

# Introduction aux Design Patterns

# Qu'est-ce qu'un Design Pattern ?

- **Définition** : Solution réutilisable à un problème récurrent en conception logicielle.
- **Origine** : Introduit par le *Gang of Four* dans *Design Patterns: Elements of Reusable Object-Oriented Software*.
- **Objectif** : Simplifier, rendre le code plus maintenable et réutilisable.

# Classification des Design Patterns

- **Patrons de Création** : Gestion de la création d'objets.
- **Patrons Structurels** : Composition des classes et des objets.
- **Patrons Comportementaux** : Interaction et responsabilité entre les objets.

# Patrons de Création

# Singleton

- **Problème** : S'assurer qu'une classe n'a qu'une seule instance.
- **Solution** : Garantir une instance unique et fournir un point d'accès global.
- **Métaphore** : Un président dans un pays : une seule personne à la fois, connue de tous.



```
1  class Singleton {
2      private static instance: Singleton;
3
4      private constructor() {
5          // Initialisation
6      }
7
8      static getInstance(): Singleton {
9          if (!Singleton.instance) {
10              Singleton.instance = new Singleton();
11          }
12          return Singleton.instance;
13      }
14
15     someBusinessLogic() {
16         console.log("Executing business logic...");
17     }
18 }
19
20 // Utilisation
21 const singleton1 = Singleton.getInstance();
22 const singleton2 = Singleton.getInstance();
23 console.log(singleton1 === singleton2); // true
```

# Fabrique Abstraite (*Abstract Factory*)

- **Problème** : Créer des familles d'objets liés sans spécifier leurs classes concrètes.
- **Solution** : Fournir une interface pour créer des familles d'objets.
- **Métaphore** : Une entreprise de meubles proposant des ensembles cohérents de salon.

```
 1 interface Button {
 2     render(): void;
 3 }
 4
 5 interface Checkbox {
 6     render(): void;
 7 }
 8
 9 interface GUIFactory {
10     createButton(): Button;
11     createCheckbox(): Checkbox;
12 }
13
14 class WindowsButton implements Button {
15     render(): void {
16         console.log("Rendering Windows Button.");
17     }
18 }
19
20 class WindowsCheckbox implements Checkbox {
21     render(): void {
22         console.log("Rendering Windows Checkbox.");
23     }
24 }
25
26 class MacButton implements Button {
27     render(): void {
28         console.log("Rendering Mac Button.");
29     }
30 }
31
32 class MacCheckbox implements Checkbox {
33     render(): void {
34         console.log("Rendering Mac Checkbox.");
35     }
36 }
37
38 class WindowsFactory implements GUIFactory {
39     createButton(): Button {
40         return new WindowsButton();
41     }
42
43     createCheckbox(): Checkbox {
44         return new WindowsCheckbox();
45     }
46 }
47
48 class MacFactory implements GUIFactory {
49     createButton(): Button {
50         return new MacButton();
51     }
52
53     createCheckbox(): Checkbox {
54         return new MacCheckbox();
55     }
56 }
57
58 // Utilisation
59 const factory: GUIFactory = new WindowsFactory();
60 const button: Button = factory.createButton();
61 const checkbox: Checkbox = factory.createCheckbox();
62 button.render();
63 checkbox.render();
```

# Constructeur (*Builder*)

- **Problème** : Construire un objet complexe étape par étape.
- **Solution** : Séparer la construction de l'objet de sa représentation.
- **Métaphore** : Un chef cuisinier ajustant une recette pour différentes variantes.

```
● ● ●
1 class Car {
2   wheels: number;
3   engine: string;
4   color: string;
5
6   constructor(builder: CarBuilder) {
7     this.wheels = builder.wheels;
8     this.engine = builder.engine;
9     this.color = builder.color;
10    }
11 }
12
13 class CarBuilder {
14   wheels: number = 4;
15   engine: string = "default engine";
16   color: string = "white";
17
18   setWheels(wheels: number): CarBuilder {
19     this.wheels = wheels;
20     return this;
21   }
22
23   setEngine(engine: string): CarBuilder {
24     this.engine = engine;
25     return this;
26   }
27
28   setColor(color: string): CarBuilder {
29     this.color = color;
30     return this;
31   }
32
33   build(): Car {
34     return new Car(this);
35   }
36 }
37
38 // Utilisation
39 const car: Car = new CarBuilder()
40   .setWheels(4)
41   .setEngine("V8")
42   .setColor("red")
43   .build();
44
45 console.log(car);
```

