Patrones Arquitectónicos

Aprendices del SENA

Julian David Vergara Ramirez

Instructor

Ing Nestor Montalo

Tecnólogo de Análisis y Desarrollo de Software

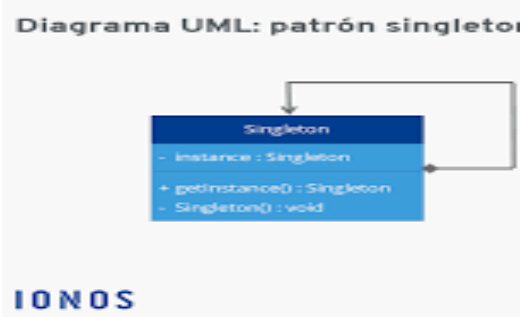
Centro de Diseño y Metrología - SENA

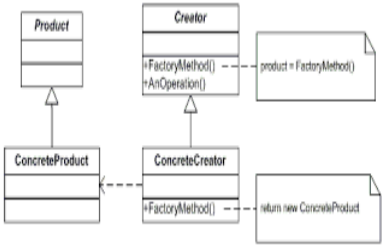
Ficha: 3064241

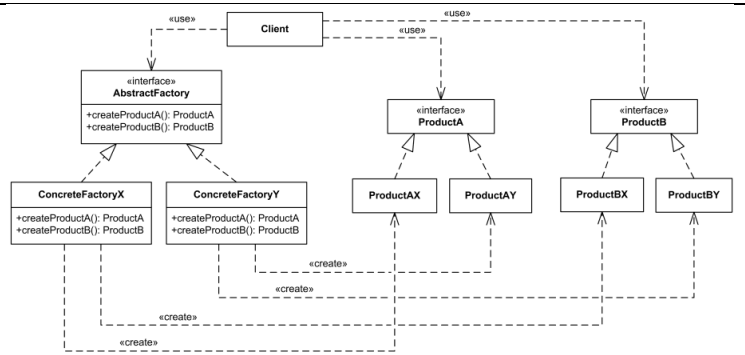
Bogotá D.C

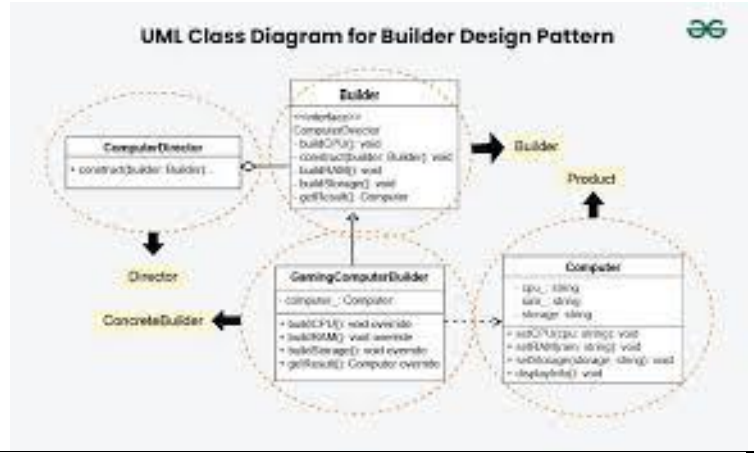
Patrones creacionales

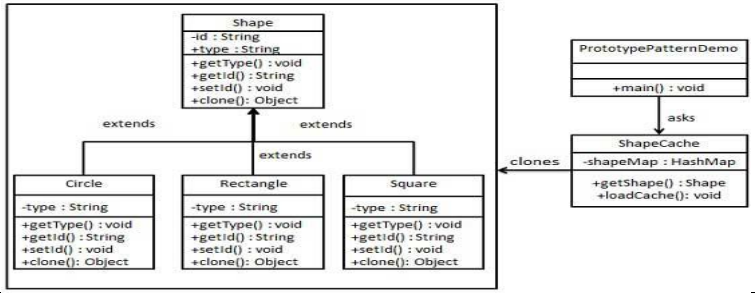
Nombre del Patrón	Definición	Clasificación	Ejemplo en UML	Implementación en JavaScript

singleton	Garantiza que una clase tenga **una sola instancia** y proporciona un acceso global a ella.	Patrón Creacional		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

