

Patrones de Diseño Arquitectónico

Aprendiz:

Juan David Buitrago Suárez

Instructor:

Ing. Néstor Montaño

Tecnólogo Análisis y Desarrollo de Software

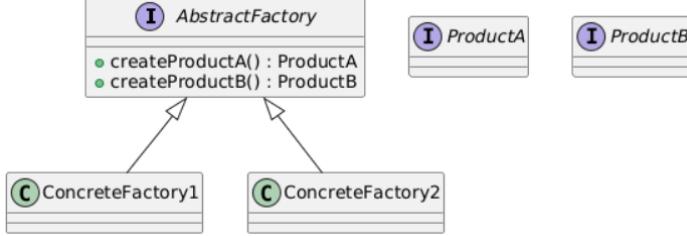
Ficha: 3064241

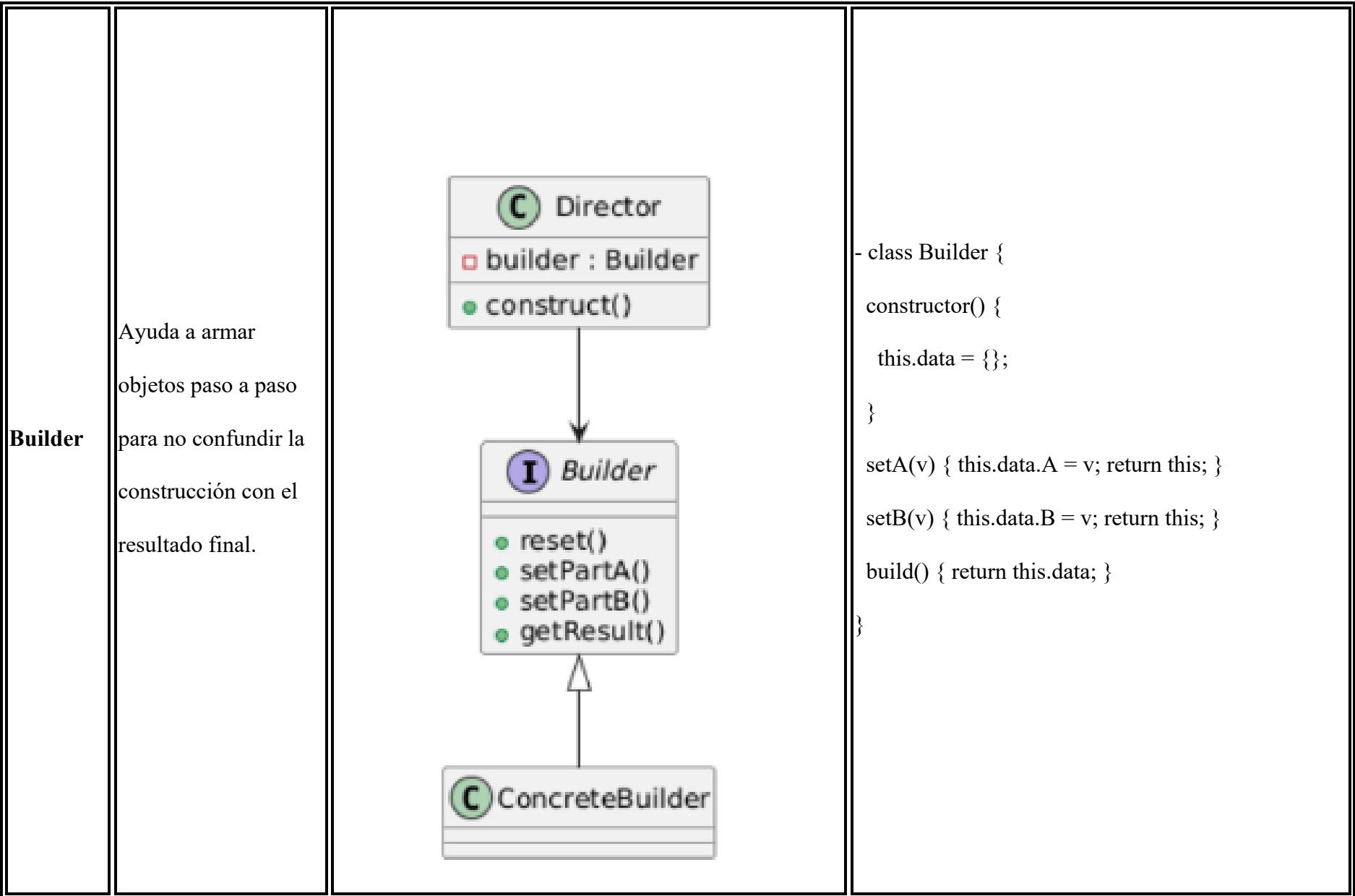
SENA Centro de Diseño y Metrología

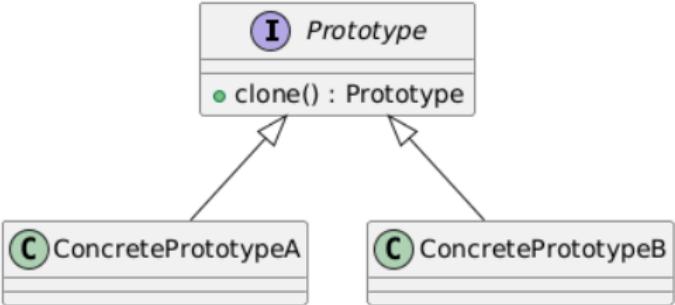
Patrones Creacionales

Nombre	Definición	Ejemplo en UML	Implementación en Node.js
Singleton	Asegura que solo existe un objeto de ese tipo y que todos usen el mismo.	<pre> classDiagram class Singleton { <<C>> #instance : Singleton <<S>> <<G>> <<I>> } </pre> <p>The diagram shows a UML class named 'Singleton'. It has a private attribute '#instance : Singleton'. It also contains a constructor 'Singleton()' and a static method 'getInstance() : Singleton'. A self-referencing association labeled 'instancia única' (unique instance) connects the class back to its own instance slot.</p>	<pre> - class Singleton { constructor() { if (Singleton.instance) return Singleton.instance; Singleton.instance = this; } } const a = new Singleton(); const b = new Singleton(); console.log(a === b); // true </pre>

