

Patrones Arquitectónicos

Aprendices del SENA

Julian David Vergara Ramirez

Instructor

Ing Nestor Montalo

Tecnólogo de Análisis y Desarrollo de Software

Centro de Diseño y Metrología - SENA

Ficha: 3064241

Bogotá D.C

Patrones creacionales

Nombre del Patrón	Definición	Clasificación	Ejemplo en UML	Implementación en JavaScript
<i>singleton</i>	Garantiza que una clase tenga una sola instancia y proporciona un acceso global a ella.	Patrón Creacional	<p>Diagrama UML: patrón singleton</p> <pre> classDiagram class Singleton { -instance : Singleton +getInstance() : Singleton -Singleton() : void } Singleton "1" --> "1" Singleton : </pre> <p>IONOS</p>	<pre> class Singleton { constructor() { if (Singleton.instancia) { return Singleton.instancia; } this.valor = "Única instancia"; Singleton.instancia = this; } } const a = new Singleton(); const b = new Singleton(); console.log(a === b); // true </pre>
Factory Method	Define una interfaz para crear objetos, dejando que las subclases decidan qué objeto crear.	Patrón Creacional	<pre> classDiagram class Product class Creator { +FactoryMethod() --> Product +AnOperation() } class ConcreteProduct { < -- ConcreteCreator } class ConcreteCreator { +FactoryMethod() --> ConcreteProduct } </pre>	<pre> class Creador { factoryMethod() {} } class AutoCreator extends Creador { factoryMethod() { return new Auto(); } } class Auto { manejar() { console.log("Conduciendo auto..."); } } const creador = new AutoCreator(); const auto = creador.factoryMethod(); auto.manejar(); </pre>

Abstract Factory	Crea familias de objetos relacionados sin especificar sus clases concretas.	Creacional	<pre> classDiagram class Client { <<use>> <<use>> } class AbstractFactory { <<interface>> +createProductA(): ProductA +createProductB(): ProductB } class ConcreteFactoryX { <<ConcreteFactory>> +createProductA(): ProductA +createProductB(): ProductB } class ConcreteFactoryY { <<ConcreteFactory>> +createProductA(): ProductA +createProductB(): ProductB } class ProductA { <<interface>> } class ProductB { <<interface>> } class ProductAX { <<ProductA>> } class ProductAY { <<ProductA>> } class ProductBX { <<ProductB>> } class ProductBY { <<ProductB>> } Client --> AbstractFactory : <<use>> Client --> AbstractFactory : <<use>> AbstractFactory --> ConcreteFactoryX : <<create>> AbstractFactory --> ConcreteFactoryY : <<create>> ConcreteFactoryX --> ProductAX : <<create>> ConcreteFactoryX --> ProductAY : <<create>> ConcreteFactoryY --> ProductBX : <<create>> ConcreteFactoryY --> ProductBY : <<create>> </pre>	<pre> class AbstractFactory { crearBoton() {} } class FactoryWindows extends AbstractFactory { crearBoton() { return new BotonWindows(); } } class BotonWindows { render() { console.log("Botón Windows"); } } </pre>
Builder	Permite construir objetos complejos paso a paso.	Creacional	<p>UML Class Diagram for Builder Design Pattern</p> <pre> classDiagram class ComputerDirector { <<ComputerDirector>> +construct(builder: Builder) } class Computer { <<Computer>> -cpu: string -ram: string -storage: string -gpu: string -os: string -monitor: string -keyboard: string -mouse: string -displayPort: string } class ComputerBuilder { <<ComputerBuilder>> +computer: Computer +addCPU(string m): void +addRAM(string m): void +addStorage(string m): void +addGPU(string m): void +addOS(string m): void +addMonitor(string m): void +addKeyboard(string m): void +addMouse(string m): void +setDisplayPort(string m): void +getComputer(): Computer } class Director { <<Director>> +construct(builder: ComputerBuilder) } class Builder { <<Builder>> +interact(ComputerDirector director) +buildCPU(): void +addCPU(string m): void +addRAM(string m): void +addStorage(string m): void +addGPU(string m): void +addOS(string m): void +addMonitor(string m): void +addKeyboard(string m): void +addMouse(string m): void +setDisplayPort(string m): void +getComputer(): Computer } </pre>	<pre> class BuilderPizza { reset() { this.pizza = {}; } setMasa(m) { this.pizza.masa = m; } setQueso(q) { this.pizza.queso = q; } build() { return this.pizza; } } const b = new BuilderPizza(); b.reset(); b.setMasa("gruesa"); b.setQueso("mozzarella"); console.log(b.build()); </pre>
Prototype	Crea nuevos objetos copiando un objeto existente (prototipo).	Creacional	<p>UML Class Diagram for Prototype Design Pattern</p> <pre> classDiagram class Shape { -id: String +type: String +getShape(): void +getId(): String +setId(id: String): void +clone(): Object } class Circle { -type: String +getShape(): void +getId(): String +setId(id: String): void +clone(): Object } class Rectangle { -type: String +getShape(): void +getId(): String +setId(id: String): void +clone(): Object } class Square { -type: String +getShape(): void +getId(): String +setId(id: String): void +clone(): Object } class PrototypePatternDemo { +main(): void } class ShapeCache { -shapeMap: HashMap +getShape(): Shape +loadCache(): void } Shape --> PrototypePatternDemo : asks PrototypePatternDemo --> ShapeCache : clones Shape --> Circle : extends Shape --> Rectangle : extends Shape --> Square : extends </pre>	<pre> const personaje = { tipo: "Guerrero", clonar() { return { ...this }; } }; const copia = personaje.clonar(); </pre>

