JS 2019 Day CANARIAS

Patrones de diseño con typescript en el mundo real

@ivanirega & @cbastospc

@cbastospc

Carlos Bastos

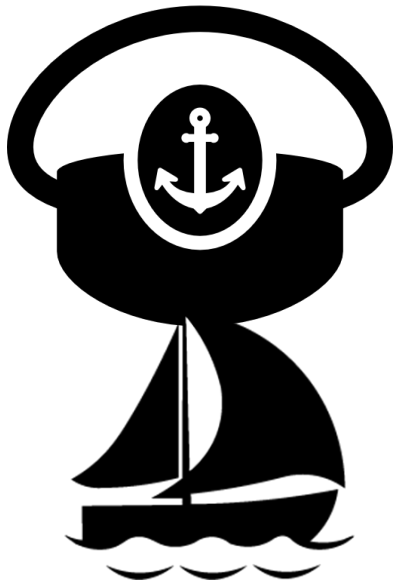
@ivanirega

Iván Reinoso

¿Qué es un patrón?

A)

B)



C)

Code

thermomix

¿Qué es un patrón?

Wikipedia:

*Los patrones de diseño son unas **técnicas para resolver problemas comunes en el desarrollo de software** y otros ámbitos referentes al diseño de interacción o interfaces.*

*Un patrón de diseño resulta ser una **solución a un problema de diseño**. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que **debe haber comprobado su efectividad** resolviendo problemas similares en ocasiones anteriores. Otra es que **debe ser reutilizable**, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.*

Factory method teoría

```
interface Product {
  // Contract members
}
```

implements

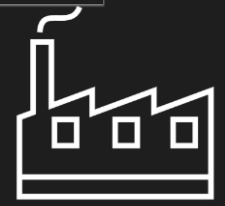
```
class ConcreteProduct1 implements Product {
  // Contract members implemented
}
```

creates

```
interface Creator {
  factoryMethod(): Product;
}
```

implements

```
class ConcreteCreator implements Creator {
  factoryMethod(): Product {
    return new ConcreteProduct1();
  }
}
```



Factory method realidad

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

implements

```ts
class AppInsightsLogger implements Logger {
  log(message: string): void {
    (<any>window).appInsights.trackTrace({ message });
  }
}
```

creates

```ts
abstract class AbstractLoggerCreator {
  protected abstract createLogger(): Logger;

  public getLogger(): Logger {
    return this.createLogger();
  }
}
```

extends

extends

```ts
class AppInsightsLoggerCreator
  extends AbstractLoggerCreator {

  protected createLogger(): Logger {
    return new AppInsightsLogger();
  }
}
```

```ts
class ConsoleLoggerCreator
  extends AbstractLoggerCreator {

  protected createLogger(): Logger {
    return console;
  }
}
```

Usage:

```ts
const consoleLoggerCreator: AbstractLoggerCreator = new ConsoleLoggerCreator();
consoleLoggerCreator.getLogger().log('a message shown in the browser console');

const appInsightsLoggerCreator: AbstractLoggerCreator = new AppInsightsLoggerCreator();
appInsightsLoggerCreator.getLogger().log('a message tracked in the cloud (Azure AppInsights)');
```

Factory method (with params) teoría

```ts
interface Product {
  // Contract members
}
```

implements

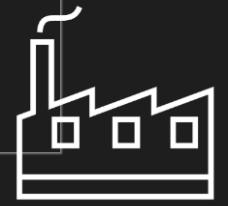
```ts
class ConcreteProduct1 implements Product {
  // Contract members implemented
}

class ConcreteProduct2 implements Product {
  // Contract members implemented
}

class ConcreteProduct3 implements Product {
  // Contract members implemented
}
```

creates

```ts
interface Creator {
  factoryMethod(param: 1 | 2 | 3): Product;
}
```

implements

```ts
class ConcreteCreator implements Creator {
  factoryMethod(param: 1 | 2 | 3): Product {
    const dictionary = {
      1: ConcreteProduct1,
      2: ConcreteProduct2,
      3: ConcreteProduct3
    }
    const ConcreteProduct = dictionary[param];
    return new ConcreteProduct();
  }
}
```