Factory Method	Define una interfaz para crear objetos, dejando que las subclases decidan qué objeto crear.	Patrón Creacional		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();

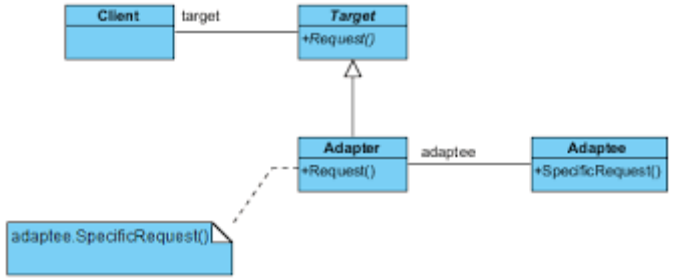
Abstract Factory	Crea familias de objetos relacionados sin especificar sus clases concretas.	Creacional		class AbstractFactory {
				crearBoton() {}
				}
				
				class FactoryWindows extends AbstractFactory {
				crearBoton() { return new BotonWindows(); }
				}
				
				class BotonWindows {
				render() { console.log("Botón Windows"); }
				}

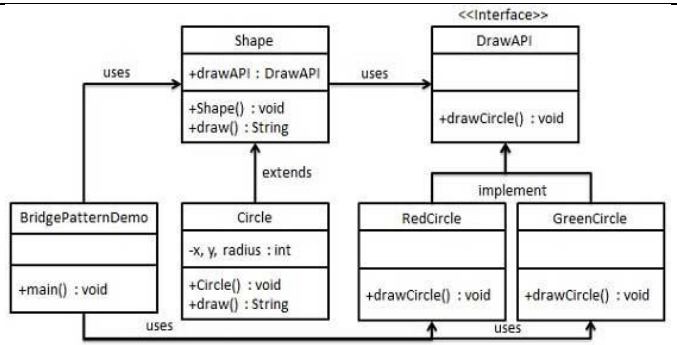
Builder	Permite construir objetos complejos paso a paso.	Creacional		class BuilderPizza {
				reset() { this.pizza = {}; }
				setMasa(m) { this.pizza.masa = m; }
				setQueso(q) { this.pizza.queso = q; }
				build() { return this.pizza; }
				}
				
				const b = new BuilderPizza();
				b.reset();
				b.setMasa("gruesa");
				b.setQueso("mozzarella");
				console.log(b.build());

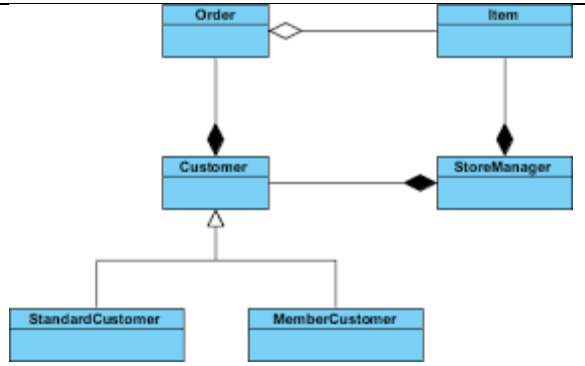
Prototype	Crea nuevos objetos copiando un objeto existente (prototipo).	Creacional		const personaje = {
				tipo: "Guerrero",
				clonar() {
				return { ...this };
				}
				};
				
				const copia = personaje.clonar();

