(amount: number): void {
        this.payment.execute(amount);
    }
}

// Usage
const creditCard = new CreditCardPayment();
const crypto = new CryptoPayment();
const processor1 = new PaymentProcessor(creditCard);
const processor2 = new PaymentProcessor(crypto);
processor1.process(100); // Processing credit card payment of $100
processor2.process(100); // Processing crypto payment of $100
```

Benefits: New payment methods can be added by implementing the Payment interface, ensuring scalability without modifying the PaymentProcessor.

1.3 Liskov Substitution Principle (LSP)

Type: Design Principle
The Liskov Substitution Principle ensures that subtypes can replace their base types without affecting program correctness, guaranteeing consistent polymorphic behavior.

Use Case: Managing different vehicle types in a transportation system.

```
abstract class Vehicle {
    abstract start(): void;
}

class Car extends Vehicle {
    start(): void {
        console.log(”Car engine started”);
        // Engine start logic
    }
}

class Bicycle extends Vehicle {
    start(): void {
        console.log(”Bicycle pedaling started”);
```

```
        // Pedaling logic
    }
}

function startVehicle(vehicle: Vehicle): void {
    vehicle.start();
}

// Usage
startVehicle(new Car()); // Car engine started
startVehicle(new Bicycle()); // Bicycle pedaling started
```

Benefits: LSP ensures that any Vehicle subtype can be used interchangeably, maintaining reliability and preventing unexpected behavior in polymorphic scenarios.

1.4 Interface Segregation Principle (ISP)

Type: Design Principle
The Interface Segregation Principle states that clients should not be forced to depend on interfaces they do not use, promoting lean and specific interfaces.

Use Case: Separating printer and scanning functionalities to avoid unnecessary method implementations.


```
interface Printer {
    print(document: string): void;
}

interface Scanner {
    scan(): string;
}

class SimplePrinter implements Printer {
    print(document: string): void {
        console.log(‘Printing ${document}‘);
        // Printing logic
    }
}

class MultiFunctionDevice implements Printer, Scanner {
    print(document: string): void {
        console.log(‘Printing ${document}‘);
        // Printing logic
    }
    scan(): string {
        console.log(”Scanning document”);
        return ”Scanned document”;
    }
}
```

Benefits: Devices like SimplePrinter avoid implementing unused methods, reducing complexity and improving code clarity.

1.5 Dependency Inversion Principle (DIP)

Type: Design Principle
The Dependency Inversion Principle states that high-level modules should depend on abstractions, not low-level modules, enhancing flexibility and decoupling.

Use Case: Decoupling a logging mechanism from an application service.

```
interface Logger {
    log(message: string): void;
}

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

class FileLogger implements Logger {
    log(message: string): void {
        console.log(‘Writing to file: ${message}‘);
        // File logging logic
    }
}

class AppService {
    constructor(private logger: Logger) {}
    performAction(): void {
        this.logger.log(”Action performed”);
        // Business logic
    }
}

// Usage
const consoleLogger = new ConsoleLogger();
const fileLogger = new FileLogger();
const service1 = new AppService(consoleLogger);
const service2 = new AppService(fileLogger);
service1.performAction(); // Action performed
service2.performAction(); // Writing to file: Action performed
```

Benefits: The AppService depends on the Logger interface, allowing easy swapping of logging implementations without modifying the service.

2 Design Patterns in TypeScript

Design patterns are reusable solutions to common software design problems, categorized into Creational, Structural, and Behavioral types. Below, we explore all standard patterns with detailed TypeScript examples, their types, and practical applications.

2.1 Creational Patterns

Creational patterns focus on object creation mechanisms, improving flexibility and reusability.

2.1.1 Singleton Pattern

Type: Creational
The Singleton Pattern ensures a class has only one instance and provides global access, ideal for managing shared resources.

Use Case: Managing application configuration settings.

```
class Config {
    private static instance: Config;
    private settings: Map<string, string> = new Map();

    private constructor() {
        this.settings.set("theme", "dark");
    }

    static getInstance(): Config {
        if (!Config.instance) {
            Config.instance = new Config();
        }
        return Config.instance;
    }

    getSetting(key: string): string | undefined {
        return this.settings.get(key);
    }

    setSetting(key: string, value: string): void {
        this.settings.set(key, value);
    }
}

// Usage
const config1 = Config.getInstance();
const config2 = Config.getInstance();
config1.setSetting("theme", "light");
console.log(config2.getSetting("theme")); // light
```

Benefits: Ensures a single configuration instance, preventing resource duplication and maintaining consistency across the application.

2.1.2 Factory Method Pattern

Type: Creational
The Factory Method Pattern defines an interface for creating objects, allowing subclasses to decide which class to instantiate.

Use Case: Creating different types of documents in a document management system.