Factory method (with params) realidad

```
@Component({ selector: 'fm-text-component', templateUrl: './text.component.html' })
export class FactoryMethodTextComponent implements IComponent {
    @Input() data: Array<any>;
}

@Component({ selector: 'fm-textarea-component', templateUrl: './textarea.component.html' })
export class FactoryMethodTextAreaComponent implements IComponent {
    @Input() data: Array<any>;
}

@Component({ selector: 'fm-select-component', templateUrl: './select.component.html' })
export class FactoryMethodSelectComponent implements IComponent {
    @Input() data: Array<any>;
}
```

implements

```
export interface IComponent {
    data: Array<any>
}
```

creates

```
export class ComponentFactory {
    static getComponentByMode(mode: ComponentModes) {
        const dictionary = {
            [ComponentModes.Text]: FactoryMethodTextComponent,
            [ComponentModes.TextArea]: FactoryMethodTextAreaComponent,
            [ComponentModes.Select]: FactoryMethodSelectComponent
        };
        return dictionary[mode];
    }
}
```

gets

```
@Component({ selector: 'fm-creator-component', template: '' })
export class FactoryMethodCreatorComponent implements OnInit {
    @Input() mode: ComponentModes;
    @Input() data: Array<any>;

    constructor(
        private componentFactoryResolver: ComponentFactoryResolver,
        public viewContainerRef: ViewContainerRef
    ) { }

    ngOnInit() {
        const componentToInstance = ComponentFactory.getComponentByMode(this.mode);
        const componentFactory =
            this.componentFactoryResolver.resolveComponentFactory(componentToInstance);

        this.viewContainerRef.clear();

        const componentInstance =
            this.viewContainerRef.createComponent(componentFactory).instance;
        componentInstance.data = this.data;
    }
}
```

Usage:

```
<fm-creator-component [mode]="textMode" [data]="genericData"></fm-creator-component>
<fm-creator-component [mode]="textAreaMode" [data]="genericData"></fm-creator-component>
<fm-creator-component [mode]="selectMode" [data]="genericData"></fm-creator-component>

@Component({
    selector: 'factory-method-with-params',
    templateUrl: './real-world.component.html'
})
export class FactoryMethodRealWorldComponent {
    textMode = ComponentModes.Text;
    textAreaMode = ComponentModes.TextArea;
    selectMode = ComponentModes.Select;
    genericData = ['first item', 'second item'];
}
```

Singleton teoría

```
class Singleton {
  private static instance: Singleton;
  private constructor() { }
  static build(): Singleton {
    Singleton.instance = Singleton.instance || new Singleton();
    return Singleton.instance;
  }
}
```

creates

```
const instance = Singleton.build();
```

Singleton realidad

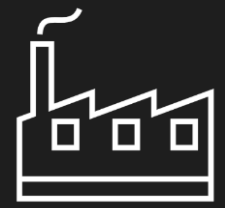
Usage:

```
export class Client {
    initializeApp(): void {
        ReportTemplatesAgentFactory.use(
            'https://my-base-url', 'my-app-id');
    }

    async generateReports(): Promise<void> {
        const reportTemplates =
            await ReportTemplatesAgentFactory
                .build().getReporTemplates();
        // Apply datasource over the report templates.
    }
}
```

```
class ReportTemplatesAgentFactory {
    private static instance: ReportTemplatesAgent;
    private static config: { baseUrl: string, applicationId: string };

    static use(baseUrl: string, applicationId: string) {
        this.config = { baseUrl, applicationId };
    }

    static build(): ReportTemplatesAgent {
        if (!this.config) {
            throw new Error('You should configure the factory before using it.')
        }
        this.instance = this.instance || new ReportTemplatesAgent(
            this.config.baseUrl,
            this.config.applicationId
        );
        return this.instance;
    }
}
```

creates

```
class ReportTemplatesAgent {
    constructor(
        private baseUrl: string,
        private applicationId: string
    ) { }

    getReporTemplates(): Promise<Array<ReportTemplate>> {
        return fetch(
            new Request('https://my-reports-api.com/get-all-report-templates')
        ).then(response => response.json());
    }
}
```



Adapter teoría

```
interface Target {
    requiredMethod(): void;
}
```

inherits

inherits

```
class Adapter implements Target {
    constructor(private adapteeObject: Adaptee) { }

    requiredMethod(): void {
        this.adapteeObject.oldMethod();
    }
}
```

```
class CompatibleObject implements Target {
    requiredMethod(): void { }
}
```

adapts

```
class Adaptee {
    oldMethod(): void { }
}
```

Usage:

```
export class Client {
    static main(): void {
        const a_compatible_object = new CompatibleObject();
        const a_non_compatible_object = new Adaptee();
        const adapter = new Adapter(a_non_compatible_object);

        a_compatible_object.requiredMethod();
        adapter.requiredMethod();
    }
}
```

Adapter realidad

```
<i [tooltip]="text" (events)="handleTooltipEvents($event)" class="material-icons">info</i>

@Component({
    selector: 'fa-tooltip',
    templateUrl: './tooltip-facade.component.html'
})
export class TooltipFacadeComponent {
    @Input() text: string;
    @Output() shown: EventEmitter<void> = new EventEmitter<void>();

    handleTooltipEvents(event: { type: string, position: DOMRect }) {
        if (event.type === 'shown') {
            this.shown.emit();
        }
    }
}
```

calls

Usage:

```
<span>One text</span>
<fa-tooltip text="one tooltip" (shown)="tooltipWasShown()"></fa-tooltip>
<span>Another text</span>
<fa-tooltip text="another tooltip" (shown)="tooltipWasShown()"></fa-tooltip>
```

