

## **Patrones de diseños Arquitectónicos**

Aprendiz:

Jorge Mauricio Cardozo Pinto

Instructor:

Néstor Guillermo Montaño Gomes

Ficha:

3064241

Análisis y Desarrollo de Software

# Patrones Creacionales

Patrones Creacionales	Definición	Uml	JavaScript
Singleton	Garantiza que una clase tenga <b>una única instancia</b> y que se pueda acceder a ella desde un punto global.	<pre> classDiagram     class Singleton {         &lt;&lt;private&gt;&gt; instance: Singleton         &lt;&lt;public&gt;&gt; Singleton()         &lt;&lt;public&gt;&gt; getInstance(): Singleton     }   </pre>	<pre> class Singleton { constructor() {     return Singleton.instance ??= this; } } const s1 = new Singleton(); const s2 = new Singleton(); console.log(s1 === s2); // true   </pre>
Factory Method	Permite delegar en las <b>subclases</b> la creación de objetos, usando un método común.	<pre> classDiagram     class Creator {         &lt;&lt;abstract&gt;&gt; factoryMethod(): Product     }     class ConcreteCreatorA {         &lt;&lt;concrete&gt;&gt; factoryMethod(): ProductA     }   </pre>	<pre> class ProductA { operation() {     return "Producto A"; } } class CreatorA { factoryMethod() {     return new ProductA(); } } const product = new CreatorA().factoryMethod(); console.log(product.operatio n());   </pre>

Abstract Factory	<p>Proporciona una interfaz para crear <b>familias de objetos relacionados</b>, sin especificar sus clases concretas.</p>	<pre> classDiagram     class GUIFactory {         &lt;&lt;I&gt;&gt;         createButton(): Button         createCheckbox(): Checkbox     }     class WinFactory {         &lt;&lt;C&gt;&gt;         createButton(): Button         createCheckbox(): Checkbox     }     class MacFactory {         &lt;&lt;C&gt;&gt;         createButton(): Button         createCheckbox(): Checkbox     }     class WinButton {         &lt;&lt;I&gt;&gt;         paint()     }     class MacButton {         &lt;&lt;I&gt;&gt;         paint()     }      GUIFactory &lt; -- WinFactory     GUIFactory &lt; -- MacFactory     WinFactory &lt; -- WinButton     MacFactory &lt; -- MacButton   </pre>	<pre> class WinButton { paint()   console.log("Botón Windows"); } class MacButton { paint()   console.log("Botón Mac"); }  const WinFactory =   createButton: () =&gt; new     WinButton(); const MacFactory =   createButton: () =&gt; new     MacButton();  const app = f =&gt; f.createButton().paint();  app(WinFactory); app(MacFactory);   </pre>
Builder	<p>Permite construir objetos complejos mediante <b>pasos separados</b>, sin depender del orden.</p>	<pre> classDiagram     class Director {         &lt;&lt;C&gt;&gt;         construct()     }     class Builder {         &lt;&lt;I&gt;&gt;         reset()         setPartA()         setPartB()     }     class ConcreteBuilder {         &lt;&lt;C&gt;&gt;         reset()         setPartA()         setPartB()         build(): Product     }     class Product {         &lt;&lt;I&gt;&gt;     }      Director &lt; --&gt; Builder     Director &lt; --&gt; ConcreteBuilder     ConcreteBuilder &lt; --&gt; Product   </pre>	<pre> class Product { constructor() {   this.parts = []; }  class Builder { constructor() { this.product = new Product(); }   setPartA() { this.product.parts.push("Parte A"); return this; }   setPartB() { this.product.parts.push("Parte B"); return this; }   build() { return this.product; } }   </pre>

Prototyp e	Permite crear nuevos objetos <b>clonando</b> un objeto existente (prototipo)	<pre> classDiagram     class Prototype {         &lt;&lt;I&gt;&gt;         clone(): Prototype     }     class ConcretePrototype {         state: string         clone(): ConcretePrototype     }     Prototype &lt; -- ConcretePrototype   </pre>	<pre> const product = new Builder().setPartA().setPart B().build(); console.log(product.parts);  const proto = { nombre: "Original", saludar() { console.log("Hola soy " + this.nombre); } };  const clon = { ...proto, nombre: "Clon" };  proto.saludar(); clon.saludar();   </pre>