```
interface Document {
    open(): void;
}

class PDFDocument implements Document {
    open(): void {
        console.log(”Opening PDF document”);
        // PDF logic
    }
}

class WordDocument implements Document {
    open(): void {
        console.log(”Opening Word document”);
        // Word logic
    }
}

abstract class DocumentCreator {
    abstract createDocument(): Document;
}

class PDFCreator extends DocumentCreator {
    createDocument(): Document {
        return new PDFDocument();
    }
}

class WordCreator extends DocumentCreator {
    createDocument(): Document {
        return new WordDocument();
    }
}

// Usage
const pdfCreator = new PDFCreator();
const wordCreator = new WordCreator();
pdfCreator.createDocument().open(); // Opening PDF document
wordCreator.createDocument().open(); // Opening Word document
```

Benefits: Allows subclasses to define object creation, promoting flexibility and extensibility.

2.1.3 Abstract Factory Pattern

Type: Creational
The Abstract Factory Pattern provides an interface for creating families of related objects without specifying their concrete classes.

Use Case: Creating UI components for different themes (e.g., light and dark).

```
interface Button {
    render(): void;
}

interface Checkbox {
    check(): void;
}

class LightButton implements Button {
    render(): void {
        console.log("Rendering light theme button");
    }
}

class LightCheckbox implements Checkbox {
    check(): void {
        console.log("Checking light theme checkbox");
    }
}

class DarkButton implements Button {
    render(): void {
        console.log("Rendering dark theme button");
    }
}

class DarkCheckbox implements Checkbox {
    check(): void {
        console.log("Checking dark theme checkbox");
    }
}

interface UIFactory {
    createButton(): Button;
    createCheckbox(): Checkbox;
}

class LightUIFactory implements UIFactory {
    createButton(): Button {
        return new LightButton();
    }
```

```
    createCheckbox(): Checkbox {
        return new LightCheckbox();
    }
}

class DarkUIFactory implements UIFactory {
    createButton(): Button {
        return new DarkButton();
    }
    createCheckbox(): Checkbox {
        return new DarkCheckbox();
    }
}

// Usage
const lightFactory = new LightUIFactory();
const darkFactory = new DarkUIFactory();
lightFactory.createButton().render(); // Rendering light theme
   button
darkFactory.createCheckbox().check(); // Checking dark theme
   checkbox
```

Bnfits: Ensures consistency among related objects and supports theme switching without modifying client code.

2.1.4 Builder Pattern

Type: Creational
The Builder Pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

Use Case: Building a complex product configuration with multiple parts.

```
class Product {
    parts: string[] = [];

    addPart(part: string): void {
        this.parts.push(part);
    }

    show(): void {
        console.log(‘Product parts: ${this.parts.join(”, ”)}‘);
    }
}

interface Builder {
    addPartA(): void;
    addPartB(): void;
    addPartC(): void;
    getProduct(): Product;
```

```
}

class ConcreteBuilder implements Builder {
    private product: Product = new Product();

    addPartA(): void {
        this.product.addPart("Part A");
    }
    addPartB(): void {
        this.product.addPart("Part B");
    }
    addPartC(): void {
        this.product.addPart("Part C");
    }
    getProduct(): Product {
        return this.product;
    }
}

class Director {
    construct(builder: Builder): void {
        builder.addPartA();
        builder.addPartB();
        builder.addPartC();
    }
}

// Usage
const builder = new ConcreteBuilder();
const director = new Director();
director.construct(builder);
builder.getProduct().show(); // Product parts: Part A, Part B, Part
   C
```


Benefits: Enables step-by-step construction and supports varying product configurations.

2.1.5 Prototype Pattern

Type: Creational
The Prototype Pattern creates new objects by copying an existing object, known as the prototype.

Use Case: Cloning shapes in a graphics editor.

