

Patrones de Diseño Arquitectónico

Aprendiz:

Juan David Buitrago Suarez

Instructor:

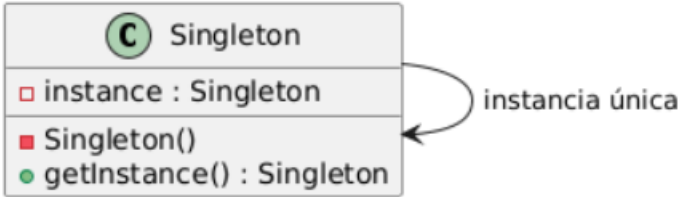
Ing. Néstor Montaña

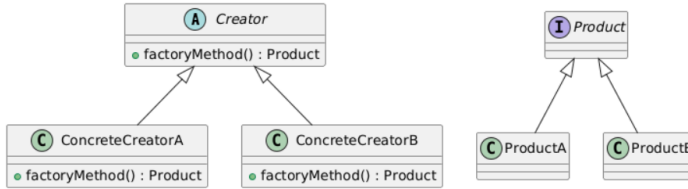
Tecnólogo Análisis y Desarrollo de Software

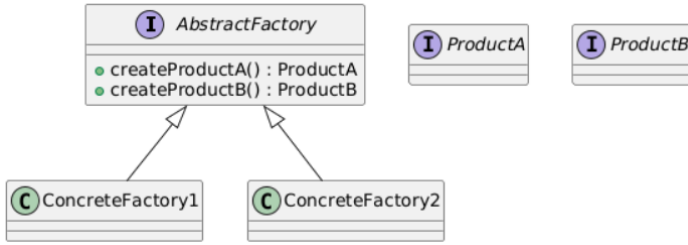
Ficha: 3064241

SENA Centro de Diseño y Metrología

## Patrones Creacionales

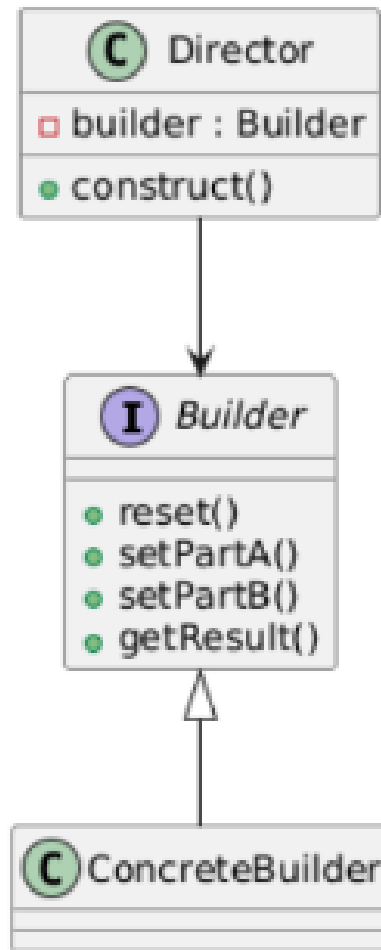
Nombre	Definición	Ejemplo en UML	Implementación en Node.js
Singleton	Asegura que solo exista un objeto de ese tipo y que todos usen el mismo.		<pre> - class Singleton {   constructor() {     if (Singleton.instance) return Singleton.instance;     Singleton.instance = this;   } }  const a = new Singleton(); const b = new Singleton(); console.log(a === b); // true </pre>

Nombre	Definición	Ejemplo en UML	Implementación en Node.js
<b>Factory Method</b>	Permite crear diferentes objetos sin decir exactamente cuál se va a crear.	 <pre> classDiagram     class Creator {         +factoryMethod() Product     }     class ConcreteCreatorA {         +factoryMethod() Product     }     class ConcreteCreatorB {         +factoryMethod() Product     }     class Product {     }     class ProductA {     }     class ProductB {     }     Creator &lt; -- ConcreteCreatorA     Creator &lt; -- ConcreteCreatorB     Product &lt; -- ProductA     Product &lt; -- 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>

