Patrones de diseños Arquitectónicos

Aprendiz:

Jorge Mauricio Cardozo Pinto

Instructor:

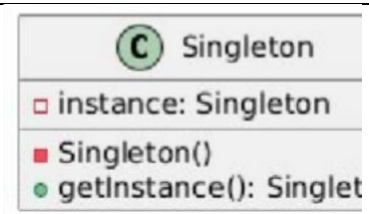
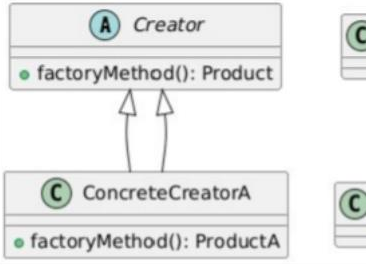
Néstor Guillermo Montaño Gomes

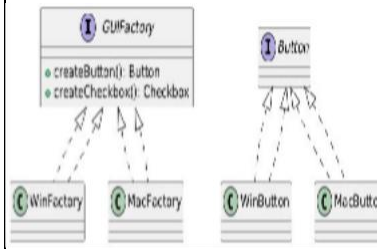
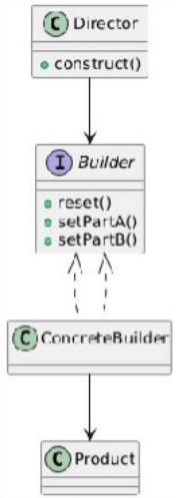
Ficha:

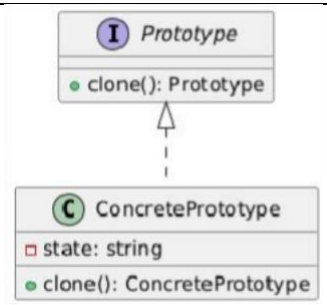
3064241

Análisis y Desarrollo de Software

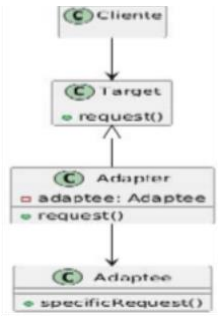
Patrones Creacionales

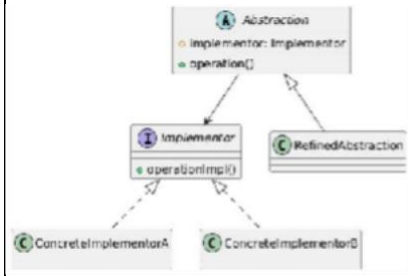
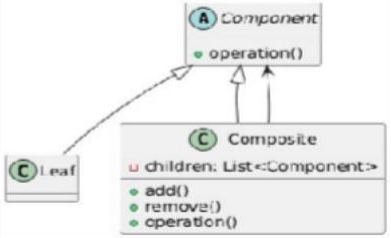
Patrones Creacion ales	Definición	Uml	JavaScript
Singleto n	Garantiza que una clase tenga **una única instancia** y que se pueda acceder a ella desde un punto global.		`class Singleton { constructor() { return Singleton.instance ??= this; } } const s1 = new Singleton(); const s2 = new Singleton(); console.log(s1 === s2); // true`
Factory Method	Permite delegar en las **subclase s** la creación de objetos, usando un método común.		`class ProductA { operation() { return "Producto A"; } } class CreatorA { factoryMethod() { return new ProductA(); } } const product = new CreatorA().factoryMethod(); console.log(product.operatio n());`

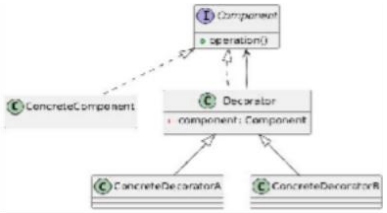
Abstract Factory	Proporcio na una interfaz para crear **familias de objetos relaciona dos**, sin especifica r sus clases concretas.		`class WinButton { paint() { console.log("Botón Windows"); } } class MacButton { paint() { console.log("Botón Mac"); } }` `const WinFactory = { createButton: () => new WinButton() }; const MacFactory = { createButton: () => new MacButton() };` `const app = f => f.createButton().paint();` `app(WinFactory); app(MacFactory);`
Builder	Permite construir objetos complejos mediante **pasos separado s**, sin depender del orden.		`class Product { constructor() { this.parts = []; } }` `class Builder { constructor() { this.product = new Product(); } setPartA() { this.product.parts.push("P arte A"); return this; } setPartB() { this.product.parts.push("P arte B"); return this; } build() { return this.product; } }`