## Patrones Estructurales

Patrón Estructural	Definición	UML	JavaScript
Adapter	Permite que dos clases con <b>interfaces incompatibles</b> trabajen juntas mediante un adaptador.	<pre> classDiagram     class Cliente     class Target {         request()     }     class Adapter {         adaptee: Adaptee         request()     }     class Adaptee {         specificRequest()     }     Cliente --&gt; Target     Adapter --&gt; Target     Adapter --&gt; Adaptee   </pre>	<pre> class Adaptee { specificRequest() { return "Datos del Adaptee"; } } class Adapter { request() { return new Adaptee().specificReq uest(); } } console.log(new Adapter().request());   </pre>

Bridge	<p>Separa una abstracción de su implementación, permitiendo que ambas cambien de manera independiente.</p>	<pre> classDiagram     class Abstraction {         &lt;&lt;Implementor: Implementor&gt;&gt;         &lt;&lt;operation()&gt;&gt;     }     class Implementor {         &lt;&lt;operationImpl()&gt;&gt;     }     class RefinedAbstraction     class ConcreteImplementorA     class ConcreteImplementorB      Abstraction --&gt; Implementor     Abstraction --&gt; RefinedAbstraction     Implementor --&gt; ConcreteImplementorA     Implementor --&gt; ConcreteImplementorB     ConcreteImplementorA --&gt; RefinedAbstraction     ConcreteImplementorB --&gt; RefinedAbstraction   </pre>	<pre> class ImplA {     operationImpl()     { return "Implementación A"; } } class ImplB {     operationImpl()     { return "Implementación B"; } }  class Abs {     constructor(i)     { this.i = i; }     operation()     { return this.i.operationImpl( ); } }  console.log(new Abs(new ImplB()).operation()) ;   </pre>
Composite	<p>Permite tratar objetos individuales y compuestos <b>de la misma manera</b>.</p>	<pre> classDiagram     class Component {         &lt;&lt;operation()&gt;&gt;     }     class Composite {         &lt;&lt;children: List&lt;Component&gt;&gt;&gt;         &lt;&lt;add()         &lt;&lt;remove()         &lt;&lt;operation()     }     class Leaf      Component --&gt; Composite     Component --&gt; Component     Composite --&gt; Composite   </pre>	<pre> class Leaf { operation() { console.log("Leaf") } }  class Composite {     constructor()     { this.children = []; }     add(c)     { this.children.push( c); }     operation()     { this.children.forEach( c =&gt; c.operation()); }   </pre>

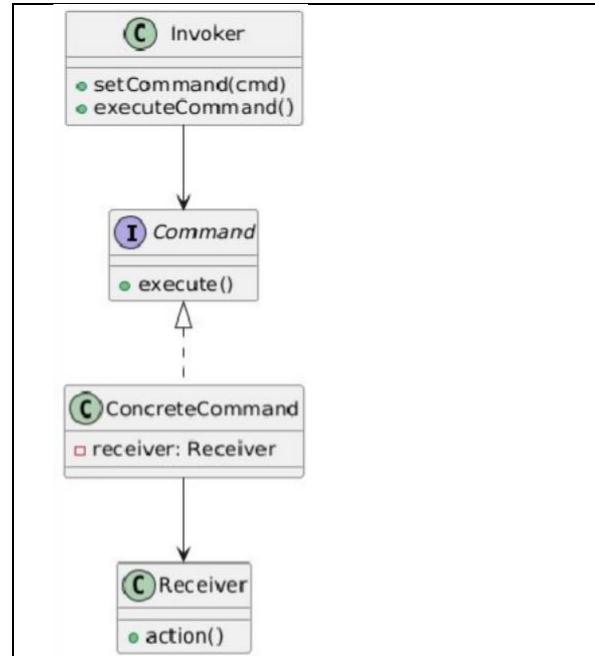
Decorador	Permite <b>agregar responsabilidad des</b> a un objeto dinámicamente sin modificar su clase.	<pre> classDiagram     class Component {         &lt;&lt;operation()&gt;&gt;     }     class ConcreteComponent {         --&gt; Component     }     class Decorator {         &lt;&lt;component : Component&gt;&gt;     }     class ConcreteDecoratorA {         --&gt; Decorator     }     class ConcreteDecoratorB {         --&gt; Decorator     }     </pre>	<pre> const tree = new Composite(); tree.add(new Leaf()); tree.add(new Leaf()); tree.operation(); }  class Component {     operation()     { return "Componente base"; } }  class Decorator { constructor(c) { this.c = c; } operation()      { return this.c.operation(); }  }  class DecoratorA extends Decorator {     operation()     { return super.operation() + " + Decorador A"; }  } console.log(new DecoratorA(new Component()).operation()); </pre>
Facade	Simplifica el acceso a un sistema complejo ofreciendo <b>una interfaz unificada</b> .		<pre> class A { metodo() { return "A"; } } class B { metodo() { return "B"; } } class C { metodo() { return "C"; } }  class Facade { constructor() </pre>

