Patrones de Diseño Arquitectónico

Aprendiz:

Kevin David Lopez Delgado

Instructor:

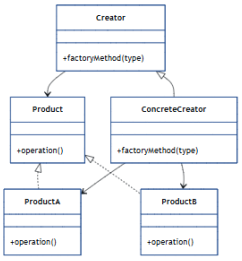
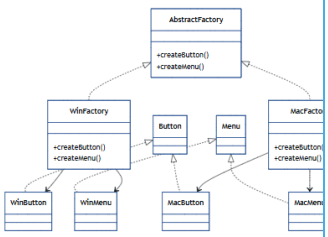
Ing. Néstor Montaño

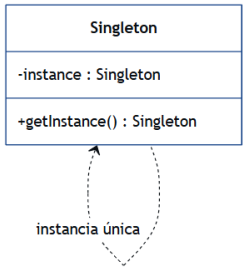
Tecnólogo Análisis y Desarrollo de Software

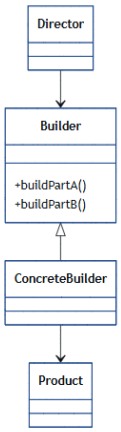
Ficha: 3064241

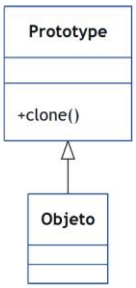
SENA Centro de Diseño y Metrología

Patrones Creacionales

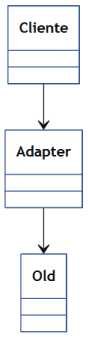
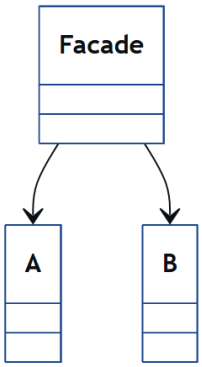
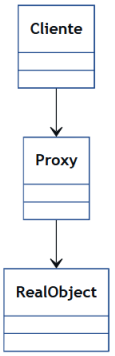
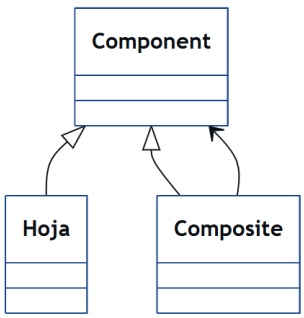
Nombre del Patrón	Definición	Clasificación	Ejemplo UML	Implementación en JavaScript
Factory Method	Define un método encargado de crear objetos, permitiendo que las subclases decidan qué tipo concreto instanciar, evitando acoplar el código cliente a clases específicas.	**Creacional**		class Creator { factoryMethod(t) { return t === 'A' ? new A() : new B(); } }
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		class UIFactory { createButton() {} createMenu() {} }

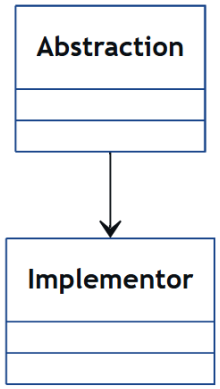
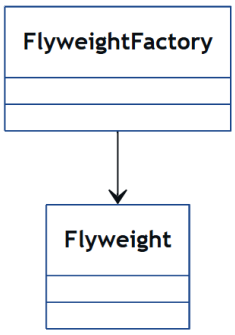
Singleton	Restringe la creación de una clase a una única instancia y ofrece un acceso global a ella desde cualquier parte del programa.	Creacional		class Singleton {
				constructor() {
				if (Singleton.instance)
				return Singleton.instance;
				Singleton.instance =
				this;
				}
				}

Builder	Permite construir objetos complejos paso a paso sin depender directamente de su construcción interna.	Creacional		class Builder {
				setA(a){ this.a = a;
				return this; }
				setB(b){ this.b = b;
				return this; }
				build(){ return { a: this.a,
				b: this.b }; }
				}
				
				const producto = new
				Builder().setA(1).setB(2).
				build();

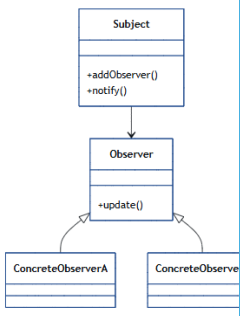
Prototype	Permite crear nuevos objetos clonando uno existente sin depender de su clase.	Creacional		const prototipo = { nombre: "Objeto" }; const copia = Object.create(prototipo);