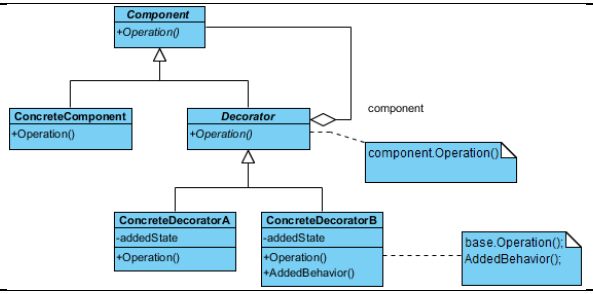
Patrones de Estructurales

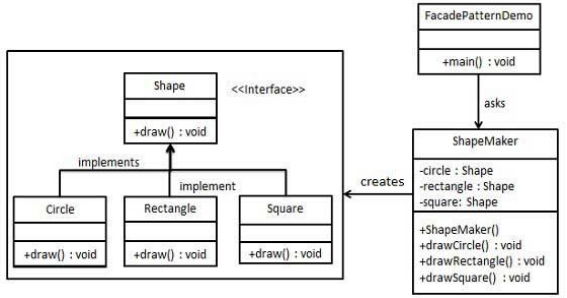
Nombre del Patrón	Definición	Clasificación	Ejemplo en UML	Implementación en JavaScript

Adapter	Convierte la interfaz de una clase en otra que el cliente espera.	Estructural		class Adaptee {
				requestOld(){ return "Antiguo"; }
				}
				
				class Adapter {
				constructor(){ this.adaptee = new Adaptee(); }
				request(){ return this.adaptee.requestOld(); }
				}

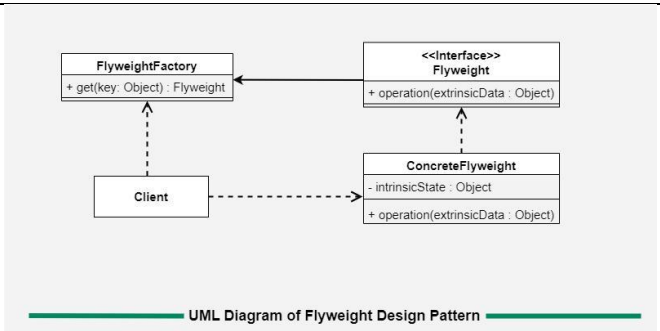
Bridge	Separa una abstracción de su implementación para permitir variarlas independientemente.	Estructural		class Dispositivo {
				
				constructor(implementacion){
				this.imp = implementacion; }
				encender(){
				this.imp.encender(); }
				}
				
				class TV {
				encender(){
				console.log("TV encendida");
				}
				}
				
				const tv = new Dispositivo(new TV());
				tv.encender();

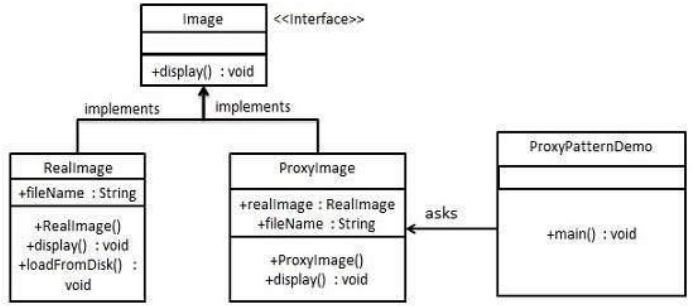
Composite	Permite tratar objetos individuales y compuestos de forma uniforme.	Estructural		class Nodo {
				mostrar() {}
				}
				
				class Hoja extends Nodo {
				mostrar(){ console.log("Hoja"); }
				}
				
				class Compuesto extends Nodo {
				constructor(){ super();
				this.hijos=[]; }
				agregar(n){ this.hijos.push(n); }
				mostrar(){ this.hijos.forEach(h =>
				h.mostrar()); }
				}

Decorator	Añade responsabilidades adicionales dinámicamente a un objeto.	Estructural		class Cafe {
				costo(){ return 5; }
				}
				
				class ConLeche {
				constructor(cafe){ this.cafe=cafe;
				}
				costo(){ return this.cafe.costo() +
				2; }
				}