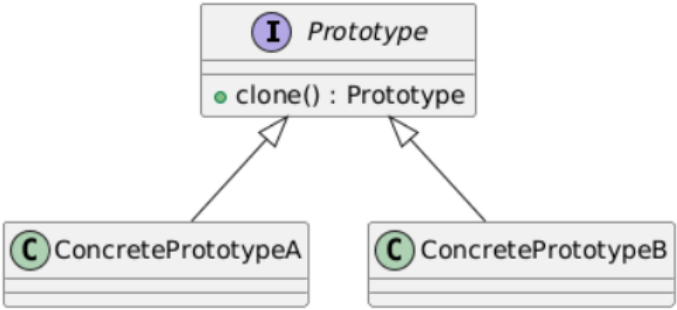
Nombre	Definición	Ejemplo en UML	Implementación en Node.js
<b>Abstract Factory</b>	Crea varios tipos de objetos que combinan bien entre sí sin mostrar cómo se hacen.	 <pre> classDiagram     class AbstractFactory {         &lt;&lt;abstract&gt;&gt;         +createProductA() ProductA         +createProductB() ProductB     }     class ProductA     class ProductB     class ConcreteFactory1     class ConcreteFactory2     AbstractFactory &lt; -- ConcreteFactory1     AbstractFactory &lt; -- ConcreteFactory2     </pre> <p>The UML diagram illustrates the Abstract Factory pattern. It features an abstract class <b>AbstractFactory</b> with two methods: <code>createProductA() : ProductA</code> and <code>createProductB() : ProductB</code>. Two concrete classes, <b>ConcreteFactory1</b> and <b>ConcreteFactory2</b>, inherit from <b>AbstractFactory</b>, as indicated by solid arrows. Additionally, there are two interface classes, <b>ProductA</b> and <b>ProductB</b>, which define the types of products created by the factories.</p>	<pre> - class ButtonWindows {     draw() { return "Botón Windows"; } } class MenuWindows {     show() { return "Menú Windows"; } } class UIFactoryWindows {     createButton() { return new ButtonWindows(); }     createMenu() { return new MenuWindows(); } } </pre>

**Builder**

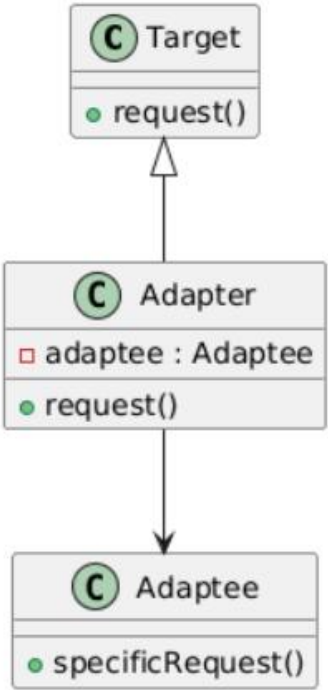
Ayuda a armar  
objetos paso a paso  
para no confundir la  
construcción con el  
resultado final.

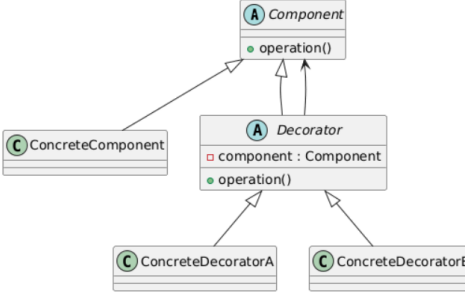
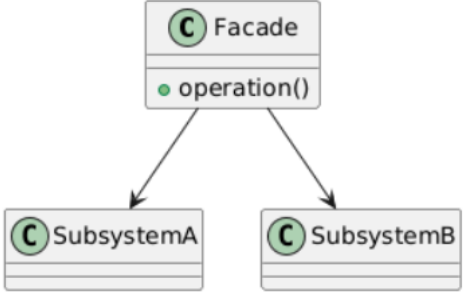


```
- class Builder {  
  constructor() {  
    this.data = {};  
  }  
  setA(v) { this.data.A = v; return this; }  
  setB(v) { this.data.B = v; return this; }  
  build() { return this.data; }  
}
```

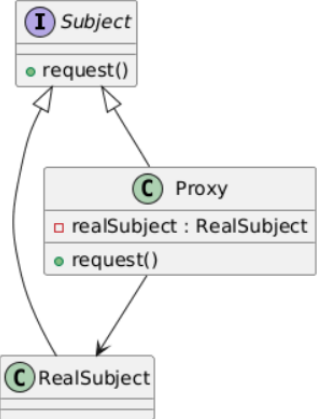
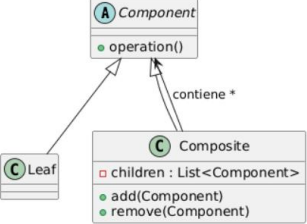
Nombre	Definición	Ejemplo en UML	Implementación en Node.js
<b>Prototype</b>	Permite copiar un objeto ya hecho para crear otro igual sin hacerlo desde cero.	 <pre>classDiagram     class Prototype {         &lt;&lt;interface&gt;&gt;         +clone() Prototype     }     class ConcretePrototypeA     class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