# Méthode Fabrique (*Factory Method*)

- **Problème** : Créer un objet sans spécifier sa classe concrète.
- **Solution** : Définir une interface pour la création d'un objet, laissant aux sous-classes le soin de décider quelle classe instancier.
- **Métaphore** : Une boulangerie proposant différents types de pains sans que le client connaisse la recette.

```
● ● ●
1 abstract class Creator {
2     abstract factoryMethod(): Product;
3
4     someOperation(): string {
5         const product = this.factoryMethod();
6         return `Creator: Working with ${product.operation()}`;
7     }
8 }
9
10 interface Product {
11     operation(): string;
12 }
13
14 class ConcreteProductA implements Product {
15     operation(): string {
16         return "ConcreteProductA";
17     }
18 }
19
20 class ConcreteProductB implements Product {
21     operation(): string {
22         return "ConcreteProductB";
23     }
24 }
25
26 class ConcreteCreatorA extends Creator {
27     factoryMethod(): Product {
28         return new ConcreteProductA();
29     }
30 }
31
32 class ConcreteCreatorB extends Creator {
33     factoryMethod(): Product {
34         return new ConcreteProductB();
35     }
36 }
37
38 // Utilisation
39 const creator: Creator = new ConcreteCreatorA();
40 console.log(creator.someOperation());
```

# Prototype

- **Problème** : Créer de nouveaux objets en copiant une instance existante.
- **Solution** : Spécifier les types d'objets à créer en utilisant une instance prototype.
- **Métaphore** : Un artiste créant des copies d'une sculpture originale en utilisant un moule.



```
1  interface Prototype {
2      clone(): Prototype;
3  }
4
5  class ConcretePrototype implements Prototype {
6      field: string;
7
8      constructor(field: string) {
9          this.field = field;
10     }
11
12     clone(): this {
13         return Object.create(this);
14     }
15 }
16
17 // Utilisation
18 const prototype = new ConcretePrototype("Initial Value");
19 const clone = prototype.clone();
20 console.log(clone.field); // "Initial Value"
```

# Patrons Structurels

# Adaptateur (*Adapter*)

- **Problème** : Permettre à des interfaces incompatibles de fonctionner ensemble.
- **Solution** : Convertir l'interface d'une classe en une autre attendue par les clients.
- **Métaphore** : Un adaptateur de prise électrique permettant de brancher un appareil étranger.

```
1 // Interface existante
2 class OldSystem {
3     request(): string {
4         return "Old system response";
5     }
6 }
7
8 // Interface attendue
9 interface Target {
10    specificRequest(): string;
11 }
12
13 // Adaptateur
14 class Adapter implements Target {
15     private adaptee: OldSystem;
16
17     constructor(adaptee: OldSystem) {
18         this.adaptee = adaptee;
19     }
20
21     specificRequest(): string {
22         return this.adaptee.request();
23     }
24 }
25
26 // Utilisation
27 const oldSystem = new OldSystem();
28 const adapter = new Adapter(oldSystem);
29 console.log(adapter.specificRequest()); // "Old system response"
```

# Pont (*Bridge*)

- **Problème** : Séparer une abstraction de son implémentation pour qu'elles puissent évoluer indépendamment.
- **Solution** : Découpler l'abstraction de son implémentation.
- **Métaphore** : Une télécommande universelle contrôlant différents appareils électroniques.

```
1 // Abstraction
2 abstract class Shape {
3     protected color: Color;
4
5     constructor(color: Color) {
6         this.color = color;
7     }
8
9     abstract draw(): void;
10}
11
12 // Implémentation
13 interface Color {
14     fill(): string;
15}
16
17 class Red implements Color {
18     fill(): string {
19         return "Red color";
20     }
21}
22
23 class Blue implements Color {
24     fill(): string {
25         return "Blue color";
26     }
27}
28
29 // Classes concrètes
30 class Circle extends Shape {
31     draw(): void {
32         console.log(`Drawing Circle with ${this.color.fill()}`);
33     }
34}
35
36 class Square extends Shape {
37     draw(): void {
38         console.log(`Drawing Square with ${this.color.fill()}`);
39     }
40}
41
42 // Utilisation
43 const redCircle = new Circle(new Red());
44 redCircle.draw();
45
46 const blueSquare = new Square(new Blue());
47 blueSquare.draw();
```