Facade teoría

```ts
export class Client {
    callOperations(): void {
        const facade = new Facade();
        facade.operation1();
        facade.operation2();
    }
}
```

calls

```ts
class Facade {
    private part1: Part1 = new Part1();
    private part2: Part2 = new Part2();
    private part3: Part3 = new Part3();

    operation1(): void {
        this.part1.method1();
        this.part2.method2();
    }

    operation2(): void {
        this.part1.method1();
        this.part3.method3();
    }
}
```

calls

```ts
class Part1 {
    method1(): void {
        console.log("method 1 (Part 1)");
    }
}
```

calls

```ts
class Part2 {
    method2(): void {
        console.log("method 2 (Part 2)");
    }
}
```

calls

```ts
class Part3 {
    method3(): void {
        console.log("method 3 (Part 3)");
    }
}
```

Facade realidad

```ts
export class Client {
    private agent = new ReportsAgentFacade();

    async showAvailableReports(): Promise<void> {
        const reports = await this.agent.loadReports();
        // Show the available reports in a dropdown
    }

    async printReport(reportId: number): Promise<void> {
        const report = await this.agent.getReport(reportId);
        document.body.innerHTML = report.template;
    }
}
```

calls

```ts
export class ReportsAgentFacade {
    private reportsService = new ReportsService();
    private assetsService = new AssetsService();
    private appModeService = new AppModeService();

    async loadReports(): Promise<IReport[]> {
        let reports: IReport[] = [];
        if (this.appModeService.isOffline()) {
            reports = await this.assetsService.getJsonFromFile('reports');
            localStorage.setItem('reports', JSON.stringify(reports));
        } else {
            reports = await this.reportsService.getAll();
        }
        return reports;
    }

    async getReport(reportId: number): Promise<IReport> {
        let reports: IReport[] = [];
        if (this.appModeService.isOffline()) {
            reports = JSON.parse(localStorage.getItem('reports'));
        } else {
            reports = await this.reportsService.getAll();
        }
        return reports.find(r => r.id === reportId);
    }
}
```

calls

```ts
export class ReportsService {
    async getAll(): Promise<IReport[]> {
        const reports = await fetch(
            new Request('http://my-reports-rest-api/reports'))
            .then(response => response.json());

        return reports.map((report, index) =>
            ({
                id: index,
                template: 'a-html-report-template',
                name: report.name
            }));
```

calls

```ts
export class AssetsService {
    getJsonFromFile(fileName: string): Promise<any> {
        return fetch(new Request(`./my-assets-path/${fileName}.json`))
            .then(response => response.json());
    }
}
```

calls

```ts
export class AppModeService {
    isOffline(): boolean {
        return !!window.navigator.userAgent.match(/Electron/);
    }
}
```

Flyweight teoría

```
interface Flyweight {
    operation(extrinsicState);
}
```

implements

```
class ConcreteFlyweight implements Flyweight {
    operation(extrinsicState) { }
}
```

creates and stores

```
class FlyweightFactory {
    private flyweights: { [key: string]: Flyweight };

    GetFlyWeight(intrinsicState): Flyweight {
        const key = this.getKey(intrinsicState);
        if (!this.flyweights[key]) {
            const newFlyweight = new ConcreteFlyweight();
            this.flyweights[key] = newFlyweight;
            return newFlyweight;
        } else {
            return this.flyweights[key];
        }
    }

    private getKey(intrinsicState): string {
        return `${intrinsicState.model}-${intrinsicState.color}`;
    }
}
```

Usage:

```
class Client {
    run() {
        const factory = new FlyweightFactory();
        const car1 = factory.GetFlyWeight({ model: 'BMW', color: 'red' });
        const car2 = factory.GetFlyWeight({ model: 'BMW', color: 'red' });
        const car3 = factory.GetFlyWeight({ model: 'Toyota', color: 'white' });
        car1.operation({ km: 1000 });
        car2.operation({ km: 200 });
        car3.operation({ km: 3500 });
    }
}
```

Flyweight realidad

```
<h3>Original implementation:</h3>
<fly-original-info infoText="first info"></fly-original-info>
<fly-original-info infoText="second info"></fly-original-info>
```

has 1..n

```
<button (click)="showDialog()">Show info</button>
<fly-dialog #dialog></fly-dialog>
```

```
@Component({
    selector: 'fly-dialog',
    templateUrl: 'dialog.component.html',
    styleUrls: ['./dialog.component.css']
})
export class FlyweightDialogComponent {
    text = '';
    visible = false;
    show(text: string) { this.text = text; this.visible = true; }
    close() { this.visible = false; }
}
```