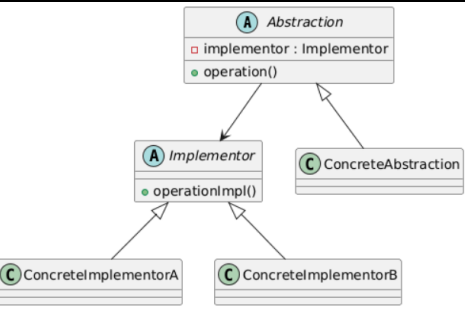
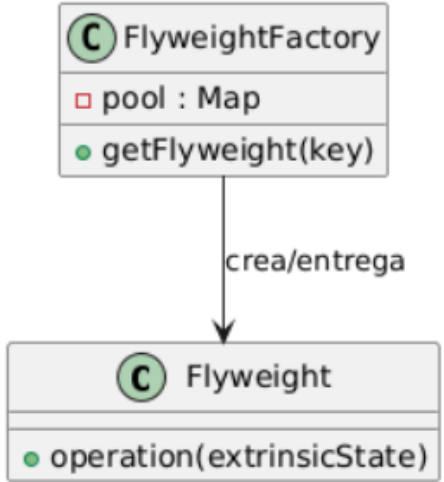
### Patrones Estructurales

Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
<b>Adapter</b>	Hace que dos cosas que no son compatibles puedan funcionar juntas.	 <pre> classDiagram     class Target {         +request()     }     class Adapter {         +adaptee : Adaptee         +request()     }     class Adaptee {         +specificRequest()     }     Target &lt; -- Adapter     Adapter --&gt; Adaptee         </pre> <p>The UML diagram illustrates the Adapter Pattern. It features three classes: <b>Target</b>, <b>Adapter</b>, and <b>Adaptee</b>. <b>Target</b> has a <code>request()</code> method. <b>Adapter</b> inherits from <b>Target</b> (indicated by a hollow triangle arrow) and implements the <code>request()</code> method. <b>Adapter</b> also has a private attribute <code>adaptee : Adaptee</code>. <b>Adaptee</b> has a <code>specificRequest()</code> method. A solid arrow points from <b>Adapter</b> to <b>Adaptee</b>, showing that the Adapter holds a reference to the Adaptee to delegate the request.</p>	<pre> - class Old {   specific() { return "Función antigua"; } }  class Adapter {   request() { return new Old().specific(); } }         </pre>

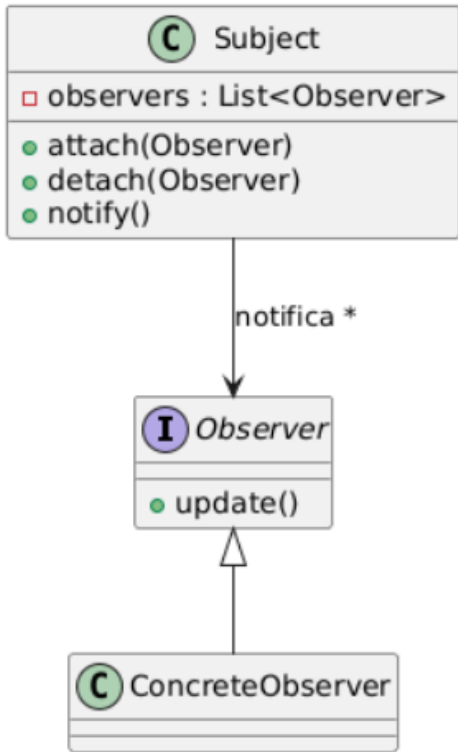
Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
<b>Decorator</b>	Permite agregar nuevas funciones a un objeto sin cambiarlo por dentro.	 <pre> classDiagram     class Component {         +operation()     }     class ConcreteComponent {     }     class Decorator {         +component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