# Composite

- **Problème** : Composer des objets en structures arborescentes pour représenter des hiérarchies partie-tout.
- **Solution** : Permettre aux clients de traiter les objets individuels et les compositions de manière uniforme.
- **Métaphore** : Un dossier sur un ordinateur contenant des fichiers et d'autres dossiers.

```
1  interface Component {
2      operation(): string;
3  }
4
5  class Leaf implements Component {
6      operation(): string {
7          return "Leaf";
8      }
9  }
10
11 class Composite implements Component {
12     private children: Component[] = [];
13
14     add(component: Component): void {
15         this.children.push(component);
16     }
17
18     operation(): string {
19         return this.children.map(child => child.operation()).join(", ");
20     }
21 }
22
23 // Utilisation
24 const leaf1 = new Leaf();
25 const leaf2 = new Leaf();
26 const composite = new Composite();
27
28 composite.add(leaf1);
29 composite.add(leaf2);
30
31 console.log(composite.operation()); // "Leaf, Leaf"
```

# Décorateur (*Decorator*)

- **Problème** : Attacher dynamiquement des responsabilités supplémentaires à un objet sans modifier sa structure.
- **Solution** : Fournir une alternative flexible à l'héritage pour étendre les fonctionnalités.
- **Métaphore** : Ajouter des garnitures à une pizza de base pour en modifier le goût.

```
1  interface Component {
2      operation(): string;
3  }
4
5  class ConcreteComponent implements Component {
6      operation(): string {
7          return "ConcreteComponent";
8      }
9  }
10
11 class Decorator implements Component {
12     protected component: Component;
13
14     constructor(component: Component) {
15         this.component = component;
16     }
17
18     operation(): string {
19         return this.component.operation();
20     }
21 }
22
23 class ConcreteDecoratorA extends Decorator {
24     operation(): string {
25         return `ConcreteDecoratorA(${super.operation()})`;
26     }
27 }
28
29 class ConcreteDecoratorB extends Decorator {
30     operation(): string {
31         return `ConcreteDecoratorB(${super.operation()})`;
32     }
33 }
34
35 // Utilisation
36 const simple = new ConcreteComponent();
37 const decoratedA = new ConcreteDecoratorA(simple);
38 const decoratedB = new ConcreteDecoratorB(decoratedA);
39
40 console.log(decoratedB.operation());
41 // Output: "ConcreteDecoratorB(ConcreteDecoratorA(ConcreteComponent))"
```

# Façade (*Facade*)

- **Problème** : Fournir une interface unifiée à un ensemble d'interfaces dans un sous-système.
- **Solution** : Simplifier l'utilisation du sous-système.
- **Métaphore** : Le tableau de bord d'une voiture regroupant les commandes du véhicule.

```
● ● ●

1  class Subsystem1 {
2      operation1(): string {
3          return "Subsystem1: Ready!";
4      }
5
6      operationN(): string {
7          return "Subsystem1: Go!";
8      }
9  }
10
11 class Subsystem2 {
12     operation1(): string {
13         return "Subsystem2: Get ready!";
14     }
15
16     operationZ(): string {
17         return "Subsystem2: Fire!";
18     }
19 }
20
21 class Facade {
22     private subsystem1: Subsystem1;
23     private subsystem2: Subsystem2;
24
25     constructor(subsystem1: Subsystem1, subsystem2: Subsystem2) {
26         this.subsystem1 = subsystem1;
27         this.subsystem2 = subsystem2;
28     }
29
30     operation(): string {
31         return `
32             ${this.subsystem1.operation1()}
33             ${this.subsystem2.operation1()}
34             ${this.subsystem1.operationN()}
35             ${this.subsystem2.operationZ()}
36         `;
37     }
38 }
39
40 // Utilisation
41 const subsystem1 = new Subsystem1();
42 const subsystem2 = new Subsystem2();
43 const facade = new Facade(subsystem1, subsystem2);
44
45 console.log(facade.operation());
```