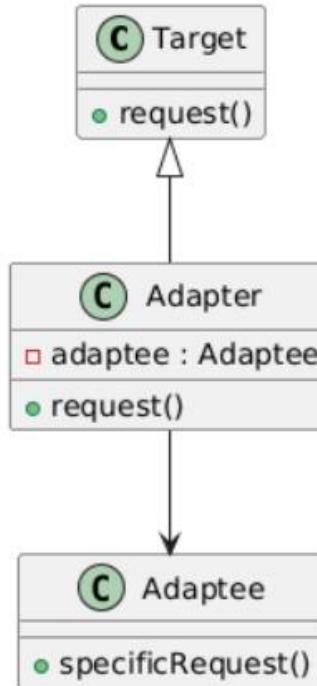
Nombre	Definición	Ejemplo en UML	Implementación en Node.js
Factory Method	Permite crear diferentes objetos sin decir exactamente cuál se va a crear.	<pre> classDiagram class Creator { <<A>> factoryMethod() : Product } class ConcreteCreatorA { <<C>> factoryMethod() : Product } class ConcreteCreatorB { <<C>> factoryMethod() : Product } class Product { <<I>> } class ProductA { <<C>> implements Product } class ProductB { <<C>> implements Product } Creator < -- ConcreteCreatorA Creator < -- ConcreteCreatorB Product < -- ProductA Product < -- ProductB </pre>	<pre> -class ProductoA { crear() { return "Producto A creado";} } class ProductoB { crear() { return "Producto B creado";} } class Creador { factoryMethod(tipo) { return tipo === "A" ? new ProductoA() : new ProductoB(); } } const creador = new Creador(); console.log(creador.factoryMethod("A").crear()); </pre>