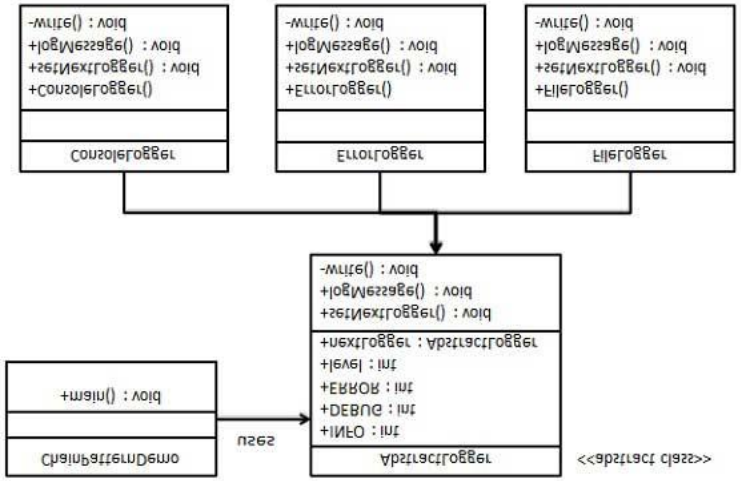
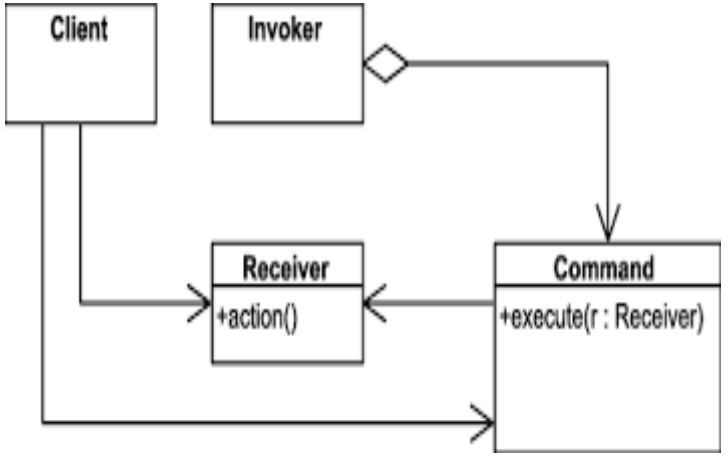
Facade	Proporciona una interfaz simplificada a un sistema complejo.	Estructural		class Subsistema {
				operacion(){ return
				"Operación interna"; }
				}
				
				class Facade {
				constructor(){ this.s = new
				Subsistema(); }
				operar(){ return
				this.s.operacion(); }
				}

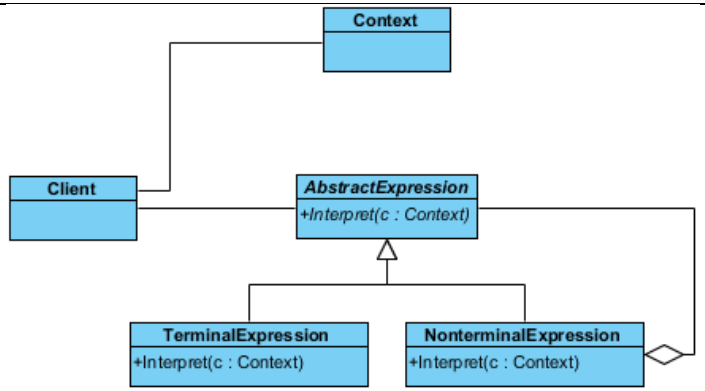
Nombre del Patrón	Definición	Clasificación	Ejemplo en UML	Implementación en JavaScript