# Poids Mouche (*Flyweight*)

- **Problème** : Réduire le coût de création et de manipulation d'un grand nombre d'objets similaires.
- **Solution** : Partager autant de données que possible entre les objets.
- **Métaphore** : Une bibliothèque où plusieurs livres partagent le même contenu avec des couvertures différentes.

```
1  class Flyweight {
2      private sharedState: string;
3
4      constructor(sharedState: string) {
5          this.sharedState = sharedState;
6      }
7
8      operation(uniqueState: string): void {
9          console.log(
10             `Flyweight: Shared state: ${this.sharedState}, Unique state: ${uniqueState}`
11         );
12     }
13 }
14
15 class FlyweightFactory {
16     private flyweights: { [key: string]: Flyweight } = {};
17
18     getFlyweight(sharedState: string): Flyweight {
19         if (!this.flyweights[sharedState]) {
20             this.flyweights[sharedState] = new Flyweight(sharedState);
21         }
22         return this.flyweights[sharedState];
23     }
24 }
25
26 // Utilisation
27 const factory = new FlyweightFactory();
28 const flyweight1 = factory.getFlyweight("State1");
29 const flyweight2 = factory.getFlyweight("State1");
30
31 flyweight1.operation("Unique1");
32 flyweight2.operation("Unique2");
33
34 console.log(flyweight1 === flyweight2); // true
```

# Proxy

- **Problème** : Fournir un substitut ou un placeholder pour un autre objet afin de contrôler l'accès à celui-ci.
- **Solution** : Contrôler l'accès à l'objet réel.
- **Métaphore** : Un assistant personnel gérant l'agenda et les rendez-vous d'une personne importante.

```
● ● ●
1 interface Subject {
2     request(): void;
3 }
4
5 class RealSubject implements Subject {
6     request(): void {
7         console.log("RealSubject: Handling request.");
8     }
9 }
10
11 class ProxySubject implements Subject {
12     private realSubject: RealSubject;
13
14     constructor(realSubject: RealSubject) {
15         this.realSubject = realSubject;
16     }
17
18     request(): void {
19         if (this.checkAccess()) {
20             this.realSubject.request();
21             this.logAccess();
22         }
23     }
24
25     private checkAccess(): boolean {
26         console.log("Proxy: Checking access...");
27         return true;
28     }
29
30     private logAccess(): void {
31         console.log("Proxy: Logging access.");
32     }
33 }
34
35 // Utilisation
36 const realSubject = new RealSubject();
37 const proxy = new ProxySubject(realSubject);
38
39 proxy.request();
```

# Patrons Comportementaux

# Chaîne de Responsabilité (*Chain of Responsibility*)

- **Problème** : Éviter de coupler l'expéditeur d'une requête à son destinataire en donnant à plusieurs objets la possibilité de traiter la requête.
- **Solution** : Transmettre la requête le long d'une chaîne de gestionnaires.
- **Métaphore** : Une chaîne de montage où chaque ouvrier vérifie si une tâche lui incombe.

```
1 abstract class Handler {
2     private nextHandler: Handler | null = null;
3
4     setNext(handler: Handler): Handler {
5         this.nextHandler = handler;
6         return handler;
7     }
8
9     handle(request: string): string | null {
10        if (this.nextHandler) {
11            return this.nextHandler.handle(request);
12        }
13        return null;
14    }
15 }
16
17 class ConcreteHandlerA extends Handler {
18     handle(request: string): string | null {
19        if (request === "A") {
20            return `Handled by Handler A`;
21        }
22        return super.handle(request);
23    }
24 }
25
26 class ConcreteHandlerB extends Handler {
27     handle(request: string): string | null {
28        if (request === "B") {
29            return `Handled by Handler B`;
30        }
31        return super.handle(request);
32    }
33 }
34
35 // Utilisation
36 const handlerA = new ConcreteHandlerA();
37 const handlerB = new ConcreteHandlerB();
38
39 handlerA.setNext(handlerB);
40
41 console.log(handlerA.handle("A")); // Handled by Handler A
42 console.log(handlerA.handle("B")); // Handled by Handler B
```