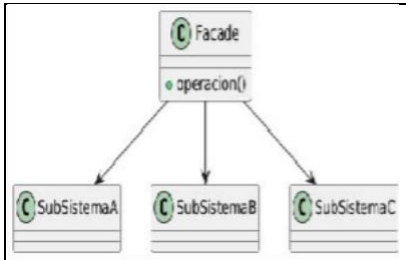
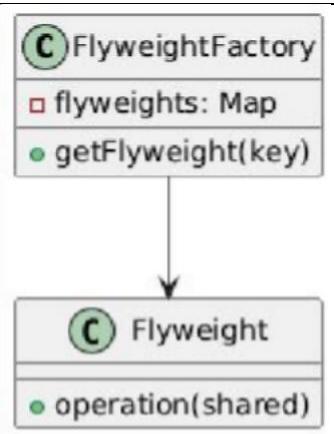
			`const product = new Builder().setPartA().setPart B().build(); console.log(product.parts);`
Prototyp e	Permite crear nuevos objetos **clonando** un objeto existente (prototipo) .		`const proto = { nombre: "Original", saludar() { console.log("Hola soy " + this.nombre); } };` `const clon = { ...proto, nombre: "Clon" };` `proto.saludar(); clon.saludar();`

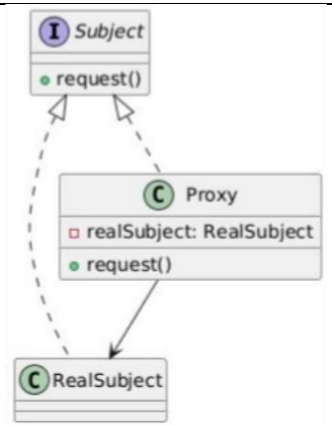
Patrones Estructurales

Patrón Estructur al	Definición	UML	JavaScript
Adapter	Permite que dos clases con **interfaces incompatibles** trabajen juntas mediante un adaptador.		`class Adaptee { specificRequest() { return "Datos del Adaptee"; } }` `class Adapter { request() { return new Adaptee().specificReq uest(); } }` `console.log(new Adapter().request());`

Bridge	Separa una abstracción de su implementación, permitiendo que ambas cambien de manera independiente.		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()) ;
Composit e	Permite tratar objetos individuales y compuestos **de la misma** **manera**.		class Leaf { operation() { console.log("Leaf") ; } } class Composite { constructor() { this.children = []; } add(c) { this.children.push(c); } operation() { this.children.forEa ch(c => c.operation()); }

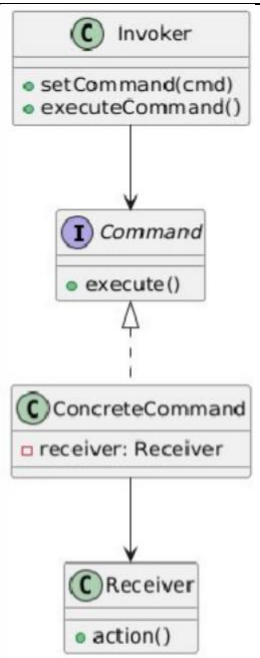
			} const tree = new Composite(); tree.add(new Leaf()); tree.add(new Leaf()); tree.operation();
Decorato r	Permite **agregar responsabilida des** a un objeto dinámicamente sin modificar su clase.		class Component { operation() { return "Componente base"; } } class Decorator { constructor(c) { this.c = c; } operation()
			{ return this.c.operation(); } } class DecoratorA extends Decorator { operation() { return super.operation() + " + Decorador A"; } } console.log(new DecoratorA(new Component()).operatio n());
	Facade	Simplifica el acceso a un sistema complejo ofreciendo **una interfaz** **unificada**.	class A { metodo() { return "A"; } } class B { metodo() { return "B"; } } class C { metodo() { return "C"; } } class Facade { constructor()