```
interface Prototype {
    clone(): Prototype;
}

class Shape implements Prototype {
    constructor(public type: string, public color: string) {}
```

```
7
8       clone(): Prototype {
9           return new Shape(this.type, this.color);
10      }
11  }
12
13  // Usage
14  const circle = new Shape("Circle", "Red");
15  const clonedCircle = circle.clone();
16  console.log(clonedCircle.type, clonedCircle.color); // Circle, Red
```

Benefits: Reduces the cost of object creation by cloning existing instances, useful for complex objects.

2.2 Structural Patterns

Structural patterns focus on object composition to form larger structures.

2.2.1 Adapter Pattern

Type: Structural
The Adapter Pattern allows incompatible interfaces to work together by wrapping an existing class with a new interface.

Use Case: Integrating a legacy payment system with a modern API.

```
1   interface NewPaymentSystem {
2       pay(amount: number): void;
3   }
4
5   class LegacyPayment {
6       makePayment(cents: number): void {
7           console.log(`Processing payment of ${cents} cents`);
8       }
9   }
10
11  class PaymentAdapter implements NewPaymentSystem {
12      constructor(private legacy: LegacyPayment) {}
13      pay(amount: number): void {
14          this.legacy.makePayment(amount * 100);
15      }
16  }
17
18  // Usage
19  const legacyPayment = new LegacyPayment();
20  const adapter = new PaymentAdapter(legacyPayment);
21  adapter.pay(50); // Processing payment of 5000 cents
```

Benefits: Enables integration of legacy code with new systems without modifying existing code.

2.2.2 Bridge Pattern

Type: Structural
The Bridge Pattern decouples an abstraction from its implementation, allowing them to vary independently.

Use Case: Rendering shapes with different rendering APIs.

```
interface Renderer {
    renderShape(shape: string): void;
}

class VectorRenderer implements Renderer {
    renderShape(shape: string): void {
        console.log(‘Rendering ${shape} with vector graphics‘);
    }
}

class RasterRenderer implements Renderer {
    renderShape(shape: string): void {
        console.log(‘Rendering ${shape} with raster graphics‘);
    }
}

abstract class Shape {
    constructor(protected renderer: Renderer) {}
    abstract draw(): void;
}

class Circle extends Shape {
    draw(): void {
        this.renderer.renderShape(”Circle”);
    }
}

// Usage
const vector = new VectorRenderer();
const raster = new RasterRenderer();
const circle = new Circle(vector);
circle.draw(); // Rendering Circle with vector graphics
```

Benefits: Allows independent variation of abstraction and implementation, enhancing flexibility.

2.2.3 Composite Pattern

Type: Structural
The Composite Pattern composes objects into tree structures to represent part-whole hierarchies, treating individual objects and compositions uniformly.

Use Case: Managing a file system with files and directories.

```
interface FileSystemComponent {
    showDetails(): void;
}

class File implements FileSystemComponent {
    constructor(private name: string) {}
    showDetails(): void {
        console.log(‘File: ${this.name}‘);
    }
}

class Directory implements FileSystemComponent {
    private components: FileSystemComponent[] = [];
    constructor(private name: string) {}

    add(component: FileSystemComponent): void {
        this.components.push(component);
    }

    showDetails(): void {
        console.log(‘Directory: ${this.name}‘);
        this.components.forEach(c => c.showDetails());
    }
}

// Usage
const file1 = new File(”file1.txt”);
const file2 = new File(”file2.txt”);
const dir = new Directory(”Docs”);
dir.add(file1);
dir.add(file2);
dir.showDetails(); // Directory: Docs, File: file1.txt, File: file2.
   txt
```


Benefits: Simplifies client code by treating individual and composite objects uniformly.

2.2.4 Decorator Pattern

Type: Structural
The Decorator Pattern attaches additional responsibilities to objects dynamically, providing a flexible alternative to subclassing.

Use Case: Adding features to a coffee order.

```
interface Coffee {
    cost(): number;
    description(): string;
}

class SimpleCoffee implements Coffee {
```

```
    cost(): number {
        return 5;
    }
    description(): string {
        return "Simple Coffee";
    }
}

abstract class CoffeeDecorator implements Coffee {
    constructor(protected coffee: Coffee) {}
    abstract cost(): number;
    abstract description(): string;
}

class MilkDecorator extends CoffeeDecorator {
    cost(): number {
        return this.coffee.cost() + 2;
    }
    description(): string {
        return `${this.coffee.description()}, Milk`;
    }
}

// Usage
let coffee: Coffee = new SimpleCoffee();
coffee = new MilkDecorator(coffee);
console.log(coffee.description(), coffee.cost()); // Simple Coffee,
   Milk, 7
```

Benefits: Enables dynamic addition of features without modifying the original class.

2.2.5 Facade Pattern

Type: Structural
The Facade Pattern provides a simplified interface to a complex subsystem.

Use Case: Simplifying interaction with a multimedia system.