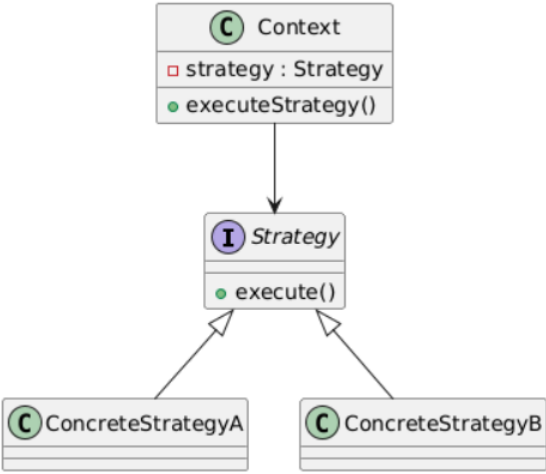


Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
<b>Proxy</b>	Es un intermediario que controla el acceso a otro objeto.	 <pre> classDiagram     class Subject {         &lt;&lt;interface&gt;&gt;         request()     }     class Proxy {         request()         realSubject : RealSubject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

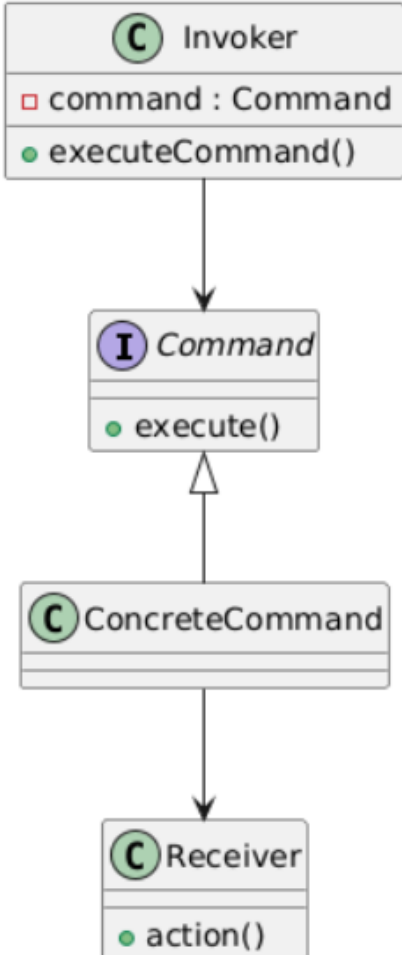
Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
<b>Bridge</b>	Separa lo que hace una clase de cómo está hecha para que sean más flexibles.	 <pre> classDiagram     class Abstraction {         +constructor         +operation()     }     class Implementor {         +operationImpl()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

### Patrones de Comportamiento

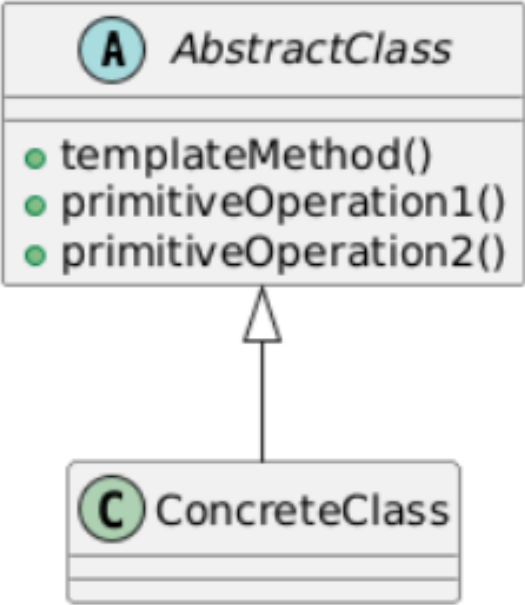
Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
<b>Observer</b>	Varios objetos se enteran automáticamente cuando otro cambia.	 <pre> classDiagram     class Subject {         +List&lt;Observer&gt; observers         +attach(Observer)         +detach(Observer)         +notify()     }     class Observer {         &lt;&lt;interface&gt;&gt;         +update()     }     class ConcreteObserver     Observer &lt; -- ConcreteObserver     Subject --&gt; Observer : notifica * </pre>	<pre> class Sub {   constructor() { this.obs = []; }   add(o) { this.obs.push(o); }   notify(d) { this.obs.forEach(o =&gt; o.update(d)); } } </pre>