Extracting intrinsic (the dialog) and extrinsic (the info text) properties:

```
<h3>Flyweight implementation:</h3>
<fly-refactored-info infoText="first info" (showDialog)=showFlyweightDialog($event)></fly-refactored-info>
<fly-refactored-info infoText="second info" (showDialog)=showFlyweightDialog($event)></fly-refactored-info>
<fly-dialog #dialog></fly-dialog>
```

```
@Component({
    selector: 'flyweight',
    templateUrl: './real-world.component.html'
})
export class FlyweightRealWorldComponent {
    @ViewChild('dialog', { static: false }) dialog: FlyweightDialogComponent;
    showFlyweightDialog(extrinsicInfoText: string) {
        this.dialog.show(extrinsicInfoText);
    }
}
```

has

```
@Component({
    selector: 'fly-dialog',
    templateUrl: 'dialog.component.html',
    styleUrls: ['./dialog.component.css']
})
export class FlyweightDialogComponent {
    text = '';
    visible = false;
    show(text: string) { this.text = text; this.visible =
    close() { this.visible = false; }
}
```

Strategy teoría

```ts
interface Strategy {
    strategicMethod(): void;
}
```

implements

implements

```ts
class ConcreteStrategyB implements Strategy {
    public strategicMethod(): void {
        console.log('Strategy B');
    }
}
```

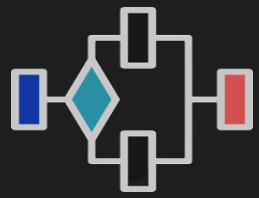
```ts
class ConcreteStrategyA implements Strategy {
    public strategicMethod(): void {
console.log('Strategy A');
    }
}
```

executes

```ts
class Context {
    private strategy: Strategy;

    constructor(strategy: Strategy) {
        this.strategy = strategy;
    }

    setStrategy(strategy: Strategy) {
        this.strategy = strategy;
    }

    performStrategicMethod(): void {
        this.strategy.strategicMethod();
    }
}
```

Usage:

```ts
export class Client {
    run(): void {
        const context = new Context(new ConcreteStrategyA());
        context.performStrategicMethod();

        context.setStrategy(new ConcreteStrategyB());
        context.performStrategicMethod();
    }
}
```



Strategy realidad

implements

executes

```
export interface FieldFormatter<T> {
    format(field: Field<T>): string;
}
```

```
export class StringFormatter implements FieldFormatter<string> {

    format(field: Field<string>): string {
        return field.value;
    }
}
```

```
export class BooleanFormatter implements FieldFormatter<boolean> {

    format(field: Field<boolean>): string {
        return field.value ? 'True' : 'False';
    }
}
```

```
@Injectable()
export class FieldFormatterService {
    private formattersDictionary: { [key: number]: FieldFormatter<any> };

    constructor() {
        this.formattersDictionary =
            FieldTypeFormatsFactory.buildFormattersDictionary();
    }

    format(field: Field<any>): string {
        const typeFormatter = this.formattersDictionary[field.type];
        return typeFormatter !== undefined && typeFormatter.format(field);
    }
}
```

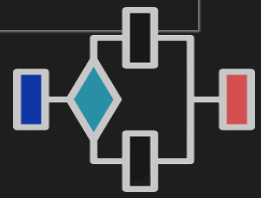
gets

```
export class FieldTypeFormatsFactory {

    static buildFormattersDictionary(
    ): { [key: number]: FieldFormatter<any> } {
        return {
            [FieldTypes.String]: new StringFormatter(),
            [FieldTypes.Boolean]: new BooleanFormatter()
        };
    }
}
```

```
<ul>
    <li *ngFor="let fieldFormatted of fieldsFormatted">{{fieldFormatted}}</li>
</ul>
```

```
@Component({
    selector: 'strategy',
    templateUrl: './real-world.component.html'
})
export class StrategyRealWorldComponent implements OnInit {
    fieldsFormatted: Array<string>;

    constructor(private fieldFormatter: FieldFormatterService) { }

    ngOnInit() {
        const fields: Array<Field<any>> = [
            { type: FieldTypes.Boolean, value: true },
            { type: FieldTypes.String, value: 'a field value' }
        ]
        this.fieldsFormatted = fields.map(f => this.fieldFormatter.format(f));
    }
}
```

Usage



Observer teoría

```ts
interface Observer {
    update(): void;
}
```

implements

```ts
class ConcreteObserver implements Observer {
    update(): void { }
}
```

notifies

registers

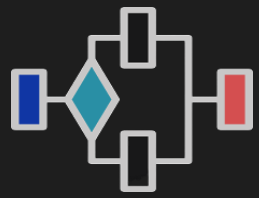
```ts
interface Subject {
    registerObserver(observer: Observer);
    unregisterObserver(observer: Observer);
    notifyObservers();
}
```

implements

```ts
class ConcreteSubject implements Subject {
    observers: Array<Observer>;

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

    unregisterObserver(observer: Observer) {
        // TODO: find and remove observer from the collection.
    }

    notifyObservers() {
        this.observers.forEach(o => o.update());
    }
}
```