```
class AudioPlayer {
    playAudio(): void {
        console.log("Playing audio");
    }
}

class VideoPlayer {
    playVideo(): void {
        console.log("Playing video");
    }
}

```

```
class MultimediaFacade {
    private audioPlayer = new AudioPlayer();
    private videoPlayer = new VideoPlayer();

    playMultimedia(): void {
        this.audioPlayer.playAudio();
        this.videoPlayer.playVideo();
    }
}

// Usage
const facade = new MultimediaFacade();
facade.playMultimedia(); // Playing audio, Playing video
```

Benefits: Simplifies client interaction with complex subsystems, improving usability.

2.2.6 Flyweight Pattern

Type: Structural
The Flyweight Pattern minimizes memory usage by sharing common data among multiple objects.

Use Case: Managing character styles in a text editor.

```
interface CharacterStyle {
    render(char: string): void;
}

class CharacterStyleFlyweight implements CharacterStyle {
    constructor(private font: string, private size: number) {}
    render(char: string): void {
        console.log(`Rendering ${char} in ${this.font}, size ${this.
            size}`);
    }
}

class FlyweightFactory {
    private styles: Map<string, CharacterStyle> = new Map();

    getStyle(font: string, size: number): CharacterStyle {
        const key = `${font}-${size}`;
        if (!this.styles.has(key)) {
            this.styles.set(key, new CharacterStyleFlyweight(font,
                size));
        }
        return this.styles.get(key)!;
    }
}

// Usage
```

```
const factory = new FlyweightFactory();
const style1 = factory.getStyle("Arial", 12);
const style2 = factory.getStyle("Arial", 12);
style1.render("A"); // Rendering A in Arial, size 12
console.log(style1 === style2); // true
```

Benefits: Reduces memory usage by sharing common data, ideal for large numbers of similar objects.

2.2.7 Proxy Pattern

Type: Structural
The Proxy Pattern provides a surrogate or placeholder for another object to control access to it.

Use Case: Lazy loading of large images.

```
interface Image {
    display(): void;
}

class RealImage implements Image {
    constructor(private filename: string) {
        console.log(`Loading ${filename}`);
    }
    display(): void {
        console.log(`Displaying ${this.filename}`);
    }
}

class ImageProxy implements Image {
    private image: RealImage | null = null;
    constructor(private filename: string) {}
    display(): void {
        if (!this.image) {
            this.image = new RealImage(this.filename);
        }
        this.image.display();
    }
}

// Usage
const image = new ImageProxy("large.jpg");
image.display(); // Loading large.jpg, Displaying large.jpg
```

Benefits: Controls access to resources, enabling lazy loading and additional functionality like caching.

2.3 Behavioral Patterns

Behavioral patterns focus on communication between objects, improving flexibility and coordination.

2.3.1 Observer Pattern

Type: Behavioral
The Observer Pattern defines a one-to-many dependency where observers are notified of state changes in a subject, ideal for event-driven systems.

Use Case: Real-time stock market updates to multiple displays.

```
interface Observer {
    update(data: string): void;
}

interface Subject {
    registerObserver(observer: Observer): void;
    removeObserver(observer: Observer): void;
    notify(data: string): void;
}

class StockMarket implements Subject {
    private observers: Observer[] = [];
    private stockData: string = ”“;

    registerObserver(observer: Observer): void {
        this.observers.push(observer);
    }

    removeObserver(observer: Observer): void {
        this.observers = this.observers.filter(obs => obs !==
           observer);
    }

    notify(data: string): void {
        this.observers.forEach(observer => observer.update(data));
    }

    updateStockPrice(data: string): void {
        this.stockData = data;
        this.notify(data);
    }
}

class StockDisplay implements Observer {
    update(data: string): void {
        console.log(‘Display updated with: ${data}‘);
    }
}

```

```
// Usage
const market = new StockMarket();
const display1 = new StockDisplay();
const display2 = new StockDisplay();
market.registerObserver(display1);
market.registerObserver(display2);
market.updateStockPrice("AAPL: $150"); // Display updated with: AAPL
   : $150
```

Benefits: Decouples subjects from observers, enabling dynamic addition/removal of observers and supporting scalable event systems.

2.3.2 Strategy Pattern

Type: Behavioral
The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable, allowing dynamic behavior changes.

Use Case: Sorting data using different algorithms.