Patrones de Estructurales

Nombre del Patrón	Definición	Clasificación	Ejemplo en UML	Implementación en JavaScript
Adapter	Convierte la interfaz de una clase en otra que el cliente espera.	Estructural	<pre> classDiagram class Client { <<Client>> } class Target { <<Target>> +Request() } class Adapter { <<Adapter>> +Request() +adaptee } class Adaptee { <<Adaptee>> +SpecificRequest() } Client --> Target : target Adapter --> Target : +Request() Adapter --> Adaptee : adaptee Adaptee --> Adaptee : +SpecificRequest() </pre>	<pre> class Adaptee { requestOld(){ return "Antiguo"; } } class Adapter { constructor(){ this.adaptee = new Adaptee(); } request(){ return this.adaptee.requestOld(); } } </pre>
Bridge	Separa una abstracción de su implementación para permitir variarlas independientemente.	Estructural	<pre> classDiagram class BridgePatternDemo { <<BridgePatternDemo>> +main() } class Shape { <<Shape>> +drawAPI : DrawAPI +Shape() +draw() } class DrawAPI { <<Interface>> +drawCircle() } class Circle { <<Circle>> -x, y, radius : int +Circle() +draw() } class RedCircle { <<RedCircle>> +drawCircle() } class GreenCircle { <<GreenCircle>> +drawCircle() } BridgePatternDemo --> Shape : uses Shape --> DrawAPI : uses Shape --> Circle : extends DrawAPI --> RedCircle : implement DrawAPI --> GreenCircle : implement Circle --> RedCircle : uses Circle --> GreenCircle : uses </pre>	<pre> class Dispositivo { constructor(implementacion){ this.imp = implementacion; } encender(){ this.imp.encender(); } } class TV { encender(){ console.log("TV encendida"); } } const tv = new Dispositivo(new TV()); tv.encender(); </pre>

Composite	Permite tratar objetos individuales y compuestos de forma uniforme.	Estructural	<pre> classDiagram class Order class Item class Customer class StandardCustomer class MemberCustomer class StoreManager Order "1..>" Item Order "1..>" Customer Customer "1..>" StoreManager Customer < -- StandardCustomer Customer < -- MemberCustomer </pre>	<pre> class Nodo { mostrar() {} } class Hoja extends Nodo { mostrar(){ console.log("Hoja"); } } class Compuesto extends Nodo { constructor(){ super(); } this.hijos=[]; agregar(n){ this.hijos.push(n); } mostrar(){ this.hijos.forEach(h => h.mostrar()); } } </pre>
Decorator	Añade responsabilidades adicionales dinámicamente a un objeto.	Estructural	<pre> classDiagram class Component { +Operation() } class ConcreteComponent { +Operation() } class Decorator { +Operation() } class ConcreteDecoratorA { -addedState +Operation() } class ConcreteDecoratorB { -addedState +Operation() +AddedBehavior() } Component < -- ConcreteComponent Component < -- Decorator Decorator < -- ConcreteDecoratorA Decorator < -- ConcreteDecoratorB ConcreteComponent --> component : Operation() ConcreteDecoratorA --> component : Operation() ConcreteDecoratorB --> component : Operation() ConcreteDecoratorB --> AddedBehavior() </pre>	<pre> class Cafe { costo(){ return 5; } } class ConLeche { constructor(cafe){ this.cafe=cafe; } costo(){ return this.cafe.costo() + 2; } } </pre>
Facade	Proporciona una interfaz simplificada a un sistema complejo.	Estructural	<pre> classDiagram class Shape { +draw() } class Circle { +draw() } class Rectangle { +draw() } class Square { +draw() } class FacadePatternDemo { +main() } class ShapeMaker { -circle : Shape -rectangle : Shape -square : Shape +ShapeMaker() +drawCircle() +drawRectangle() +drawSquare() } Shape <<Interface>> Shape < -- Circle Shape < -- Rectangle Shape < -- Square Circle --> implements : draw() Rectangle --> implements : draw() Square --> implements : draw() FacadePatternDemo --> creates : ShapeMaker FacadePatternDemo --> asks : draw() </pre>	<pre> class Subsistema { operacion(){ return "Operación interna"; } } class Facade { constructor(){ this.s = new Subsistema(); } operar(){ return this.s.operacion(); } } </pre>