Usage:

```ts
class Client {
    run() {
        const subject = new ConcreteSubject();
        const observer1 = new ConcreteObserver();
        const observer2 = new ConcreteObserver();

        subject.registerObserver(observer1);
        subject.registerObserver(observer2);

        subject.notifyObservers();
    }
}
```



Observer realidad

```
<observer1></observer1>
<observer2></observer2>

@Component({
    selector: 'observer',
    templateUrl: './real-world.component.html'
})
export class ObserverRealWorldComponent { }
```

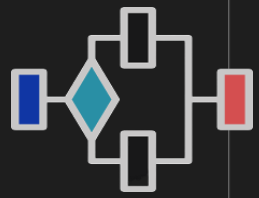
```
@Injectable()
export class ConnectorService {
    private currentState: Order;
    private observers = [];

    constructor(private store: Store) {
        this.currentState = this.store.state;
    }

    registerObserver(component) {
        this.observers.push(component);
        this.notifyObserver(component);
    }

    updateState(newState: Order) {
        if (this.isStateChanged(newState)) {
            this.currentState = newState;
            this.notifyObservers();
        }
    }

    private notifyObservers() {
        this.observers.forEach(o => this.notifyObserver(o));
    }
```

notifies

registers and updates

```
<ul>
    <li *ngFor="let productName of productNames">{{productName}}</li>
</ul>
<button (click)="addNewProduct()">Add new product</button>
```

```
@Component({
    selector: 'observer2',
    templateUrl: './observer2.component.html'
})
export class Observer2Component {
    public productNames: Array<string> = [];
    private model: Order;

    @Input() set order(orderFromState: Order) {
        this.model = orderFromState;
        this.buildFormattedOrderList();
    }
    get order() { return this.model; }

    constructor(private connector: ConnectorService) {
        this.connector.registerObserver(this);
    }

    private buildFormattedOrderList(): void {
        this.productNames = this.model.productList.map(p => p.name);
    }

    addNewProduct() {
        this.connector.updateState(
            addNewProductStateAction(this.model)
        );
    }
}
```



Page-Object teoría

```html
<ul>
    <li class="id">1</li>
    <li class="name">A product</li>
    <li><a href="#" class="view-detail">View Detail</a></li>
</ul>
<ul>
    <li class="id">2</li>
    <li class="name">Another product</li>
    <li><a href="#" class="view-detail">View Detail</a></li>
</ul>
```

reads and interacts

```typescript
class TestClient {
    private pageObject = new ProductListPageObject();

    is_a_valid_product_id(): boolean {
        const productId = this.pageObject.getProductId();
        return productId > 0;
    }

    is_a_valid_product_name(): boolean {
        const productName = this.pageObject.getProductName();
        return productName.length < 255;
    }

    is_a_product_detail_visible(): boolean {
        this.pageObject.openDetail()
        // ...
```

calls

```typescript
export class ProductListPageObject {
    getProductId(): number {
        const productId = this.getHTMLElementByClass('id')[0].textContent;
        return parseInt(productId);
    }

    getProductName(): string {
        const productName = this.getHTMLElementByClass('name')[0].textContent;
        return productName;
    }

    openDetail(): void {
        const viewDetailButton = this.getHTMLElementByClass('view-detail')[0];
        viewDetailButton.click();
    }

    private getHTMLElementByClass(className: string): Array<HTMLElement> {
        return [new HTMLElement()];
    }
}
```

Page-Object realidad

```
<label for="name"></label>
<input type="text" name="name" [value]="name" (input)="name = $event.target.value" />
<button (click)="sayHello()">Say Hello!</button>
<span id="greeting">{{greeting}}</span>
```

reads and interacts

```
describe('The page object pattern real world example', () => {
    let testable_component: TestablePageObjectRealWorldComponent;

    beforeEach(() => {
        TestBed.configureTestingModule({
            imports: [AppModule]
        }).compileComponents();
        testable_component = TestablePageObjectRealWorldComponent.build();
    });

    it('should initialize the name as empty', () => {
        expect(testable_component.is_name_empty()).toBeTruthy();
    });

    describe('when the user requests a greeting', () => {

        const one_valid_name = 'JsDay';

        beforeEach(() => {
            testable_component.set_name(one_valid_name);
            testable_component.request_greeting();
        });

        it('should show the greeting', () => {
            expect(testable_component.get_greeting()).toBe(`Hello ${one_valid_name}!`);
        });

    });

});
```