		<pre> class FlyweightFactory {     flyweights: Map     getFlyweight(key) }  class Flyweight {     operation(shared) } </pre>	<pre> class Flyweight { constructor(color) { this.color = color; } }  class FlyweightFactory {     constructor() { this.pool = {}; } get(c) { return this.pool[c] ??= new Flyweight(c); } }  const f = new FlyweightFactory(); const f1 = f.get("red"); const f2 = f.get("red"); </pre>
Flyweight	Optimiza memoria compartiendo objetos que son <b>idénticos o muy similares.</b>		<pre> console.log(f1 === f2); // true </pre>

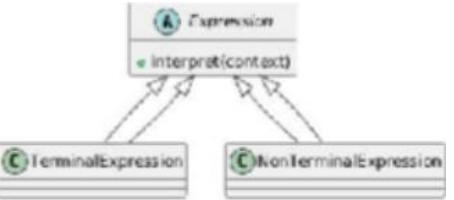
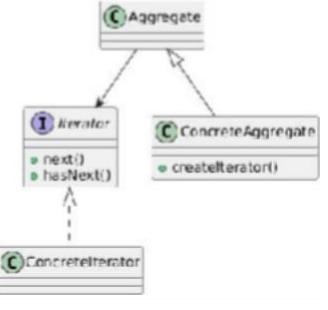
Proxy	Proporciona un objeto sustituto para <b>controlar el acceso</b> al objeto real.	<pre> classDiagram     class Subject {         &lt;&lt;I&gt;&gt;         &lt;&lt;request()&gt;&gt;     }     class Proxy {         &lt;&lt;C&gt;&gt;         &lt;&lt;realSubject: RealSubject&gt;&gt;         &lt;&lt;request()&gt;&gt;     }     class RealSubject {         &lt;&lt;C&gt;&gt;         &lt;&lt;request()&gt;&gt;     }     Subject &lt; -- Proxy     Proxy &lt; -- RealSubject     </pre>	<pre> class Real {     request()     { console.log("Operación real");     } }  class Proxy { request() { console.log("Accediendo por Proxy"); new Real().request(); } }  new Proxy().request(); </pre>
-------	---	--	--

### Patrones de Comportamiento

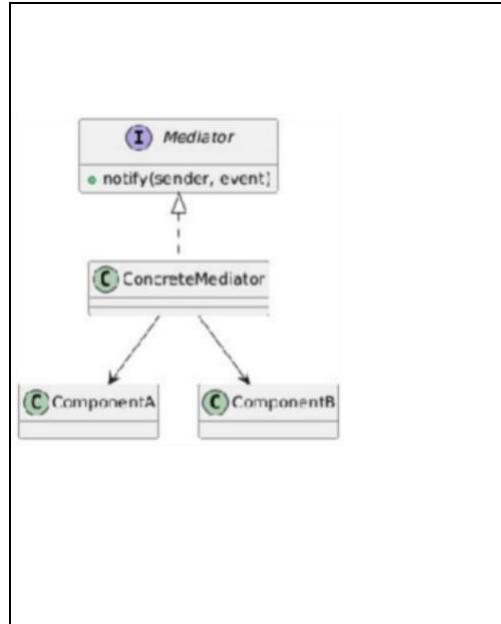
Patrones de Comportamiento	Definición	UML	JavaScript
Command	Convierte una solicitud en un objeto, permitiendo deshacer, almacenar y ejecutar comandos.		<pre> class Receiver {     action()     { console.log("Acción ejecutada");     } }  class Command { constructor(r) { this.r = r; } execute()  { this.r.action(); }  class Invoker {     setCommand(c) { this.c = c; }     executeCommand() { this.c.execute(); } </pre>



```
const invoker = new
Invoker();
invoker.setCommand(new
Command(new
Receiver()));
invoker.executeCommand()
;
```