```
interface SortStrategy {
    sort(data: number[]): number[];
}

class BubbleSort implements SortStrategy {
    sort(data: number[]): number[] {
        console.log("Sorting with BubbleSort");
        return [...data].sort((a, b) => a - b); // Simplified for
           demo
    }
}

class QuickSort implements SortStrategy {
    sort(data: number[]): number[] {
        console.log("Sorting with QuickSort");
        return [...data].sort((a, b) => a - b); // Simplified for
           demo
    }
}

class Sorter {
    constructor(private strategy: SortStrategy) {}

    setStrategy(strategy: SortStrategy): void {
        this.strategy = strategy;
    }

    sort(data: number[]): number[] {
        return this.strategy.sort(data);
    }
}
```

```
30
31 // Usage
32 const sorter = new Sorter(new BubbleSort());
33 console.log(sorter.sort([3, 1, 2])); // Sorting with BubbleSort, [1,
       2, 3]
34 sorter.setStrategy(new QuickSort());
35 console.log(sorter.sort([3, 1, 2])); // Sorting with QuickSort, [1,
      2, 3]
```

Benefits: Enables runtime switching of algorithms, promoting flexibility and reusability.

2.3.3 Template Method Pattern

Type: Behavioral
The Template Method Pattern defines the skeleton of an algorithm, allowing subclasses to customize specific steps without altering the structure.

Use Case: Processing different data file formats with a consistent workflow.

```
1  abstract class DataProcessor {
2      process(): void {
3          this.readData();
4          this.processData();
5          this.saveData();
6      }                          Sarthak Education
7
8      abstract readData(): void;
9      abstract processData(): void;
10
11     saveData(): void {
12         console.log("Saving data to database");
13         // Save logic
14     }
15 }
16
17 class CSVProcessor extends DataProcessor {
18     readData(): void {
19         console.log("Reading CSV file");
20         // CSV reading logic
21     }
22
23     processData(): void {
24         console.log("Processing CSV data");
25         // CSV processing logic
26     }
27 }
28
29 class JSONProcessor extends DataProcessor {
30     readData(): void {
31         console.log("Reading JSON file");
```

```
        // JSON reading logic
    }

    processData(): void {
        console.log("Processing JSON data");
        // JSON processing logic
    }
}

// Usage
const csvProcessor = new CSVProcessor();
csvProcessor.process(); // Reading CSV file, Processing CSV data,
   Saving data to database
```

Benefits: Ensures a consistent algorithm structure while allowing customization of specific steps.

2.3.4 Visitor Pattern

Type: Behavioral
The Visitor Pattern separates algorithms from object structures, allowing new operations without modifying classes.

Use Case: Exporting shapes to different formats.

```
interface Visitor {
    visitCircle(circle: Circle): void;
    visitRectangle(rectangle: Rectangle): void;
}

interface Shape {
    accept(visitor: Visitor): void;
}

class Circle implements Shape {
    constructor(private radius: number) {}

    accept(visitor: Visitor): void {
        visitor.visitCircle(this);
    }

    getRadius(): number {
        return this.radius;
    }
}

class Rectangle implements Shape {
    constructor(private width: number, private height: number) {}

    accept(visitor: Visitor): void {
        visitor.visitRectangle(this);
```

```
    }

    getDimensions(): { width: number; height: number } {
        return { width: this.width, height: this.height };
    }
}

class XMLExportVisitor implements Visitor {
    visitCircle(circle: Circle): void {
        console.log(`Exporting Circle with radius ${circle.getRadius
           ()} to XML`);
        // XML export logic
    }

    visitRectangle(rectangle: Rectangle): void {
        const dims = rectangle.getDimensions();
        console.log(`Exporting Rectangle with width ${dims.width}
           and height ${dims.height} to XML`);
    }
}

class JSONExportVisitor implements Visitor {
    visitCircle(circle: Circle): void {
        console.log(`Exporting Circle with radius ${circle.getRadius
           ()} to JSON`);
        // JSON export logic
    }

    visitRectangle(rectangle: Rectangle): void {
        const dims = rectangle.getDimensions();
        console.log(`Exporting Rectangle with width ${dims.width}
           and height ${dims.height} to JSON`);
    }
}