calls

```
export class TestablePageObjectRealWorldComponent {
    get instance(): PageObjectRealWorldComponent { return this.fixture.componentInstance; }

    private constructor(private fixture: ComponentFixture<PageObjectRealWorldComponent>) { }

    static build(): TestablePageObjectRealWorldComponent {
        const page_object_fixture = TestBed.createComponent(PageObjectRealWorldComponent);
        const testable_component = new TestablePageObjectRealWorldComponent(page_object_fixture);
        testable_component.fixture.detectChanges();
        return testable_component;
    }

    set_name(name: string): void {
        this.instance.name = name;
        this.fixture.detectChanges();
    }

    is_name_empty(): boolean {
        const name = this.fixture.nativeElement.querySelector('[name="name"]');
        return !name || name.value === '';
    }

    request_greeting(): void {
        const request_greeting_button = this.fixture.nativeElement.querySelector('button');
        request_greeting_button.click();
        this.fixture.detectChanges();
    }

    get_greeting(): string {
        const greeting = this.fixture.nativeElement.querySelector('#greeting');
        return !greeting ? '' : greeting.textContent;
    }
}
```

¿Preguntas?

¡Muchas gracias!

@ivanirega & @cbastospc

Y aún hay más…

Builder teoría

```
class ProductBuilder {
  private product_attr_1: string;
  private product_attr_N: string;

  public addSomeConfig(param: string): ProductBuilder {
    this.product_attr_1 = param;
    return this;
  }

  public addAnotherConfig(param: string): ProductBuilder {
    this.product_attr_N = param;
    return this;
  }

  public build(): Product {
    const product = new Product();
    product.attr_1 = this.product_attr_1;
    product.attr_N = this.product_attr_N;
    return product;
  }
}
```

creates

```
class Client_and_director {
  public static main(args: String[]): void {
    const director = new ProductBuilder()
      .addSomeConfig('some param value')
      .addAnotherConfig('another param value');

    const product = director.build();
  }
}
```

returns

```
class Product {
  attr_1: string = "";
  attr_N: string = "";
}
```

Builder realidad

```
export default class Migrations {
  private migrations = [];

  last() {
    return this.migrations[this.migrations.length - 1];
  }

  add(migration: Migration) {
    this.migrations.push(migration);
    return this;
  }

  getGreaterThan(version: string) {
    const { migrations } = this;
    const migrationOfVersionIndex = migrations.indexOf(
      migrations.filter(m => m.version === version)[0]
    );
    return migrations.slice(migrationOfVersionIndex + 1, migrations.length);
  }

  build() {
    return new Migrator(new DataStorage(), this);
  }
}
```

creates

```
const migrations = new Migrations()
  .add(new Migration_v1())
  .add(new Migration_v2());

const migrator = migrations.build();

migrator.migrateToLastVersion();
```

returns

```
export default class Migrator {
  constructor(
    private storage: DataStorage,
    private migrations: Migrations
  ) { }

  migrateToLastVersion(): void {
    const current_storage_version = this.storage.get('db_pm:version');
    if (current_storage_version) {
      let current_storage_data = this.storage.get('db_pm:data');
      if (current_storage_data) {
        this.migrations.getGreaterThan(current_storage_version).forEach((migration) => {
          current_storage_data = migration.migrate(current_storage_data);
        });
        this.storage.set('db_pm:data', current_storage_data);
        this.storage.set('db_pm:version', this.migrations.last().version);
      }
    }
  }
}
```

Prototype teoría

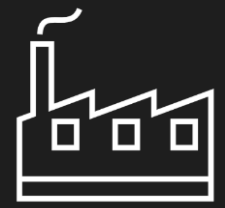
```typescript
class ClaseConcreta implements Cloneable<ClaseConcreta> {
  constructor(public attr_1: string) { }

  public clone() {
    // Member clonation
    return new ClaseConcreta(this.attr_1);
  }
}
```

implements

```typescript
interface Cloneable<T> {
  clone(): T;
}
```

Usage:

```typescript
class PrototipoClient {
  static main() {
    const p1 = new ClaseConcreta("Clone-I");
    const c1 = p1.clone();
    console.log(`Clonación: ${c1.attr_1}`);
  }
}
```



Prototype realidad

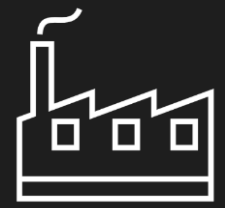
```ts
export function clone<T>(obj: T): T {
  if (obj === null || typeof obj !== 'object') {
    return obj;
  }
  const clonator = typeClonators.find(c => obj instanceof c.for);
  if (clonator) {
    return clonator.copy(obj) as T;
  }
  throw new Error('Unable to copy obj! Its type isn\'t supported.');
}
```

calls

calls

calls

```ts
const typeClonators = [
  {
    for: Date,
    copy(date) {
      const copy = new Date();
      copy.setTime(date.getTime());
      return copy;
    },
  },
  {
    for: Array,
    copy(array) {
      const copy = [];
      for (let i = 0, len = array.length; i < len; i++) {
        copy[i] = clone(array[i]);
      }
      return copy;
    }
  },
  {
    for: Object,
    copy(obj) {
      let copy = {};
      try {
        copy = obj.constructor ? new obj.constructor() : {};
      } catch (e) {
        copy = {};
      }
      Object.keys(obj).forEach((attr) => {
        /* eslint no-prototype-builtins: "off" */
        if (obj.hasOwnProperty(attr)) {
          copy[attr] = clone(obj[attr]);
        }
      });
      return copy;
    }
  }
];
```



