

Patrones de Diseño Arquitectónico

Aprendiz:

Kevin David Lopez Delgado

Instructor:

Ing. Néstor Montaño

Tecnólogo Análisis y Desarrollo de Software

Ficha: 3064241

SENA Centro de Diseño y Metrología

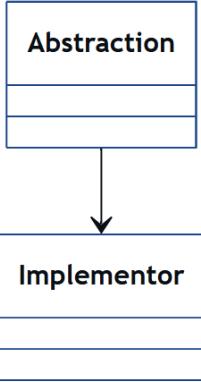
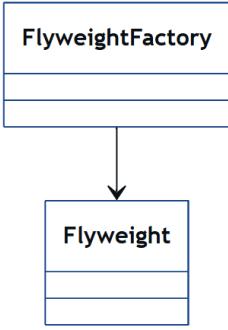
Patrones Creacionales

Nombr e del Patrón	Definició n	Clasificac ión	Ejemplo UML	Implementación en JavaScript
Factory Method	Define un método encargado de crear objetos, permitiendo que las subclases decidan qué tipo concreto instanciar, evitando acoplar el código cliente a clases específicas.	Creaciona l	<pre> classDiagram class Creator { +factoryMethod(type) } class ConcreteCreator { +factoryMethod(type) } class Product { +operation() } class ProductA { +operation() } class ProductB { +operation() } Creator --> Product ConcreteCreator --> Product ConcreteCreator --> ProductA ConcreteCreator --> ProductB ProductA --> ConcreteCreator </pre>	<pre> class Creator { factoryMethod(t) { return t === 'A' ? new A() : new B(); } } </pre>
Abstract Factory	Proporciona una interfaz para crear familias de objetos relacionados entre sí sin especificar sus clases concretas, garantizando que los productos creados sean compatibles.	Creacional	<pre> classDiagram class AbstractFactory { +createButton() +createMenu() } class WinFactory { +createButton() +createMenu() } class MacFactory { +createButton() +createMenu() } class Button { <<Abstract Class>> } class WinButton { <<Concrete Class>> } class MacButton { <<Concrete Class>> } class Menu { <<Abstract Class>> } class WinMenu { <<Concrete Class>> } class MacMenu { <<Concrete Class>> } AbstractFactory --> WinFactory AbstractFactory --> MacFactory WinFactory --> WinButton WinFactory --> WinMenu MacFactory --> MacButton MacFactory --> MacMenu </pre>	<pre> class UIFactory { createButton() {} createMenu() {} } </pre>

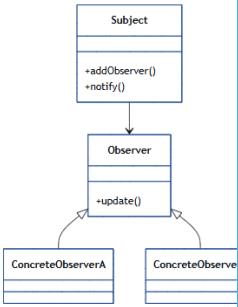
Singlet on	Restringe la creación de una clase a una única instancia y ofrece un acceso global a ella desde cualquier parte del programa.	Creacional	<pre> class Singleton { constructor() { if (Singleton.instance) return Singleton.instance; Singleton.instance = this; } } </pre>
Builder	Permite construir objetos complejos paso a paso sin depender directamente de su construcción interna.	Creacional	<pre> class Director { Director(); } class Builder { Director(); +buildPartA(); +buildPartB(); } class ConcreteBuilder { Builder(); +ConcreteBuilder(); } class Product { Product(); } </pre> <pre> class Builder { setA(a){ this.a = a; return this; } setB(b){ this.b = b; return this; } build(){ return { a: this.a, b: this.b }; } } </pre> <pre> const producto = new Builder().setA(1).setB(2). build(); </pre>
Prototy pe	Permite crear nuevos objetos clonando uno existente sin depender de su clase.	Creacional	<pre> class Prototype { Prototype(); +clone(); } class Objeto { Prototype(); } </pre> <pre> const prototipo = { nombre: "Objeto" }; const copia = Object.create(prototipo); </pre>

Patrones Estructurales

Nombre del Patrón	Definición	Clasificación	Ejemplo UML	Implementación en JavaScript
Adapter	Permite que interfaces incompatibles trabajen juntas convirtiendo la interfaz de una clase en otra que el cliente espera.	Estructural	<pre> classDiagram Cliente --> Adapter Adapter --> Old </pre>	<pre> class Adapter{ request(){ return new Old().specific() } } </pre>
Facade	Proporciona una interfaz más simple y unificada para un conjunto de interfaces complejas de un subsistema, facilitando su uso.	Estructural	<pre> classDiagram Facade --> A Facade --> B </pre>	<pre> class Facade{ start(){ new A().a(); new B().b(); } } </pre>
Proxy	Proporciona un representante o sustituto para controlar el acceso a un objeto real, pudiendo añadir lógica (caching, permisos, lazy load).	Estructural	<pre> classDiagram Cliente --> Proxy Proxy --> RealObject </pre>	<pre> const proxy = new Proxy(obj,{ get(t,p){ return t[p] } }) </pre>
Composite	Permite componer objetos en estructuras árbol para representar jerarquías parte-todo; trata a objetos individuales y compuestos de forma uniforme.	Estructural	<pre> classDiagram Component --> Hoja Component --> Composite </pre>	<pre> class Component{ add(){} operation(){} } </pre>