// Usage
const circle = new Circle(5);
const rectangle = new Rectangle(10, 20);
const xmlVisitor = new XMLExportVisitor();
const jsonVisitor = new JSONExportVisitor();
circle.accept(xmlVisitor); // Exporting Circle with radius 5 to XML
rectangle.accept(jsonVisitor); // Exporting Rectangle with width 10
   and height 20 to JSON
```


Benefits: Allows adding new operations without modifying object structures, enhancing extensibility.

2.3.5 Chain of Responsibility Pattern

Type: Behavioral
The Chain of Responsibility Pattern passes requests along a chain of handlers,

allowing multiple objects to handle a request.

Use Case: Processing support tickets with different priority levels.

```
interface Handler {
    setNext(handler: Handler): Handler;
    handle(request: string): string | null;
}

abstract class AbstractHandler implements Handler {
    private nextHandler: Handler | null = null;

    setNext(handler: Handler): Handler {
        this.nextHandler = handler;
        return handler;
    }

    handle(request: string): string | null {
        if (this.nextHandler) {
            return this.nextHandler.handle(request);
        }
        return null;
    }
}

class LowPriorityHandler extends AbstractHandler {
    handle(request: string): string | null {
        if (request === "low") {
            return "Handled by LowPriorityHandler";
        }
        return super.handle(request);
    }
}

class HighPriorityHandler extends AbstractHandler {
    handle(request: string): string | null {
        if (request === "high") {
            return "Handled by HighPriorityHandler";
        }
        return super.handle(request);
    }
}

// Usage
const lowHandler = new LowPriorityHandler();
const highHandler = new HighPriorityHandler();
lowHandler.setNext(highHandler);
console.log(lowHandler.handle("low")); // Handled by
   LowPriorityHandler
console.log(lowHandler.handle("high")); // Handled by
   HighPriorityHandler
```

Benefits: Decouples senders from receivers, allowing dynamic handler chains and flexible request handling.

2.3.6 Command Pattern

Type: Behavioral
The Command Pattern encapsulates a request as an object, enabling parameterization, queuing, and undoable operations.

Use Case: Implementing undo/redo in a text editor.

```
interface Command {
    execute(): void;
    undo(): void;
}

class TextEditor {
    private content: string = "";

    append(text: string): void {
        this.content += text;
        console.log(`Content: ${this.content}`);
    }

    remove(length: number): void {
        this.content = this.content.slice(0, -length);
        return this.content;
    }
}

class AppendCommand implements Command {
    constructor(private editor: TextEditor, private text: string) {}

    execute(): void {
        this.editor.append(this.text);
    }

    undo(): void {
        this.editor.remove(this.text.length);
    }
}

class EditorHistory {
    private commands: Command[] = [];

    push(command: Command): void {
        this.commands.push(command);
    }

    pop(): Command | undefined {
        return this.commands.pop();
    }
```

```
}

// Usage
const editor = new TextEditor();
const history = new EditorHistory();
const command = new AppendCommand(editor, "Hello");
command.execute(); // Content: Hello
history.push(command);
const undoneCommand = history.pop();
undoneCommand?.undo(); // Content after undo:
```

Benefits: Encapsulates operations, enabling features like undo/redo and command queuing.

2.3.7 Iterator Pattern

Type: Behavioral
The Iterator Pattern provides a way to access elements of a collection sequentially without exposing its underlying structure.

Use Case: Iterating over a collection of books.

: Provides a standardized way to traverse collections, hiding their internal structure.

2.3.8 Mediator Pattern

Type: Behavioral
The Mediator Pattern defines an object that encapsulates how a set of objects interact, promoting loose coupling.

Use Case: Managing communication between UI components.