Interpreter	<p>Define una gramática y un intérprete para evaluar expresiones</p> <p>.</p>	 <pre> classDiagram     class Expression {         &lt;&lt;Expression&gt;&gt;         &lt;&lt;interpret(context)&gt;&gt;     }     class TerminalExpression {         &lt;&lt;TerminalExpression&gt;&gt;         &lt;&lt;interpret(context)&gt;&gt;     }     class NonterminalExpression {         &lt;&lt;NonterminalExpression&gt;&gt;         &lt;&lt;interpret(context)&gt;&gt;     }     Expression &lt; -- TerminalExpression     Expression &lt; -- NonterminalExpression   </pre>	<pre> class Num { constructor(v) { this.v = v; } interpret() { return this.v; } } class Add { constructor(l, r) { this.l = l; this.r = r; } interpret() { return this.l.interpret() + this.r.interpret(); } }  const expr = new Add(new Num(5), new Num(3)); console.log(expr.interpret()); // 8   </pre>
Iterator	<p>Proporciona una forma uniforme de recorrer elementos de una colección.</p>	 <pre> classDiagram     class Aggregate {         &lt;&lt;Aggregate&gt;&gt;     }     class ConcreteAggregate {         &lt;&lt;ConcreteAggregate&gt;&gt;         &lt;&lt;createIterator()&gt;&gt;     }     interface Iterator {         &lt;&lt;Iterator&gt;&gt;         &lt;&lt;next()&gt;&gt;         &lt;&lt;hasNext()&gt;&gt;     }     ConcreteAggregate &lt; -- Aggregate     ConcreteAggregate &lt; -- Iterator     Iterator &lt; -- ConcreteIterator     ConcreteIterator &lt; -- Iterator   </pre>	<pre> class Iterator {     constructor(items)     { this.i = 0; this.items = items; }     next() { return this.items[this.i++]; }     hasNext() { return this.i &lt; this.items.length; } }  const it = new Iterator([1,2,3]); while(it.hasNext())   console.log(it.next());   </pre>
Mediator	<p>Centraliza la</p>		<pre> class Mediator { notify(s,e) {} }   </pre>

comunicación entre objetos para evitar dependencias directas.



```

class ConcreteMediator
extends Mediator {
  constructor(a,b){ super();
  [this.a,this.b] = [a,b];
  a.setMediator(this);
  b.setMediator(this);
}
notify(s,e){
  if(e==="A")
  this.b.doB();
  if(e==="B")
  this.a.doA();
}
}

class Component
{ setMediator(m) { this.m = m; } }

  
```

```

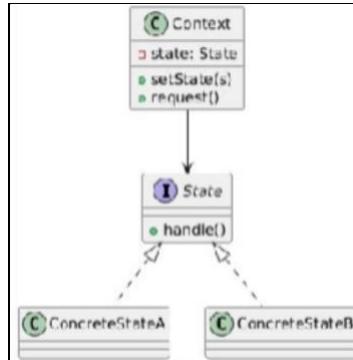
class A extends Component
{ doA(){ console.log("A ejecuta"); }
action(){ this.m.notify(this,"A"); } }
class B extends Component
{ doB(){ console.log("B ejecuta"); }
action(){ this.m.notify(this,"B"); } }

const a = new A(), b = new B();
new ConcreteMediator(a,b);
a.action();

  
```

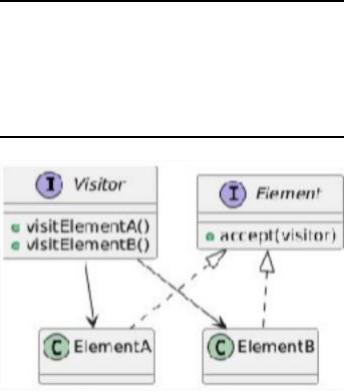
Memento	Guarda el estado de un objeto sin violar encapsulado, permitiendo restaurarlo.	<pre> classDiagram     class Originator {         &lt;&lt;Originator&gt;&gt;         &lt;&lt;Memento&gt;&gt;         &lt;&lt;Caretaker&gt;&gt;     }     class Memento {         &lt;&lt;state&gt;&gt;     }     class Caretaker {         &lt;&lt;mementos: List&lt;Memento&gt;&gt;&gt;     }     Originator &lt; -- Memento     Originator &lt; -- Caretaker     Originator &lt;--&gt; Memento     Caretaker --&gt; Memento   </pre>	<pre> class Memento { constructor(s){ this.s = s; } getState(){ return this.s; } } class Originator {  setState(s){ this.s=s; } save(){ return new Memento(this.s); }  restore(m){ this.s=m.getState(); } } class Caretaker {  constructor(){ this.h=[]; }  add(m){ this.h.push(m); }  get(i){ return this.h[i]; } }  const o = new Originator(), c = new Caretaker(); o.setState("A"); c.add(o.save()); o.setState("B"); o.restore(c.get(0)); console.log(o.s); // "A"   </pre>
---------	--	--	--