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

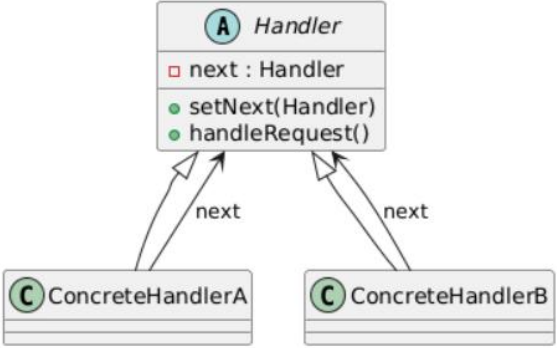
Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
State	Cambia el comportamiento de un objeto según la situación en la que se encuentre.	<pre>classDiagram     class Context {         +State state         +setState(State)     }     class State {         +handle()     }     class ConcreteStateA     class ConcreteStateB     Context --&gt; State     State &lt; -- ConcreteStateA     State &lt; -- ConcreteStateB</pre> <p>The UML diagram illustrates the State Design Pattern. It features three classes: <b>Context</b>, <b>State</b>, and two concrete classes, <b>ConcreteStateA</b> and <b>ConcreteStateB</b>. The <b>Context</b> class (marked with a green circle 'C') contains a <b>state</b> attribute of type <b>State</b> (marked with a red square) and a <b>setState(State)</b> method (marked with a green dot). An arrow points from <b>Context</b> to <b>State</b>. The <b>State</b> class (marked with a purple circle 'I') defines a <b>handle()</b> method (marked with a green dot). Both <b>ConcreteStateA</b> and <b>ConcreteStateB</b> (both marked with green circles 'C') inherit from <b>State</b>, as indicated by hollow triangle arrows pointing to the <b>State</b> class.</p>	<pre>- class StateA {   handle() { return "A"; } }</pre>

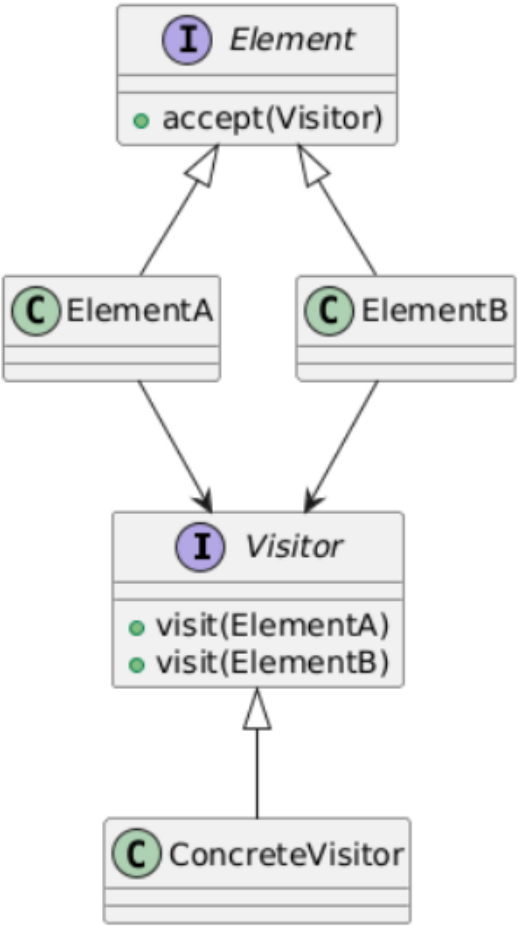
Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
<b>Command</b>	Convierte acciones en objetos para poder guardarlas, deshacerlas o enviarlas.	 <pre> classDiagram     class Invoker {         +Command command         +executeCommand()     }     class Command {         +execute()     }     class ConcreteCommand     class Receiver {         +action()     }     Invoker --&gt; Command     Command &lt; -- ConcreteCommand     ConcreteCommand --&gt; Receiver </pre> <p>The UML diagram illustrates the Command Pattern structure. It features four classes: <b>Invoker</b>, <b>Command</b>, <b>ConcreteCommand</b>, and <b>Receiver</b>. The <b>Invoker</b> class contains a <b>command</b> attribute of type <b>Command</b> and an <b>executeCommand()</b> method. It has a directed association to the <b>Command</b> interface. The <b>Command</b> interface defines an <b>execute()</b> method. <b>ConcreteCommand</b> is a concrete implementation that inherits from <b>Command</b> (indicated by a hollow triangle) and holds a reference to a <b>Receiver</b> object. The <b>Receiver</b> class implements the <b>action()</b> method that <b>ConcreteCommand</b> delegates to.</p>	<pre> - class Cmd {   exec() {} } </pre>