```
interface Mediator {
    notify(sender: Component, event: string): void;
}

class Component {
    constructor(private mediator: Mediator, private name: string) {}

    triggerEvent(event: string): void {
        this.mediator.notify(this, event);
    }
}

class UIMediator implements Mediator {
    private components: Component[] = [];

    addComponent(component: Component): void {
        this.components.push(component);
    }

```

```
    notify(sender: Component, event: string): void {
        console.log(`${sender} triggered ${event}`);
        // Coordinate components
    }
}

// Usage
const mediator = new UIMediator();
const button = new Component(mediator, "Button");
const textBox = new Component(mediator, "TextBox");
mediator.addComponent(button);
mediator.addComponent(textBox);
button.triggerEvent("click"); // Button triggered click
```

Benefits: Reduces direct dependencies between objects, simplifying communication and maintenance.

2.3.9 Memento Pattern

Type: Behavioral
The Memento Pattern captures and externalizes an object's internal state for later restoration without violating encapsulation.

Use Case: Saving and restoring editor states.

```
class EditorMemento {
    constructor(private content: string) {}
    getContent(): string {
        return this.content;
    }
}

class Editor {
    private content: string = "";

    setContent(content: string): void {
        this.content = content;
    }

    getContent(): string {
        return this.content;
    }

    save(): EditorMemento {
        return new EditorMemento(this.content);
    }

    restore(memento: EditorMemento): void {
        this.content = memento.getContent();
    }
}
```

```

class Caretaker {
    private mementos: EditorMemento[] = [];

    addMemento(memento: EditorMemento): void {
        this.mementos.push(memento);
    }

    getMemento(index: number): EditorMemento {
        return this.mementos[index];
    }
}

// Usage
const editor = new Editor();
const caretaker = new Caretaker();
editor.setContent("State 1");
caretaker.addMemento(editor.save());
editor.setContent("State 2");
editor.restore(caretaker.getMemento(0));
console.log(editor.getContent()); // State 1
```

Benefits: Enables state restoration without exposing internal details, useful for undo functionality.

2.3.10 State Pattern

Type: Behavioral
The State Pattern allows an object to alter its behavior when its internal state changes, appearing as if it changes its class.

Use Case: Managing states of a traffic light.

```
interface State {
    handle(context: TrafficLight): void;
}

class RedState implements State {
    handle(context: TrafficLight): void {
        console.log("Red light: Stop");
        context.setState(new GreenState());
    }
}

class GreenState implements State {
    handle(context: TrafficLight): void {
        console.log("Green light: Go");
        context.setState(new YellowState());
    }
}

```

```
class YellowState implements State {
    handle(context: TrafficLight): void {
        console.log("Yellow light: Slow down");
        context.setState(new RedState());
    }
}

class TrafficLight {
    private state: State;

    constructor() {
        this.state = new RedState();
    }

    setState(state: State): void {
        this.state = state;
    }

    change(): void {
        this.state.handle(this);
    }
}

// Usage
const trafficLight = new TrafficLight();
trafficLight.change(); // Red light: Stop
trafficLight.change(); // Green light: Go
trafficLight.change(); // Yellow light: Slow down
```

Benefits: Encapsulates state-specific behavior, making state transitions explicit and maintainable.

2.3.11 Interpreter Pattern

Type: Behavioral
The Interpreter Pattern defines a representation for a language's grammar and an interpreter to process it.

Use Case: Evaluating simple arithmetic expressions.

```
interface Expression {
    interpret(): number;
}

class NumberExpression implements Expression {
    constructor(private value: number) {}
    interpret(): number {
        return this.value;
    }
}

```

```
class AddExpression implements Expression {
    constructor(private left: Expression, private right: Expression)
        {}
    interpret(): number {
        return this.left.interpret() + this.right.interpret();
    }
}

// Usage
const expression = new AddExpression(new NumberExpression(5), new
   NumberExpression(3));
console.log(expression.interpret()); // 8
```

Benefits: Allows defining and interpreting custom languages, useful for scripting or configuration.

3 Summary

SOLID principles ensure clean, maintainable, and scalable code by promoting modularity and flexibility. Design patterns, categorized into Creational, Structural, and Behavioral types, provide reusable solutions to common problems. By combining SOLID principles with these patterns in TypeScript, developers can build robust, adaptable software systems for diverse applications.

Acknowledgements

Proud to be trained by Learn2Earn Labs (www.learntoearnlabs.com) – where learning meets real-world industry exposure.

For more details, contact us:
Email: sarthakblatnagar2005@gmail.com
GitHub: https://github.com/Sarthakbhai395
LinkedIn: https://www.linkedin.com/in/sarthak-blatnagar-801a02343