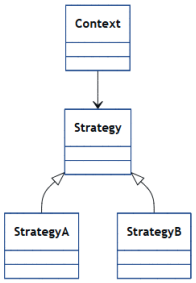
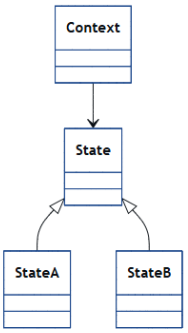
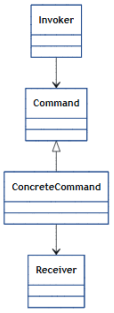
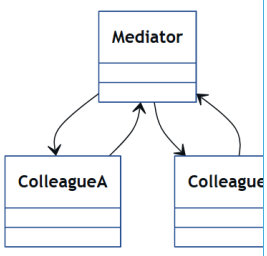
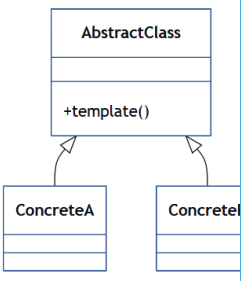
Patrones Estructurales

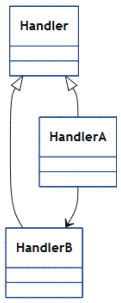
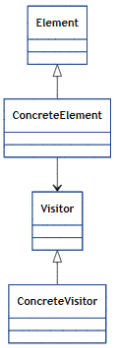
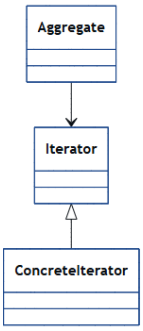
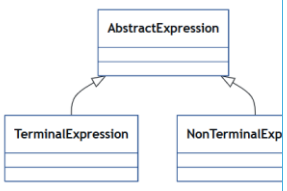
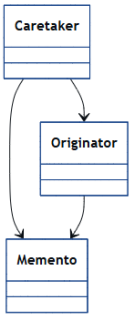
Nombre del Patrón	Definición	Clasificación	Ejemplo UML	Implementación en JavaScript
Adapter	Permite que interfaces incompatibles trabajen juntas convirtiendo la interfaz de una clase en otra que el cliente espera.	Estructural		**class Adapter{ request(){ return new Old().specific() } }**
Facade	Proporciona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

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

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		**class Sub{ obs=[] notify(d){ this.obs.forEach(o=>o.up(d)) } }**

Strategy	Permite cambiar algoritmos dinámicamente según la estrategia seleccionada.	Comportamiento		**class Ctx{ set(s){ this.s=s } run(){ return this.s.exec() } }**
State	Permite cambiar el comportamiento de un objeto dependiendo de su estado interno.	Comportamiento		**class StateA{ handle(){ return "A" } }**
command	Convierte una solicitud en un objeto independiente que puede ejecutarse, deshacerse o almacenarse.	Comportamiento		**class Cmd{ exec(){} }**
Mediator	Permite que los objetos se comuniquen entre sí mediante un mediador, reduciendo dependencias directas.	Comportamiento		**class Mediator{ send(msg,to){ to.receive(msg) } }**
Template Method	Define la estructura de un algoritmo y deja que las subclases implementen pasos específicos.	Comportamiento		**class T{ run(){ this.a(); this.b() } }**

Chain of responsibility	Envía solicitudes a través de una cadena de objetos hasta que uno de ellos la procese.	Comportamiento		**class H{ setNext(n){ this.n=n } handle(r){ if(this.n) this.n.handle(r) } }**
Vistor	Permite agregar nuevas operaciones a objetos sin modificar sus clases.	Comportamiento		**class Visitor{ visit(obj){ obj.accept(this) } }**
Iterator	Proporciona una forma estándar de recorrer una colección secuencialmente.	Comportamiento		**for(const i of [1,2,3]) console.log(i)**
Interpreter	Define reglas gramaticales para interpretar un lenguaje de forma ejecutable.	Comportamiento		**class Expr{ interp(){} }**
Memento	Permite guardar y restaurar estados anteriores de un objeto sin violar su encapsulamiento.	Comportamiento		**class Origin{ save(){ return {...this} } }**