# Commande (*Command*)

- **Problème** : Encapsuler une requête en tant qu'objet, permettant ainsi de paramétriser les clients avec des files d'attente, des requêtes et des opérations.
- **Solution** : Encapsuler une requête en tant qu'objet.
- **Métaphore** : Une télécommande enregistrant des commandes (allumer, éteindre, changer de chaîne) exécutables ultérieurement.

```
1 interface Command {
2     execute(): void;
3 }
4
5 class Light {
6     turnOn(): void {
7         console.log("Light is ON");
8     }
9
10    turnOff(): void {
11        console.log("Light is OFF");
12    }
13 }
14
15 class TurnOnCommand implements Command {
16     private light: Light;
17
18     constructor(light: Light) {
19         this.light = light;
20     }
21
22     execute(): void {
23         this.light.turnOn();
24     }
25 }
26
27 class TurnOffCommand implements Command {
28     private light: Light;
29
30     constructor(light: Light) {
31         this.light = light;
32     }
33
34     execute(): void {
35         this.light.turnOff();
36     }
37 }
38
39 // Invoker
40 class RemoteControl {
41     private command: Command;
42
43     setCommand(command: Command): void {
44         this.command = command;
45     }
46
47     pressButton(): void {
48         this.command.execute();
49     }
50 }
51
52 // Utilisation
53 const light = new Light();
54 const turnOn = new TurnOnCommand(light);
55 const turnOff = new TurnOffCommand(light);
56
57 const remote = new RemoteControl();
58 remote.setCommand(turnOn);
59 remote.pressButton();
60
61 remote.setCommand(turnOff);
62 remote.pressButton();
```

# Interprète (*Interpreter*)

- **Problème** : Définir une représentation grammaticale pour un langage et un interprète pour interpréter la grammaire.
- **Solution** : Utiliser une classe pour représenter chaque règle grammaticale.
- **Métaphore** : Un traducteur interprétant des phrases d'une langue à une autre en suivant des règles grammaticales.

```
● ● ●
1 // Interface pour tous les expressions
2 interface Expression {
3     interpret(): number;
4 }
5
6 // Expression pour les nombres
7 class NumberExpression implements Expression {
8     private value: number;
9
10    constructor(value: number) {
11        this.value = value;
12    }
13
14    interpret(): number {
15        return this.value;
16    }
17 }
18
19 // Expression pour l'addition
20 class AddExpression implements Expression {
21     private left: Expression;
22     private right: Expression;
23
24    constructor(left: Expression, right: Expression) {
25        this.left = left;
26        this.right = right;
27    }
28
29    interpret(): number {
30        return this.left.interpret() + this.right.interpret();
31    }
32 }
33
34 // Expression pour la soustraction
35 class SubtractExpression implements Expression {
36     private left: Expression;
37     private right: Expression;
38
39    constructor(left: Expression, right: Expression) {
40        this.left = left;
41        this.right = right;
42    }
43
44    interpret(): number {
45        return this.left.interpret() - this.right.interpret();
46    }
47 }
48
49 // Utilisation : Construire et interpréter l'expression  $(2 + 3) - 1$ 
50 const number1 = new NumberExpression(2);
51 const number2 = new NumberExpression(3);
52 const number3 = new NumberExpression(1);
53
54 const addition = new AddExpression(number1, number2); //  $2 + 3$ 
55 const subtraction = new SubtractExpression(addition, number3); //  $(2 + 3) - 1$ 
56
57 console.log(subtraction.interpret()); // Résultat : 4
```