Bridge	Separa una abstracción de su implementación para que ambas puedan variar independientemente (evita combinaciones explosivas).	Estructural		<pre>class Abstraction{ constructor(im p){ this.imp = imp } run(){ this.imp.run() } }</pre>
Flyweight	Reduce el uso de memoria compartiendo (reutilizando) gran cantidad de objetos similares almacenando el estado intrínseco compartido.	Estructural		<pre>class Fly{ constructor(s){ this.s = s } }</pre>

Patrones de Comportamiento

Nombre del Patrón	Definición	Clasificación	Ejemplo UML	Implementación en JavaScript
Observer	Notifica automáticamente a los observadores cuando ocurre un cambio en el estado de un objeto.	Comportamiento		<pre>class Sub{ obs=[] notify(d){ this.obs.forEach(o=>o.up(d)) } } class Sub{ obs=[] notify(d){ this.obs.forEach(o=>o.up(d)) } }</pre>

Strategy	Permite cambiar algoritmos dinámicamente según la estrategia seleccionada.	Comportamiento	<pre> graph TD Context --> Strategy Strategy --> StrategyA Strategy --> StrategyB </pre>	<pre> class Ctx{ set(s){ this.s=s } run(){ return this.s.exec() } } </pre>
State	Permite cambiar el comportamiento de un objeto dependiendo de su estado interno.	Comportamiento	<pre> graph TD Context --> State State --> StateA State --> StateB </pre>	<pre> class StateA{ handle(){ return "A" } } </pre>
command	Convierte una solicitud en un objeto independiente que puede ejecutarse, deshacerse o almacenarse.	Comportamiento	<pre> graph TD Invoker --> Command Command --> ConcreteCommand ConcreteCommand --> Receiver </pre>	<pre> class Cmd{ exec(){} } </pre>
Mediator	Permite que los objetos se comuniquen entre sí mediante un mediador, reduciendo dependencias directas.	Comportamiento	<pre> graph TD Mediator <--> ColleagueA Mediator <--> ColleagueB </pre>	<pre> class Mediator{ send(msg,to){ to.receive(msg) } } </pre>
Template Method	Define la estructura de un algoritmo y deja que las subclases implementen pasos específicos.	Comportamiento	<pre> graph TD AbstractClass +template() ConcreteA --> AbstractClass ConcreteB --> AbstractClass </pre>	<pre> class T{ run(){ this.a(); this.b() } } </pre>

Chain of responsibility	Envía solicitudes a través de una cadena de objetos hasta que uno de ellos la procese.	Comportamiento	<pre> graph TD Handler --> HandlerA HandlerA --> HandlerB HandlerB --> Handler </pre>	<pre> class H{ setNext(n){ this.n=n } handle(r){ if(this.n) this.n.handle(r) } } </pre>
Visitor	Permite agregar nuevas operaciones a objetos sin modificar sus clases.	Comportamiento	<pre> graph TD Element --> ConcreteElement ConcreteElement --> Visitor Visitor --> ConcreteVisitor </pre>	<pre> class Visitor{ visit(obj){ obj.accept(this) } } </pre>
Iterator	Proporciona una forma estándar de recorrer una colección secuencialmente.	Comportamiento	<pre> graph TD Aggregate --> Iterator Iterator --> ConcretIterator </pre>	<pre> for(const i of [1,2,3]) console.log(i) </pre>
Interpretador	Define reglas gramaticales para interpretar un lenguaje de forma ejecutable.	Comportamiento	<pre> graph TD AbstractExpression --> TerminalExpression AbstractExpression --> NonTerminalExp </pre>	<pre> class Expr{ interp(){} } </pre>
Memento	Permite guardar y restaurar estados anteriores de un objeto sin violar su encapsulamiento.	Comportamiento	<pre> graph TD Caretaker --> Originator Originator --> Memento Memento --> Caretaker </pre>	<pre> class Origin{ save(){ return {...this} } } </pre>