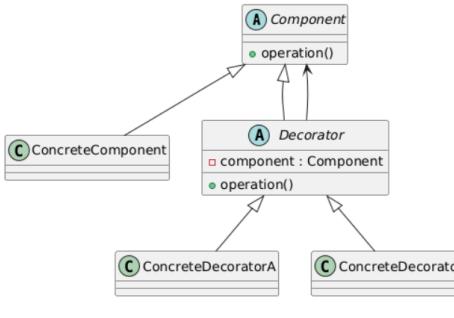
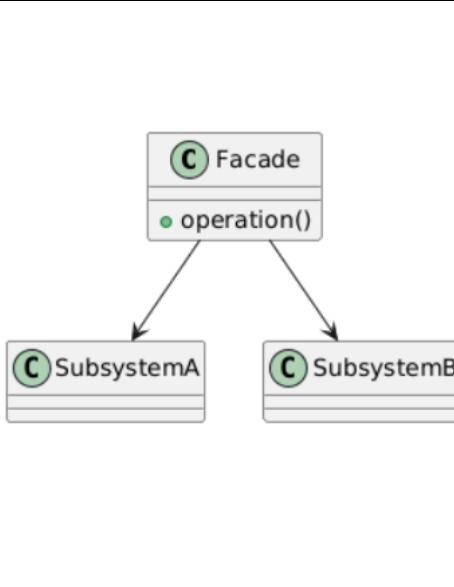
Nombre	Definición	Ejemplo en UML	Implementación en Node.js
Abstract Factory	Crea varios tipos de objetos que combinan bien entre sí sin mostrar cómo se hacen.	 <pre> classDiagram class AbstractFactory { <<I>> createProductA() : ProductA createProductB() : ProductB } class ProductA { <<I>> } class ProductB { <<I>> } class ConcreteFactory1 { <<C>> } class ConcreteFactory2 { <<C>> } AbstractFactory < -- ConcreteFactory1 AbstractFactory < -- ConcreteFactory2 ConcreteFactory1 --> ProductA ConcreteFactory1 --> ProductB ConcreteFactory2 --> ProductA ConcreteFactory2 --> ProductB </pre>	<pre> - class ButtonWindows { draw() { return "Botón Windows"; } } class MenuWindows { show() { return "Menú Windows"; } } class UIFactoryWindows { createButton() { return new ButtonWindows(); } createMenu() { return new MenuWindows(); } } </pre>

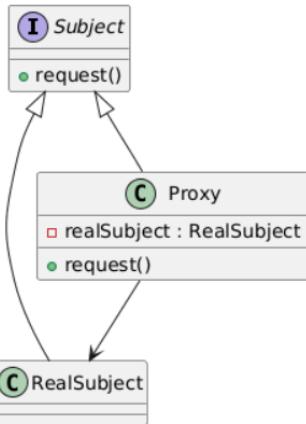
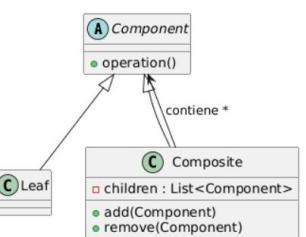


Nombre	Definición	Ejemplo en UML	Implementación en Node.js
Prototype	Permite copiar un objeto ya hecho para crear otro igual sin hacerlo desde cero.	 <pre> classDiagram class Prototype { +clone() : Prototype } class ConcretePrototypeA { +clone() : ConcretePrototypeA } class ConcretePrototypeB { +clone() : ConcretePrototypeB } Prototype < -- ConcretePrototypeA Prototype < -- ConcretePrototypeB Prototype --> clone() : Prototype ConcretePrototypeA --> clone() : ConcretePrototypeA ConcretePrototypeB --> clone() : ConcretePrototypeB </pre>	<pre> - const proto = { a: 1 }; const obj = Object.create(proto); </pre>