Observer	Permite que múltiples objetos se suscriban a cambios de otro objeto.	<pre> classDiagram     class Subject {         &lt;&lt;C&gt;&gt;         +attach(observer)         +detach(observer)         +notify()     }     class Observer {         &lt;&lt;I&gt;&gt;         +update()     }     Subject --&gt; Observer     </pre>	<pre> class Subject { constructor() {     this.o = []; } attach(x) { this.o.push(x); }  notify() { this.o.forEach (x =&gt; x.update()); }  }  class Observer {     update() { console.log("Notificado"); } }  const s = new Subject(); s.attach(new Observer()); s.notify(); </pre>
State	Permite que un objeto cambie su comportamiento según su estado interno.		<pre> class A {     handle() { console.log( "Estado A"); } } class B {     handle() { console.log( "Estado B"); } }  class Context {     setState(s) {         this.s = s;     }     request() { this.s.handle(); } } </pre>



```
const ctx = new Context();
ctx.setState(new A());
ctx.request();
ctx.setState(new B());
ctx.request();
```

Strategy	Permite cambiar algoritmos en tiempo de ejecución.	<pre> classDiagram     class Context {         strategy: Strategy         setStrategy(s)         execute()     }     class IStrategy {         execute()     }     class StrategyA     class StrategyB     Context "1" --&gt; "1" IStrategy     Context "1" --&gt; "1" StrategyA     Context "1" --&gt; "1" StrategyB     IStrategy &lt; -- StrategyA     IStrategy &lt; -- StrategyB   </pre>	<pre> class A { execute(){ console.log ("Estrategia A"); } } class B { execute(){ console.log ("Estrategia B"); } }  class Context { setStrategy(s){ this.s = s; } execute(){ this.s.execute(); } }  const ctx = new Context(); ctx.setStrategy(new A()); ctx.execute();   </pre>
Template Method	Define el esqueleto de un algoritmo en una clase base, dejando pasos definidos por las subclases.	<pre> classDiagram     class AbstractClass {         templateMethod()         step1()         step2()     }     class ConcreteClass     AbstractClass &lt; -- ConcreteClass   </pre>	<pre> class Abstract {     templateMethod(){ this.step1(); this.step2(); }     step1(){ throw "abstract"; }     step2(){ throw "abstract"; } }  class Concrete extends Abstract {     step1(){ console.log("Pa so 1"); }     step2(){ console.log("Pa so 2"); } }   </pre>

			new Concrete().templateMethod();
Visitor	Permite agregar nuevas operaciones a objetos sin modificar sus clases.	 <pre> classDiagram     class Visitor {         &lt;&lt;I&gt;&gt; Visitor         &lt;&lt;I&gt;&gt; visitElementA()         &lt;&lt;I&gt;&gt; visitElementB()     }     class Element {         &lt;&lt;I&gt;&gt; Element         &lt;&lt;I&gt;&gt; accept(visitor)     }     class ConcreteElement {         &lt;&lt;C&gt;&gt; ElementA         &lt;&lt;C&gt;&gt; ElementB     }     Visitor --&gt; Element : accept(visitor)     Visitor --&gt; ElementA : visitElementA()     Visitor --&gt; ElementB : visitElementB()   </pre>	<pre> class A { accept(v){ v.visitA(this); } }  class B { accept(v){ v.visitB(this); } }  class Visitor {      visitA(){ console.log("Visiting A"); }      visitB(){ console.log("Visiting B"); }  }  const elems = [new A(), new B()]; const visitor = new Visitor(); elems.forEach(e =&gt; e.accept(visitor));   </pre>
		<pre> visitB(){ console.log("Visiting B"); }  const elems = [new A(), new B()]; const visitor = new Visitor(); elems.forEach(e =&gt; e.accept(visitor));   </pre>	