# Itérateur (*Iterator*)

- **Problème** : Fournir un moyen d'accéder séquentiellement aux éléments d'un agrégat sans exposer sa représentation sous-jacente.
- **Solution** : Utiliser un objet pour traverser une collection sans exposer sa structure interne.
- **Métaphore** : Une télécommande permettant de parcourir les chaînes de télévision sans connaître leur disposition interne.

```
● ● ●
1 // Interface pour l'itérateur
2 interface Iterator<T> {
3     next(): T | null;
4     hasNext(): boolean;
5 }
6
7 // Interface pour la collection
8 interface IterableCollection<T> {
9     createIterator(): Iterator<T>;
10 }
11
12 // Collection concrète
13 class NumberCollection implements IterableCollection<number> {
14     private items: number[] = [];
15
16     addItem(item: number): void {
17         this.items.push(item);
18     }
19
20     createIterator(): Iterator<number> {
21         return new NumberIterator(this);
22     }
23
24     getItems(): number[] {
25         return this.items;
26     }
27 }
28
29 // Itérateur concret
30 class NumberIterator implements Iterator<number> {
31     private collection: NumberCollection;
32     private index: number = 0;
33
34     constructor(collection: NumberCollection) {
35         this.collection = collection;
36     }
37
38     next(): number | null {
39         if (this.hasNext()) {
40             return this.collection.getItems()[this.index++];
41         }
42         return null;
43     }
44
45     hasNext(): boolean {
46         return this.index < this.collection.getItems().length;
47     }
48 }
49
50 // Utilisation
51 const collection = new NumberCollection();
52 collection.addItem(1);
53 collection.addItem(2);
54 collection.addItem(3);
55
56 const iterator = collection.createIterator();
57 while (iterator.hasNext()) {
58     console.log(iterator.next());
59 }
60 // Output: 1, 2, 3
```

# Médiateur (*Mediator*)

- **Problème** : Définir un objet qui encapsule la manière dont un ensemble d'objets interagit, favorisant un couplage lâche en empêchant les objets de se référer explicitement les uns aux autres.
- **Solution** : Utiliser un médiateur pour centraliser la communication entre les objets.
- **Métaphore** : Une tour de contrôle coordonnant les atterrissages et décollages des avions.

```
 1 // Interface du médiateur
 2 interface Mediator {
 3     send(message: string, colleague: Colleague): void;
 4 }
 5
 6 // Colleague abstrait
 7 abstract class Colleague {
 8     protected mediator: Mediator;
 9
10    constructor(mediator: Mediator) {
11        this.mediator = mediator;
12    }
13
14    abstract receive(message: string): void;
15 }
16
17 // Colleague concret
18 class User extends Colleague {
19     private name: string;
20
21    constructor(name: string, mediator: Mediator) {
22        super(mediator);
23        this.name = name;
24    }
25
26    send(message: string): void {
27        console.log(` ${this.name} sends: ${message}`);
28        this.mediator.send(message, this);
29    }
30
31    receive(message: string): void {
32        console.log(` ${this.name} receives: ${message}`);
33    }
34 }
35
36 // Médiateur concret
37 class ChatRoom implements Mediator {
38     private users: User[] = [];
39
40    addUser(user: User): void {
41        this.users.push(user);
42    }
43
44    send(message: string, sender: User): void {
45        for (const user of this.users) {
46            if (user !== sender) {
47                user.receive(message);
48            }
49        }
50    }
51
52    // Utilisation
53    const chatRoom = new ChatRoom();
54
55    const user1 = new User("Alice", chatRoom);
56    const user2 = new User("Bob", chatRoom);
57    const user3 = new User("Charlie", chatRoom);
58
59    chatRoom.addUser(user1);
60    chatRoom.addUser(user2);
61    chatRoom.addUser(user3);
62
63    user1.send("Hello everyone!");
64    user2.send("Hi Alice!");
```

# Memento (*Memento*)

- **Problème** : Capturer et externaliser l'état interne d'un objet sans violer son encapsulation.
- **Solution** : Permettre à l'objet de restaurer un état précédent.
- **Métaphore** : Un marque-page dans un livre pour revenir à une position précise.

```
● ● ●
1 // Memento
2 class Memento {
3     private state: string;
4
5     constructor(state: string) {
6         this.state = state;
7     }
8
9     getState(): string {
10    return this.state;
11 }
12 }
13
14 // Originator
15 class TextEditor {
16     private content: string = "";
17
18     setContent(content: string): void {
19         this.content = content;
20     }
21
22     getContent(): string {
23         return this.content;
24     }
25
26     save(): Memento {
27         return new Memento(this.content);
28     }
29
30     restore(memento: Memento): void {
31         this.content = memento.getState();
32     }
33 }
34
35 // Caretaker
36 class History {
37     private mementos: Memento[] = [];
38
39     addMemento(memento: Memento): void {
40         this.mementos.push(memento);
41     }
42
43     getMemento(index: number): Memento {
44         return this.mementos[index];
45     }
46 }
47
48 // Utilisation
49 const editor = new TextEditor();
50 const history = new History();
51
52 editor.setContent("Version 1");
53 history.addMemento(editor.save());
54
55 editor.setContent("Version 2");
56 history.addMemento(editor.save());
57
58 editor.setContent("Version 3");
59
60 console.log("Current content:", editor.getContent()); // Version 3
61
62 editor.restore(history.getMemento(0));
63 console.log("Restored to:", editor.getContent()); // Version 1
```