Patrones Estructurales

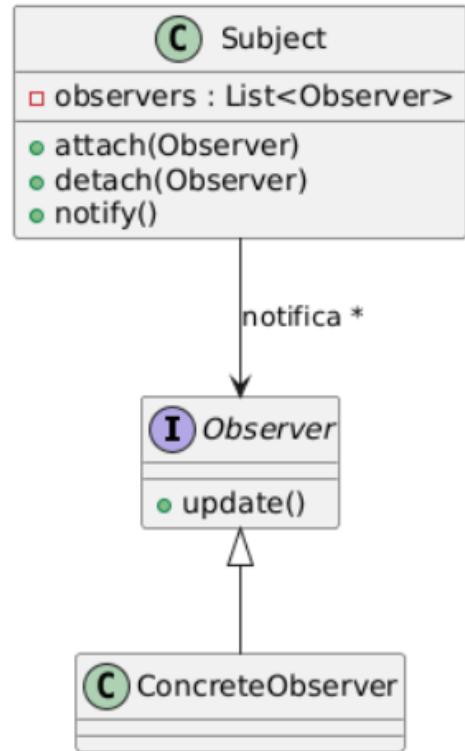
Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
Adapter	Hace que dos cosas que no son compatibles puedan funcionar juntas.	 <pre> classDiagram class Target { <<C>> request() } class Adapter { <<C>> adaptee : Adaptee request() } class Adaptee { <<C>> specificRequest() } Adapter "3..1" --> "1..1" Adaptee Adapter "3..1" --> "1..1" Target </pre>	<pre> class Old { specific() { return "Función antigua"; } } class Adapter { request() { return new Old().specific(); } } </pre>

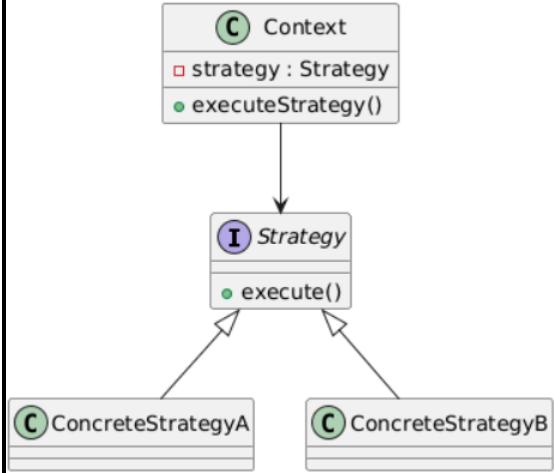
Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
Decorator	Permite agregar nuevas funciones a un objeto sin cambiarlo por dentro.	 <pre> classDiagram class Component { <<A>> operation() } class ConcreteComponent { <<C>> } class Decorator { <<A>> component : Component operation() } class ConcreteDecoratorA { <<C>> } class ConcreteDecoratorB { <<C>> } Component < -- ConcreteComponent Component < -- Decorator Decorator < -- ConcreteDecoratorA Decorator < -- ConcreteDecoratorB ConcreteComponent --> Component : operation() ConcreteDecoratorA --> Component : operation() ConcreteDecoratorB --> Component : operation() </pre>	<pre> function deco(fn) { return () => fn() + " decorado"; } </pre>
Facade	Da una forma fácil de usar un sistema complicado.	 <pre> classDiagram class Facade { <<C>> operation() } class SubsystemA { <<C>> } class SubsystemB { <<C>> } Facade < -- SubsystemA Facade < -- SubsystemB SubsystemA --> Facade : operation() SubsystemB --> Facade : operation() </pre>	<pre> class A { a() { console.log("A"); } } class B { b() { console.log("B"); } } class Facade { start() { new A().a(); new B().b(); } } </pre>