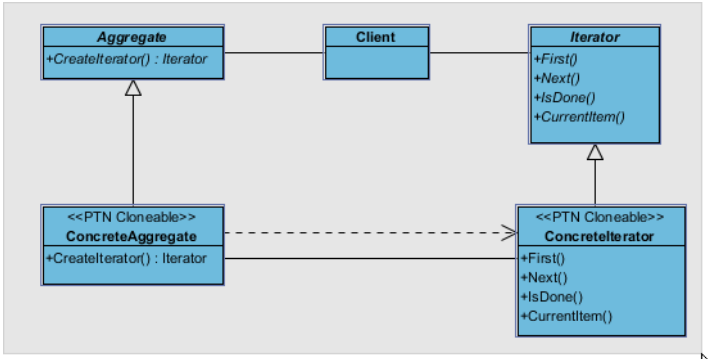
Flyweight	Minimiza el uso de memoria compartiendo objetos de bajo peso.	Estructural		class Flyweight {
				constructor(color){
				this.color=color; }
				}
				
				class Factory {
				constructor(){ this.pool={}; }
				get(color){
				if(!this.pool[color])
				this.pool[color] = new
				Flyweight(color);
				return this.pool[color];
				}
				}

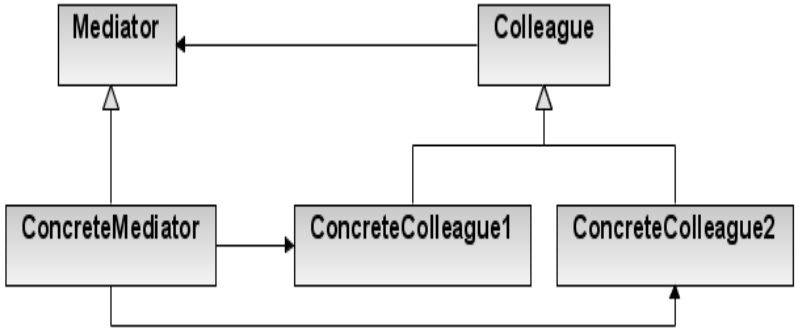
Proxy	Proporciona un sustituto que controla el acceso a otro objeto.	Estructural		class ServicioReal {
				request(){ return
				"Datos reales"; }
				}
				
				class ProxyServicio {
				constructor(){ this.real
				= new ServicioReal(); }
				request(){ return
				this.real.request(); }
				}

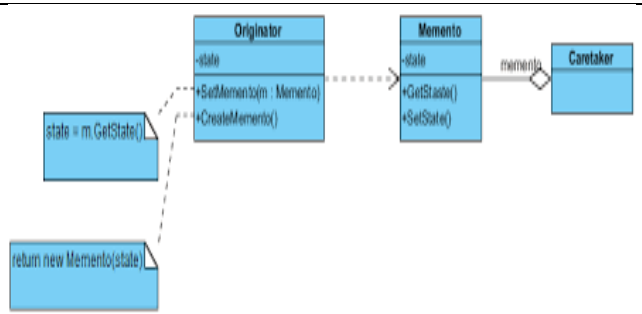
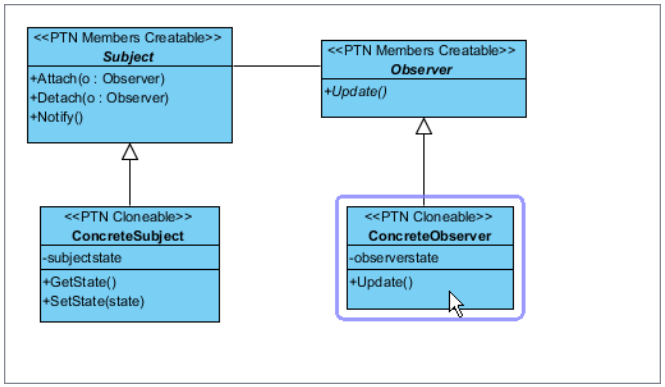
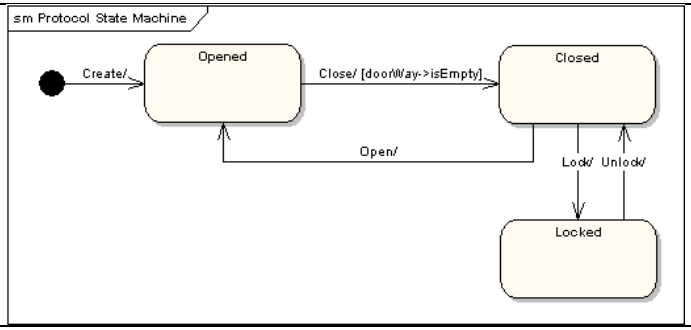
Patrones de comportamiento