Composite teoría

```abstract class Component {
    operation(): any { }
}```

extends

extends

```class Leaf extends Component {
    operation(): any { }
}```

```class Composite extends Component {
    protected children: Component[] = [];

    add(component: Component) { }
    remove(component: Component) { }
    operation(): any { }
}```


```

Composite realidad

```ts
export class Customer implements Validatable {
    constructor(private name: string) { }

    validate(): Array<string> {
        const errors = [''];
        if (!this.name) {
            errors.push('name is mandatory');
        }
        return errors;
    }
}
```

implements

```ts
export interface Validatable {
    validate(): Array<string>;
}
```

implements

```ts
export class Product implements Validatable {
    constructor(private name: string) { }

    validate(): Array<string> {
        const errors = [''];
        if (!this.name) {
            errors.push('name is mandatory');
        }
        return errors;
    }
}
```

implements

has

```ts
export class Order implements Validatable {
    constructor(private productList: Array<Product>, private customer: Customer) { }

    validate(): Array<string> {
        let errors = [''];
        this.productList.forEach(p => errors = errors.concat(p.validate()));
        errors = errors.concat(this.customer.validate());
        return errors;
    }
}
```

has

Usage:

```ts
export class Client {
    validateEntity(): void {
        const productList = [new Product('one product'), new Product('another product')];
        const customer = new Customer('a customer');
        const errors = new Order(productList, customer).validate();
        console.log(errors);
    }
}
```

Mediator teoría

```
interface IComponent {
    operation(): any;
}
```

implements

implements

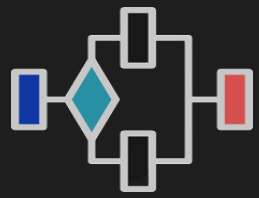
```
class Component1 implements IComponent {
    operation(): any { }
}
```

```
class Component2 implements IComponent {
    operation(): any { }
}
```

calls

calls

```
class Mediator {
    component1: IComponent;
    component2: IComponent;

    operation(): any {
        this.component1.operation();
        this.component2.operation();
    }
}
```



Mediator realidad

```
<observer1></observer1>
<observer2></observer2>

@Component({
    selector: 'observer',
    templateUrl: './real-world.component.html'
})
export class ObserverRealWorldComponent { }
```

updates

calls

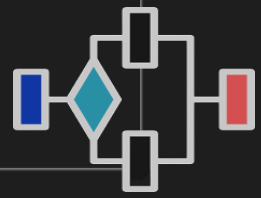
calls

updates

```
@Injectable()
export class ConnectorService {
    private currentState: Order;
    private observers = [];

    constructor(private store: Store) {
        this.currentState = this.store.state;
    }

    registerObserver(component) {
        this.observers.push(component);
        this.notifyObserver(component);
    }

    updateState(newState: Order) {
        if (this.isStateChanged(newState)) {
            this.currentState = newState;
            this.notifyObservers();
        }
    }

    private notifyObservers() {
        this.observers.forEach(o => this.notifyObserver(o));
    }
```

```
@Component({
    selector: 'observer1',
    templateUrl: './observer1.component.html'
})
export class Observer1Component {
    public productNames: Array<string> = [];
    private model: Order;

    @Input() set order(orderFromState: Order) {
        this.model = orderFromState;
        this.buildFormattedOrderList();
    }
    get order() { return this.model; }

    constructor(private connector: ConnectorService) {
        this.connector.registerObserver(this);
    }

    private buildFormattedOrderList(): void {
        this.productNames = this.model.productList.map(p => p.name);
    }

    addNewProduct() {
        this.connector.updateState(
            addNewProductStateAction(this.model)
        );
    }
}
```

```
@Component({
    selector: 'observer2',
    templateUrl: './observer2.component.html'
})
export class Observer2Component {
    public productNames: Array<string> = [];
    private model: Order;

    @Input() set order(orderFromState: Order) {
        this.model = orderFromState;
        this.buildFormattedOrderList();
    }
    get order() { return this.model; }

    constructor(private connector: ConnectorService) {
        this.connector.registerObserver(this);
    }

    private buildFormattedOrderList(): void {
        this.productNames = this.model.productList.map(p => p.name);
    }

    addNewProduct() {
        this.connector.updateState(
            addNewProductStateAction(this.model)
        );
    }
}
```