Nombre del Patrón	Definición	Clasificación	Ejemplo en UML	Implementación en JavaScript
Flyweight	Minimiza el uso de memoria compartiendo objetos de bajo peso.	Estructural	<pre> classDiagram class FlyweightFactory { +get(key: Object) : Flyweight } interface Flyweight { +operation(extrinsicData : Object) } class ConcreteFlyweight { -intrinsicState : Object +operation(extrinsicData : Object) } class Client { <<Client>> } Client --> FlyweightFactory Client --> ConcreteFlyweight FlyweightFactory --> Flyweight ConcreteFlyweight --> Flyweight </pre> <p>UML Diagram of Flyweight Design Pattern</p>	<pre> class Flyweight { constructor(color){ this.color=color; } class Factory { constructor(){ this.pool={ }; } get(color){ if(!this.pool[color]) this.pool[color] = new Flyweight(color); return this.pool[color]; } } </pre>
Proxy	Proporciona un sustituto que controla el acceso a otro objeto.	Estructural	<pre> classDiagram interface Image { +display() : void } class RealImage { +fileName : String +RealImage() +display() : void +loadFromDisk() : void } class ProxyImage { +realImage : RealImage +fileName : String +ProxyImage() +display() : void } class ProxyPatternDemo { +main() : void } Image <<Image>> RealImage implements Image ProxyImage implements Image ProxyPatternDemo --> ProxyImage ProxyImage --> RealImage </pre>	<pre> class ServicioReal { request(){ return "Datos reales"; } class ProxyServicio { constructor(){ this.real = new ServicioReal(); } request(){ return this.real.request(); } } </pre>

Patrones de comportamiento

Nombre del Patrón	Definición	Clasificación	Ejemplo en UML	Implementación en JavaScript
Chain of Responsibility	Permite pasar una petición por una cadena de receptores hasta que uno la procese.	Comportamiento	<pre> sequenceDiagram participant Consojeria participant Encargado participant Director Consojeria->>Encargado: activate Encargado Encargado->>Director: activate Director Director-->>Reserva: activate Reserva Note over Reserva: self->>self: setNext(h) self->>self: handle(req) self-->>Reserva: deactivate Director deactivate Encargado deactivate Consojeria </pre>	<pre> class Handler { setNext(h){ this.next=h; } handle(req){ if(this.next) this.next.handle(req); } } </pre>
Command	Encapsula una petición como un objeto.	Comportamiento	<pre> classDiagram class Client class Invoker class Receiver class Command Client --> Invoker Invoker --> Receiver Invoker --> Command Receiver --> Command Command --> Receiver </pre>	<pre> class EncenderLuz { ejecutar(){ console.log("Luz encendida"); } } </pre>

Interpreter	Define una gramática y un intérprete para evaluar expresiones.	Comportamiento	<pre> classDiagram class Client class Context class AbstractExpression { +Interpret(c : Context) } class TerminalExpression { +Interpret(c : Context) } class NonterminalExpression { +Interpret(c : Context) } Client --> AbstractExpression : Context --> AbstractExpression : TerminalExpression --> AbstractExpression : NonterminalExpression --> AbstractExpression : </pre>	<pre> class Numero { constructor(v){ this.v=v; } interpretar(){ return this.v; } } </pre>
Iterator	Permite recorrer elementos de una colección sin exponer su estructura interna.	Comportamiento	<pre> classDiagram class Client class Aggregate { +CreateIterator() : Iterator } class Iterator { +First() +Next() +IsDone() +CurrentItem() } <<PTN Cloneable>> ConcreteAggregate <<PTN Cloneable>> ConcreteIterator Client --> Aggregate : Client --> Iterator : Aggregate --> ConcreteAggregate : Iterator --> ConcreteIterator : </pre>	<pre> const lista = [1,2,3]; for (let n of lista) console.log(n); </pre>
Mediator	Centraliza la comunicación entre objetos para reducir dependencias.	Comportamiento	<pre> classDiagram class Mediator class ConcreteMediator class Colleague class ConcreteColleague1 class ConcreteColleague2 Mediator < -- ConcreteMediator Mediator < -- Colleague Colleague < -- ConcreteColleague1 Colleague < -- ConcreteColleague2 ConcreteMediator --> ConcreteColleague1 ConcreteMediator --> ConcreteColleague2 </pre>	<pre> class Chat { enviar(msg, usuario){ console.log(`\${ usuario}: \${msg}`); } } </pre>