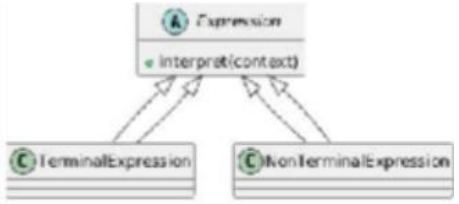
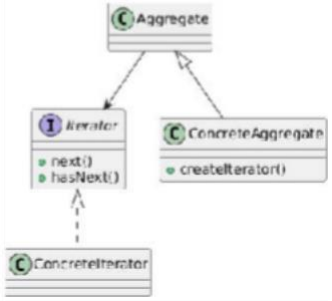
			`{ [this.a, this.b, this.c] = [new A(), new B(), new C()]; } operacion() { return `${this.a.metodo()} ${this.b.metodo()} ${this.c.metodo()}`; } } console.log(new Facade().operacion()) ;`
Flyweigh t	Optimiza memoria compartiendo objetos que son **idénticos o muy similares**.		`class Flyweight { constructor(color) { 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");`
			`console.log(f1 === f2); // true`

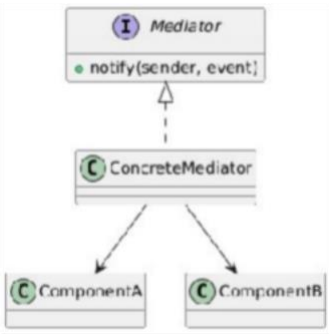
Proxy	Proporciona un objeto sustituto para **controlar el acceso** al objeto real.		class Real { request() { console.log("Operac ión real"); } } class Proxy { request() { console.log("Accedi endo por Proxy"); new Real().request(); } } new Proxy().request();

Patrones de Comportamiento

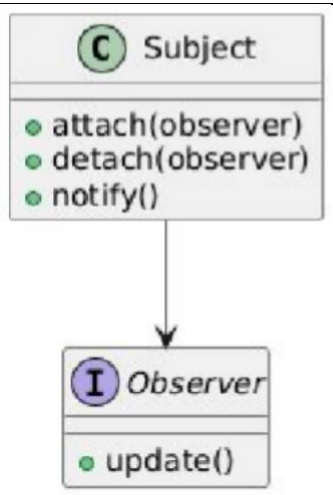
Patrones de Comportamie nto	Definición	UML	JavaScript
Command	Convierte una solicitud en un objeto, permitiendo deshacer, almacenar y ejecutar comandos.		class Receiver { action() { console.log("Acción ejecutada"); } } class Command { constructor(r) { this.r = r; } execute() { this.r.action(); } } class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