Memento teoría

```
class Memento {
    constructor(private state: string) { }

    getSavedState(): string {
        return this.state;
    }
}
```

remembers

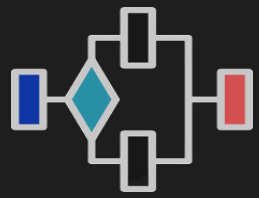
```
class Caretaker {
    private savedStates: Array<Memento> = [];
    addMemento(m: Memento): void { this.savedStates.push(m); }
    getMemento(index: number): Memento { return this.savedStates[index]; }
}
```

creates and restores

```
class Originator {
    private state: string;

    set(state: string): void {
        this.state = state;
    }

    saveToMemento(): Memento {
        return new Memento(this.state);
    }

    restoreFromMemento(m: Memento): void {
        this.state = m.getSavedState();
    }
}
```

Usage:

```
export class Client {
    run() {
        const caretaker = new Caretaker();

        const originator = new Originator();
        originator.set("State1");
        originator.set("State2");
        caretaker.addMemento(originator.saveToMemento());
        originator.set("State3");
        caretaker.addMemento(originator.saveToMemento());
        originator.set("State4");

        originator.restoreFromMemento(caretaker.getMemento(1));
    }
}
```



Memento realidad

```
<observer1></observer1>
<observer2></observer2>

@Component({
    selector: 'observer',
    templateUrl: './real-world.component.html'
})
export class ObserverRealWorldComponent { }
```

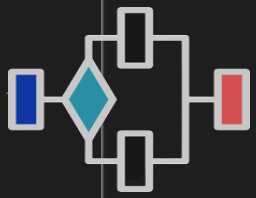
updates

```
@Component({
    selector: 'observer1',
    templateUrl: './observer1.component.html'
})
export class Observer1Component {
    public productNames: Array<string> = [];
    private model: Order;

    @Input() set order(orderFromState: Order) {
        this.model = orderFromState;
        this.buildFormattedOrderList();
    }
    get order() { return this.model; }

    constructor(private connector: ConnectorService) {
        this.connector.registerObserver(this);
    }

    private buildFormattedOrderList(): void {
        this.productNames = this.model.productList.map(p => p.name);
    }

    addNewProduct() {
        this.connector.updateState(
            addNewProductStateAction(this.model)
        );
    }
}
```

```
@Component({
    selector: 'observer2',
    templateUrl: './observer2.component.html'
})
export class Observer2Component {
    public productNames: Array<string> = [];
    private model: Order;

    @Input() set order(orderFromState: Order) {
        this.model = orderFromState;
        this.buildFormattedOrderList();
    }
    get order() { return this.model; }

    constructor(private connector: ConnectorService) {
        this.connector.registerObserver(this);
    }

    private buildFormattedOrderList(): void {
        this.productNames = this.model.productList.map(p => p.name);
    }

    addNewProduct() {
        this.connector.updateState(
            addNewProductStateAction(this.model)
        );
    }
}
```

updates

```
@Injectable()
export class ConnectorService {
    private currentState: Order;
    private observers = [];

    constructor(private store: Store) {
        this.currentState = this.store.state;
    }

    registerObserver(component) {
        this.observers.push(component);
        this.notifyObserver(component);
    }

    updateState(newState: Order) {
        if (this.isStateChanged(newState)) {
            this.currentState = newState;
            this.notifyObservers();
        }
    }

    private notifyObservers() {
        this.observers.forEach(o => this.notifyObserver(o));
    }

    private notifyObserver(observer) {
        observer.order = this.currentState;
    }

    private isStateChanged(newState: Order): boolean
        return this.currentState.id !== newState.id;
    }
}
```



Model View ViewModel teoría

Model View ViewModel realidad

```
@Component({
    selector: 'mvvm',
    templateUrl: './real-world.component.html'
})
export class MvvmRealWorldComponent {
    public productNames: Array<string> = [];
    private model: Order;

    @Input() set order(orderFromState: Order) {
        this.model = orderFromState;
        this.buildFormattedOrderList();
    }
    get order() { return this.model; }

    constructor(private connector: ConnectorService) {
        this.connector.registerObserver(this);
    }

    private buildFormattedOrderList(): void {
        this.productNames = this.model.productList.map(p => p.name);
    }

    addNewProduct() {
        this.connector.updateState(
            addNewProductStateAction(this.model)
        );
    }
}
```

shows

calls

```
<ul>
    <li *ngFor="let productName of productNames">{{productName}}</li>
</ul>
<button (click)="addNewProduct()">Add new product</button>
```

contains

```
export class Order implements Validatable {
    constructor(public id: string,
        public productList: Array<Product>,
        private customer: Customer) { }

    validate(): Array<string> {
        let errors = [''];
        this.productList.forEach(p => errors = errors.concat(p.validate()));
        errors = errors.concat(this.customer.validate());
        return errors;
    }
}
```