# Observer (Observateur)

- **Problème** : Comment définir une dépendance un-à-plusieurs pour notifier automatiquement les objets lorsque l'état change ?
- **Solution** : Les objets observés (subject) notifient les observateurs (observers) lorsque leur état change.
- **Utilisation courante** : Interfaces graphiques (GUI), abonnements aux événements.

**Métaphore :**

- Un abonnement à un journal : lorsqu'un nouveau numéro est publié, tous les abonnés le reçoivent.

```
● ● ●

1 interface Observer {
2     update(state: string): void;
3 }
4
5 class Subject {
6     private observers: Observer[] = [];
7     private state: string;
8
9     attach(observer: Observer): void {
10     this.observers.push(observer);
11 }
12
13 detach(observer: Observer): void {
14     this.observers = this.observers.filter(o => o !== observer);
15 }
16
17 notify(): void {
18     for (const observer of this.observers) {
19         observer.update(this.state);
20     }
21 }
22
23 setState(state: string): void {
24     this.state = state;
25     this.notify();
26 }
27 }
28
29 class ConcreteObserver implements Observer {
30     update(state: string): void {
31         console.log(`Observer received new state: ${state}`);
32     }
33 }
34
35 // Utilisation
36 const subject = new Subject();
37 const observer1 = new ConcreteObserver();
38 const observer2 = new ConcreteObserver();
39
40 subject.attach(observer1);
41 subject.attach(observer2);
42
43 subject.setState("State 1");
44 subject.setState("State 2");
```

# Strategy (Stratégie)

- **Problème** : Comment définir une famille d'algorithmes interchangeables et permettre leur substitution au moment de l'exécution ?
- **Solution** : Encapsuler chaque algorithme dans une classe distincte et les rendre interchangeables.
- **Utilisation courante** : Calculs, tri, sélection de stratégies en fonction du contexte.

**Métaphore :**

- Différents modes de transport : en voiture, à vélo ou à pied. Vous choisissez la stratégie en fonction de vos besoins

```
● ● ●

1  interface Strategy {
2      execute(a: number, b: number): number;
3  }
4
5  class AddStrategy implements Strategy {
6      execute(a: number, b: number): number {
7          return a + b;
8      }
9  }
10
11 class MultiplyStrategy implements Strategy {
12     execute(a: number, b: number): number {
13         return a * b;
14     }
15 }
16
17 class Context {
18     private strategy: Strategy;
19
20     setStrategy(strategy: Strategy): void {
21         this.strategy = strategy;
22     }
23
24     executeStrategy(a: number, b: number): number {
25         return this.strategy.execute(a, b);
26     }
27 }
28
29 // Utilisation
30 const context = new Context();
31
32 context.setStrategy(new AddStrategy());
33 console.log("Add: ", context.executeStrategy(3, 4)); // Add: 7
34
35 context.setStrategy(new MultiplyStrategy());
36 console.log("Multiply: ", context.executeStrategy(3, 4)); // Multiply: 12
```

# State (État)

- **Problème** : Comment permettre à un objet de modifier son comportement lorsqu'il change d'état interne ?
- **Solution** : Encapsuler les états dans des classes distinctes et déléguer à ces classes le comportement correspondant.
- **Utilisation courante** : Automates finis, machines à états (state machines).