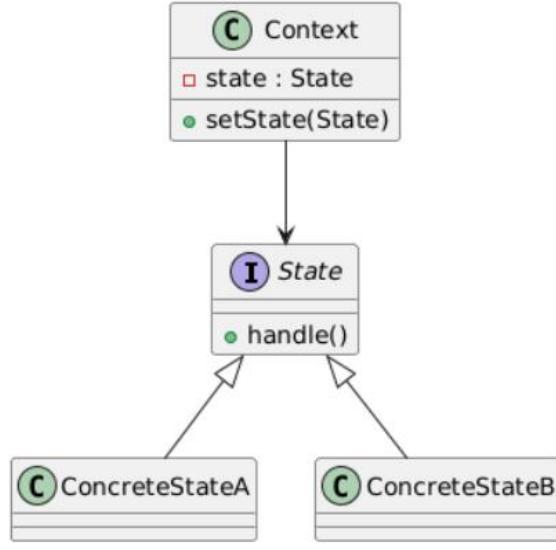
Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
Proxy	Es un intermediario que controla el acceso a otro objeto.	 <pre> classDiagram class ISubject { <<ISubject>> <<request()>> } class Proxy { <<Proxy>> <<realSubject : RealSubject>> <<request()>> } class RealSubject { <<RealSubject>> <<request()>> } ISubject "2" -- "1" Proxy Proxy --> "1" RealSubject </pre>	<pre> - const obj = { name: "Real" }; const proxy = new Proxy(obj, { get(target, prop) { console.log("Acceso controlado"); return target[prop]; } }); </pre>
Composite	Permite manejar varios objetos como si fueran uno solo.	 <pre> classDiagram class Component { <<Component>> <<operation()>> } class Composite { <<Composite>> <<children : List<Component>> <<add(Component)>> <<remove(Component)>> } class Leaf { <<Leaf>> <<operation()>> } Component "1" -- "1" Composite Composite "1" -- "1" Leaf </pre>	<pre> - class Component { add() {} operation() {} } </pre>

Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
Bridge	Separa lo que hace una clase de cómo está hecha para que sean más flexibles.	<pre> classDiagram class Abstraction { implementor : Implementor operation() } class Implementor { operationImpl() } class ConcreteAbstraction { <<ConcreteAbstraction>> } class ConcreteImplementorA { <<ConcreteImplementorA>> } class ConcreteImplementorB { <<ConcreteImplementorB>> } Abstraction "2" --> Implementor ConcreteAbstraction "2" --> ConcreteImplementorA ConcreteAbstraction "2" --> ConcreteImplementorB </pre>	<pre> class Abstraction { constructor(imp) { this.imp = imp; } run() { this.imp.run(); } } </pre>
Flyweight	Reduce el uso de memoria compartiendo partes repetidas entre objetos.	<pre> classDiagram class FlyweightFactory { pool : Map getFlyweight(key) } class Flyweight { operation(extrinsicState) } FlyweightFactory "2" --> Flyweight Flyweight "2" --> FlyweightFactory </pre>	<pre> class Fly { constructor(s) { this.s = s; } } </pre>