Nombre del Patrón	Definición	Clasificación	Ejemplo en UML	Implementación en JavaScript
Memento	Guarda y restaura el estado interno de un objeto.	Comportamiento	<pre> classDiagram class Originator { state +SetMemento(m : Memento) +CreateMemento() } class Memento { state +GetState() +SetState() } class Caretaker Originator "2..>" Memento Originator --> Caretaker : memento Memento --> Caretaker : memento Note over Originator, Memento: state = m.GetState(); Note over Originator: +return new Memento(state); </pre>	<pre> class Memento { constructor(estado){ this.estado=estado; } } </pre>
Observer	Actualiza automáticamente a los observadores cuando el sujeto cambia.	Comportamiento	<pre> classDiagram class <<PTN Members Creatable>> Subject { +Attach(o : Observer) +Detach(o : Observer) +Notify() } class <<PTN Members Creatable>> Observer { +Update() } class <<PTN Cloneable>> ConcreteSubject { -subjectstate +GetState() +SetState(state) } class <<PTN Cloneable>> ConcreteObserver { -observerstate +Update() } Subject < -- ConcreteSubject Observer < -- ConcreteObserver ConcreteSubject --> Observer : update() </pre>	<pre> class Subject { constructor(){ this.obs=[]; } attach(o){ this.obs.push(o); } notify(v){ this.obs.forEach(o=>o.update(v)); } } </pre>
State	Permite cambiar el comportamiento de un objeto según su estado interno.	Comportamiento	<pre> stateDiagram-v2 [*] --> Opened : Create/ Opened --> Closed : Close/[doorWay->isEmpty] Opened --> Locked : Open/ Closed --> Locked : Lock/ Locked --> Closed : Unlock/ </pre>	<pre> class EstadoA { manejar(){ console.log("Estado A"); } } </pre>

Strategy	Define algoritmos intercambiables.	Comportamiento	<pre> classDiagram class Context class Strategy { <<interface>> +execute() } class ConcreteStrategyA { +execute() } class ConcreteStrategyB { +execute() } Context "1" -- "1" Strategy Strategy < -- ConcreteStrategyA Strategy < -- ConcreteStrategyB </pre>	<pre> class A { ejecutar(){ console.log("A"); } } </pre>
Template Method	Define el esqueleto de un algoritmo, dejando pasos a subclases.	Comportamiento	<pre> classDiagram class Client class AbstractTemplate { +templateMethod() #step1() #step2() #step3() } class ImplementationA { #step1() #step2() #step3() } class ImplementationB { #step1() #step2() #step3() } Client --> AbstractTemplate AbstractTemplate < -- ImplementationA AbstractTemplate < -- ImplementationB </pre> <p style="text-align: center;"><i>Template Method – Class diagram</i></p>	<pre> class Receta { cocinar(){ this.preparar(); this.servir(); } } </pre>
Visitor	Permite agregar operaciones a objetos sin modificar sus clases.	Comportamiento	<pre> classDiagram class Client class Element { +accept(IVisitor) } class IVisitor { +visitElementA() +visitElementB() } class ConcreteElementA { +accept(IVisitor) } class ConcreteElementB { +accept(IVisitor) } class ConcreteVisitorA { +visitElementA() +visitElementB() } class ConcreteVisitorB { +visitElementA() +visitElementB() } Client "use" --> Element Client "use" --> IVisitor Element --> ConcreteElementA Element --> ConcreteElementB IVisitor < -- ConcreteVisitorA IVisitor < -- ConcreteVisitorB </pre> <p style="text-align: center;"><i>Visitor pattern – Class diagram</i></p>	<pre> class Visitor { visitar(e){ e.aceptar(this); } } </pre>