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

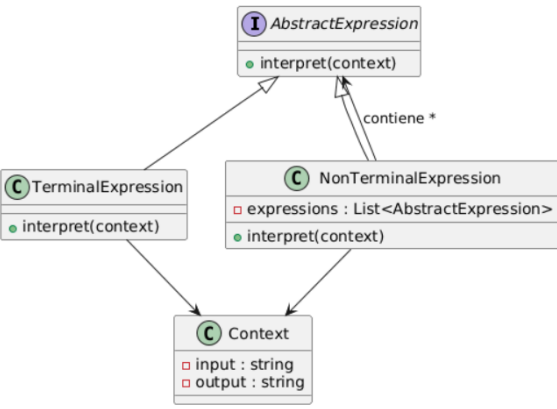
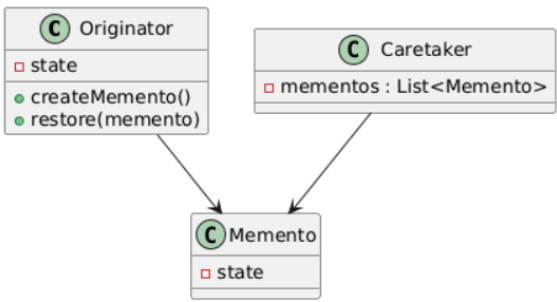
Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
Template Method	Marca un “paso a paso” general y deja que cada clase cambie los detalles.	 <pre>classDiagram     class AbstractClass {         &lt;&lt;abstract&gt;&gt;         +templateMethod()         +primitiveOperation1()         +primitiveOperation2()     }     class ConcreteClass     AbstractClass &lt; -- ConcreteClass</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 {         &lt;&lt;abstract&gt;&gt;         +next : Handler         +setNext(Handler)         +handleRequest()     }     class ConcreteHandlerA     class ConcreteHandlerB     Handler &lt; -- ConcreteHandlerA     Handler &lt; -- 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
<b>Visitor</b>	Permite agregar funciones nuevas sin modificar las clases originales.	 <pre>classDiagram     class Element {         &lt;&lt;interface&gt;&gt;         +accept(Visitor)     }     class ElementA {         &lt;&lt;concrete&gt;&gt;     }     class ElementB {         &lt;&lt;concrete&gt;&gt;     }     class Visitor {         &lt;&lt;interface&gt;&gt;         +visit(ElementA)         +visit(ElementB)     }     class ConcreteVisitor {         &lt;&lt;concrete&gt;&gt;     }     Element &lt; -- ElementA     Element &lt; -- ElementB     Visitor &lt; -- ConcreteVisitor     ElementA --&gt; Visitor     ElementB --&gt; Visitor</pre>	<pre>- class Visitor {   visit(obj) { obj.accept(this); } }</pre>

Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
<b>Iterator</b>	Da una forma sencilla de recorrer elementos uno por uno.	<pre>classDiagram     class Aggregate {         &lt;&lt;interface&gt;&gt;         +createIterator() Iterator     }     class Iterator {         &lt;&lt;interface&gt;&gt;         +next()         +hasNext()     }     class ConcreteAggregate     class ConcreteIterator     Aggregate &lt; -- ConcreteAggregate     Iterator &lt; -- ConcreteIterator     ConcreteAggregate --&gt; ConcreteIterator</pre>	<pre>- for (const i of [1,2,3]) console.log(i);</pre>

Nombre del Patrón	Definición	Ejemplo en UML	Implementación en Node.js
<b>Interpreter</b>	Permite crear reglas para entender y ejecutar un pequeño lenguaje.	 <pre> classDiagram     class AbstractExpression {         &lt;&lt;abstract&gt;&gt;         +interpret(context)     }     class TerminalExpression {         +interpret(context)     }     class NonTerminalExpression {         +expressions : List&lt;AbstractExpression&gt;         +interpret(context)     }     class Context {         +input : string         +output : string     }     AbstractExpression &lt; -- TerminalExpression     AbstractExpression &lt; -- NonTerminalExpression     NonTerminalExpression "1" -- "*" AbstractExpression : contiene     TerminalExpression --&gt; Context     NonTerminalExpression --&gt; Context </pre>	<pre> - class Expr {   interp() {} } </pre>
<b>Memento</b>	Guarda y recupera estados anteriores de un objeto.	 <pre> classDiagram     class Originator {         +state         +createMemento()         +restore(memento)     }     class Caretaker {         +mementos : List&lt;Memento&gt;     }     class Memento {         +state     }     Originator --&gt; Memento     Caretaker --&gt; Memento </pre>	<pre> - class Origin {   save() { return { ...this }; } } </pre>