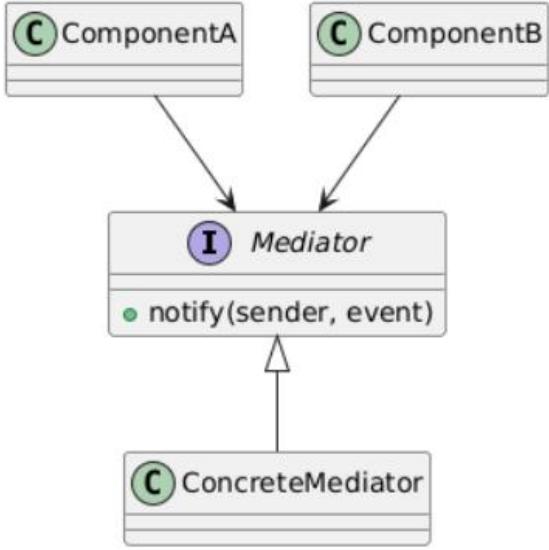
Patrones de Comportamiento

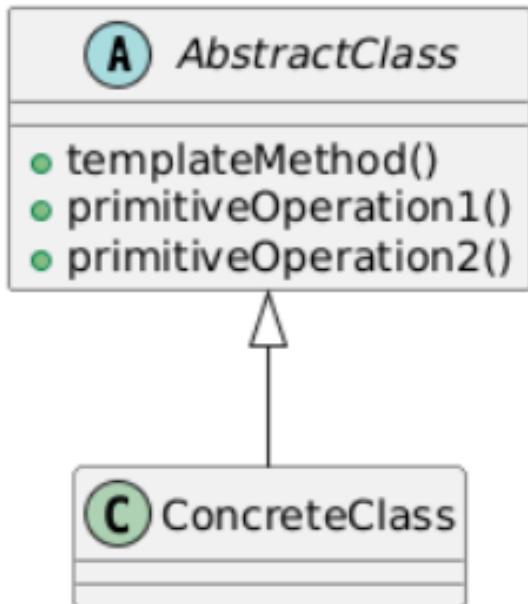
Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
Observer	Varios objetos se enteran automáticamente cuando otro cambia.	 <pre> classDiagram class Subject { <<C>> List<Observer> observers attach(Observer) detach(Observer) notify() } class Observer { <<I>> update() } class ConcreteObserver Subject "1" -- "*" Observer : notifica * ConcreteObserver "1" --> Observer : update() </pre> <p>The UML diagram illustrates the Observer pattern. It features three classes: Subject, Observer, and ConcreteObserver. The Subject class contains a list of Observers and provides methods for attaching, detaching, and notifying them. The Observer interface defines a update() method. The ConcreteObserver class implements the Observer interface. A dependency arrow labeled notifica * points from Subject to Observer, and another arrow points from ConcreteObserver to Observer.</p>	<pre> - class Sub { constructor() { this.obs = []; } add(o) { this.obs.push(o); } notify(d) { this.obs.forEach(o => o.update(d)); } } </pre>

Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
Strategy	<p>Permite cambiar la forma de hacer algo sin modificar el objeto principal.</p>	 <pre> classDiagram class Context { strategy : Strategy executeStrategy() } class Strategy { execute() } class ConcreteStrategyA class ConcreteStrategyB Context "2" --> "1" Strategy ConcreteStrategyA --> "1" Strategy ConcreteStrategyB --> "1" Strategy </pre>	<pre> - class Ctx { set(s) { this.s = s; } run() { return this.s.exec(); } } </pre>

Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
State	<p>Cambia el comportamiento de un objeto según la situación en la que se encuentre.</p>	 <pre> classDiagram class Context { state : State setState(State) } class State { handle() } class ConcreteStateA class ConcreteStateB Context "1" -- "1" State State "1" -- "2" ConcreteStateA State "1" -- "2" ConcreteStateB </pre>	<pre> - class StateA { handle() { return "A"; } } </pre>

Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
Command	<p>Convierte acciones en objetos para poder guardarlas, deshacerlas o enviarlas.</p>	<pre> classDiagram class Invoker { <<C>> command : Command executeCommand() } class Command { <<I>> execute() } class ConcreteCommand { <<C>> Receiver action() } class Receiver { <<C>> action() } Invoker --> Command Command < -- ConcreteCommand ConcreteCommand --> Receiver </pre>	<pre> - class Cmd { exec() {} } </pre>

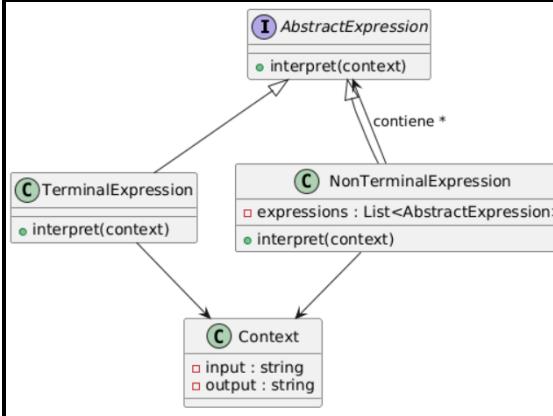
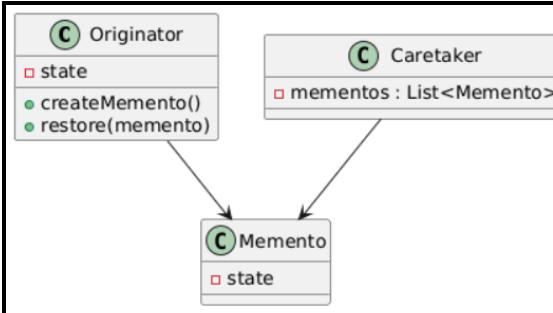
Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
Mediator	Hace que varios objetos se comuniquen sin hablar entre ellos directamente.	 <pre> classDiagram class ComponentA class ComponentB class Mediator { <<I>> <<notify(sender, event)>> } class ConcreteMediator { <<Mediator>> } ComponentA --> Mediator ComponentB --> Mediator Mediator --> ConcreteMediator </pre>	<pre> - class Mediator { send(msg, to) { to.receive(msg); } } </pre>

Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
Template Method	Marca un “paso a paso” general y deja que cada clase cambie los detalles.	 <pre> classDiagram class AbstractClass { <<A>> templateMethod() primitiveOperation1() primitiveOperation2() } class ConcreteClass { <<C>> } AbstractClass < -- ConcreteClass ConcreteClass --> AbstractClass </pre>	<pre> - class T { run() { this.a(); this.b(); } } </pre>

Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
Chain of Responsibility	Pasa una petición por varios objetos hasta que uno pueda atenderla.	<pre> classDiagram class Handler { Handler next; void setNext(Handler n); void handleRequest(); } class ConcreteHandlerA { // Implementation } class ConcreteHandlerB { // Implementation } Handler < -- ConcreteHandlerA Handler < -- ConcreteHandlerB </pre>	<pre> - class H { setNext(n) { this.n = n; } handle(r) { if (this.n) this.n.handle(r); } } </pre>

Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
Visitor	<p>Permite agregar funciones nuevas sin modificar las clases originales.</p>	<pre> classDiagram class Element { <<I>> accept(Visitor) } class ElementA { <<C>> } class ElementB { <<C>> } class Visitor { <<I>> visit(ElementA) visit(ElementB) } class ConcreteVisitor { <<C>> } Element < -- ElementA Element < -- ElementB Visitor < -- ConcreteVisitor Element --> Visitor : accept(Visitor) ElementA --> Visitor : visit(ElementA) ElementB --> Visitor : visit(ElementB) </pre>	<pre> - class Visitor { visit(obj) { obj.accept(this); } } </pre>

Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
Iterator	Da una forma sencilla de recorrer elementos uno por uno.	<pre> classDiagram class Aggregate { <<I>> createliterator() : Iterator } class Iterator { <<I>> next() <<I>> hasNext() } class ConcreteAggregate class Concreteliterator Concreteliterator --> ConcreteAggregate ConcreteAggregate --> Iterator </pre> <p>The diagram illustrates the Iterator pattern's UML structure. It features four classes: Aggregate, Iterator, ConcreteAggregate, and Concreteliterator. Aggregate has a private attribute <code>createliterator() : Iterator</code>. Iterator defines methods <code>next()</code> and <code>hasNext()</code>. ConcreteAggregate and Concreteliterator are associated with each other via directed associations, indicating that Concreteliterator iterates over ConcreteAggregate.</p>	<pre>- for (const i of [1,2,3]) console.log(i);</pre>

Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
Interpretar	Permite crear reglas para entender y ejecutar un pequeño lenguaje.	 <pre> classDiagram class AbstractExpression { <<I>> +interpret(context) } class TerminalExpression { +interpret(context) } class NonTerminalExpression { -expressions : List<AbstractExpression> +interpret(context) } class Context { -input : string -output : string } TerminalExpression --> AbstractExpression NonTerminalExpression --> AbstractExpression NonTerminalExpression "3..4" --> Context : contiene * </pre>	<pre> - class Expr { interp() {} } </pre>
Memento	Guarda y recupera estados anteriores de un objeto.	 <pre> classDiagram class Originator { -state +createMemento() +restore(memento) } class Caretaker { -mementos : List<Memento> } class Memento { -state } Originator --> Memento Caretaker --> Memento </pre>	<pre> - class Origin { save() { return { ...this }; } } </pre>