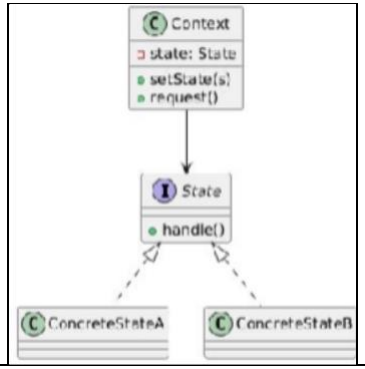
			``` const invoker = new Invoker(); invoker.setCommand(new Command(new Receiver())); invoker.executeCommand() ; ```

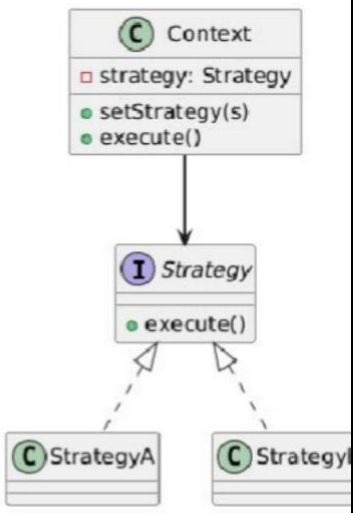
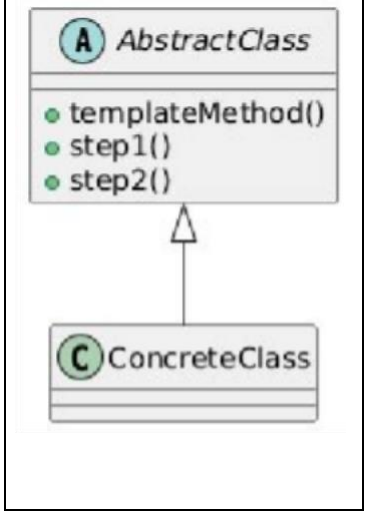
Interpreter	Define una gramática y un intérprete para evaluar expresiones .		class Num { constructor(v) { this.v = v; } interpret() { return this.v; } } class Add { constructor(l, r) { this.l = l; this.r = r; } interpret() { return this.l.interpret() + this.r.interpret(); } } const expr = new Add(new Num(5), new Num(3)); console.log(expr.interpret()); // 8
Iterator	Proporciona una forma uniforme de recorrer elementos de una colección.		class Iterator { constructor(items) { this.i = 0; this.items = items; } next() { return this.items[this.i++]; } hasNext() { return this.i < this.items.length; } } const it = new Iterator([1,2,3]); while(it.hasNext()) console.log(it.next());
Mediator	Centraliza la		class Mediator { notify(s,e) {} }

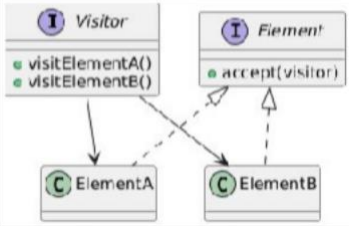
	comunicaci ón entre objetos para evitar dependenci as directas.		`class ConcreteMediator extends Mediator { constructor(a,b){ super(); [this.a,this.b] = [a,b]; a.setMediator(this); b.setMediator(this); } notify(s,e){ if(e==="A") this.b.doB(); if(e==="B") this.a.doA(); } } class Component { setMediator(m){ 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 encapsulam iento, permitiendo restaurarlo.		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.get State(); } } 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"

Observer	Permite que múltiples objetos se suscriban a cambios de otro objeto.		`class Subject { constructor(){ this.o = []; }` `attach(x){ this.o.push(x ); }` `notify(){ this.o.forEach (x => x.update()); } }` `class Observer { update(){ console.log( "Notificado"); } }` `const s = new Subject(); s.attach(new Observer()); s.notify();`
State	Permite que un objeto cambie su comportami ento según su estado interno.		`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 (); } }`

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

Strategy	Permite cambiar algoritmos en tiempo de ejecución.		class A { execute(){ console.log ("Estrategia A"); } } class B { execute(){ console.log ("Estrategia B"); } } class Context { setStrategy(s){ this.s = s; } execute(){ this.s.execut e(); } } const ctx = new Context(); ctx.setStrategy(new A()); ctx.execute();
Template Method	Define el esqueleto de un algoritmo en una clase base, dejando pasos definidos por las subclases.		class Abstract { templateMethod(){ this.s tep1(); this.step2(); } step1(){ throw "abstract"; } step2(){ throw "abstract"; } } class Concrete extends Abstract { step1(){ console.log("Pa so 1"); } step2(){ console.log("Pa so 2"); } }

			`new Concrete().templateMetho d();`
Visitor	Permite agregar nuevas operacione s a objetos sin modificar sus clases.		`class A` `{ accept(v){ v.visitA(th is); } }` `class B` `{ accept(v){ v.visitB(th is); } }` `class Visitor {` `visitA(){ console.log("V isitando A"); }`
			`visitB(){ console.log("V isitando B"); }` `}` `const elems = [new A(), new B()];` `const visitor = new Visitor(); elems.forEach(e =>` `e.accept(visitor));`