## Métaphore :

- Un lecteur multimédia : les actions disponibles (lecture, pause, stop) dépendent de l'état actuel.

```
● ● ●

1  interface State {
2      handle(): void;
3  }
4
5  class Context {
6      private state: State;
7
8      setState(state: State): void {
9          this.state = state;
10     }
11
12     request(): void {
13         this.state.handle();
14     }
15 }
16
17 class ConcreteStateA implements State {
18     handle(): void {
19         console.log("Handling in State A");
20     }
21 }
22
23 class ConcreteStateB implements State {
24     handle(): void {
25         console.log("Handling in State B");
26     }
27 }
28
29 // Utilisation
30 const context = new Context();
31
32 context.setState(new ConcreteStateA());
33 context.request();
34
35 context.setState(new ConcreteStateB());
36 context.request();
```

# Template Method (Méthode Template)

- **Problème** : Comment définir le squelette d'un algorithme tout en laissant certaines étapes aux sous-classes ?
- **Solution** : Implémenter les étapes communes dans une classe parent et permettre aux sous-classes de personnaliser les étapes spécifiques.
- **Utilisation courante** : Génération de rapports, gestion de pipelines de traitement.

**Métaphore :**

- Préparer une tasse de thé : l'ordre général est fixé (faire bouillir l'eau, infuser, ajouter des ingrédients), mais les détails varient

```
● ● ●
```

```
1 abstract class Template {
2     finalOperation(): void {
3         this.step1();
4         this.step2();
5         this.step3();
6     }
7
8     protected abstract step1(): void;
9
10    protected abstract step2(): void;
11
12    protected step3(): void {
13        console.log("Common step 3");
14    }
15}
16
17 class ConcreteTemplateA extends Template {
18     protected step1(): void {
19         console.log("ConcreteTemplateA step 1");
20     }
21
22     protected step2(): void {
23         console.log("ConcreteTemplateA step 2");
24     }
25}
26
27 class ConcreteTemplateB extends Template {
28     protected step1(): void {
29         console.log("ConcreteTemplateB step 1");
30     }
31
32     protected step2(): void {
33         console.log("ConcreteTemplateB step 2");
34     }
35}
36
37 // Utilisation
38 const templateA = new ConcreteTemplateA();
39 templateA.finalOperation();
40
41 const templateB = new ConcreteTemplateB();
42 templateB.finalOperation();
```

# Visitor (Visiteur)

- **Problème** : Comment ajouter des opérations à une hiérarchie d'objets sans modifier leurs classes ?
- **Solution** : Séparer les opérations des objets en passant par un visiteur (visitor) qui implémente les opérations nécessaires.
- **Utilisation courante** : Traverser des structures complexes (arbres, graphiques) pour exécuter des opérations spécifiques.

**Métaphore :**

- Un inspecteur dans une usine : il vérifie différentes parties (machines, chaînes de montage) sans modifier leur fonctionnement interne

```
● ● ●

1 interface Visitor {
2     visitConcreteElementA(element: ConcreteElementA): void;
3     visitConcreteElementB(element: ConcreteElementB): void;
4 }
5
6 interface Element {
7     accept(visitor: Visitor): void;
8 }
9
10 class ConcreteElementA implements Element {
11     accept(visitor: Visitor): void {
12         visitor.visitConcreteElementA(this);
13     }
14
15     operationA(): string {
16         return "ConcreteElementA operation";
17     }
18 }
19
20 class ConcreteElementB implements Element {
21     accept(visitor: Visitor): void {
22         visitor.visitConcreteElementB(this);
23     }
24
25     operationB(): string {
26         return "ConcreteElementB operation";
27     }
28 }
29
30 class ConcreteVisitor implements Visitor {
31     visitConcreteElementA(element: ConcreteElementA): void {
32         console.log(`Visitor operating on ${element.operationA()}`);
33     }
34
35     visitConcreteElementB(element: ConcreteElementB): void {
36         console.log(`Visitor operating on ${element.operationB()}`);
37     }
38 }
39
40 // Utilisation
41 const elements: Element[] = [new ConcreteElementA(), new ConcreteElementB()];
42 const visitor = new ConcreteVisitor();
43
44 for (const element of elements) {
45     element.accept(visitor);
46 }
```

# Fin du cours

- Merci pour votre attention !
- Questions ou remarques ?