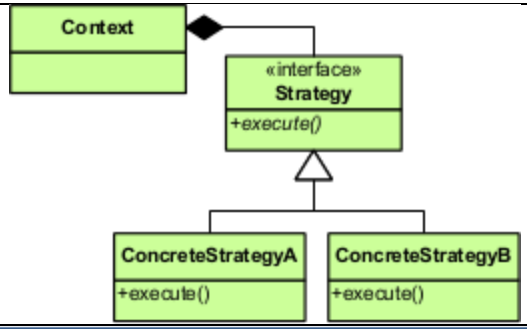
Nombre del Patrón	Definición	Clasificación	Ejemplo en UML	Implementación en JavaScript
Chain of Responsibility	Permite pasar una petición por una cadena de receptores hasta que uno la procese.	Comportamiento		class Handler { setNext(h){ this.next=h; } handle(req){ if(this.next) this.next.handle(req); } }
Command	Encapsula una petición como un objeto.	Comportamiento		class EncenderLuz { ejecutar(){ console.log("Luz encendida"); } }

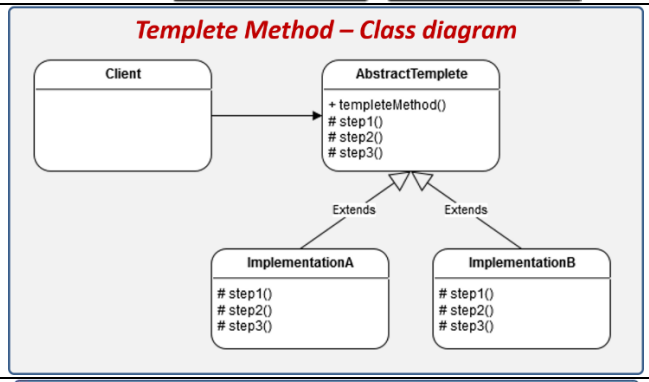
Interpreter	Define una gramática y un intérprete para evaluar expresiones.	Comportamiento		class Numero {
				constructor(v){
				this.v=v; }
				interpretar(){ return
				this.v; }
				}

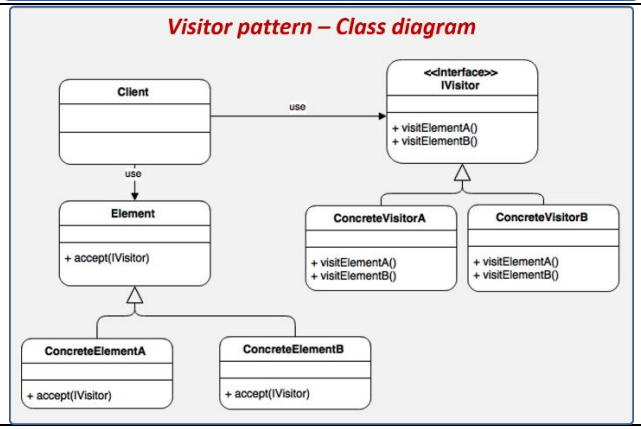
Iterator	Permite recorrer elementos de una colección sin exponer su estructura interna.	Comportamiento		const lista = [1,2,3];
				for (let n of lista)
				console.log(n);

Mediator	Centraliza la comunicación entre objetos para reducir dependencias.	Comportamiento		class Chat { enviar(msg, usuario){ console.log(`${usuario}: ${msg}`); } }

Nombre del Patrón	Definición	Clasificación	Ejemplo en UML	Implementación en JavaScript
Memento	Guarda y restaura el estado interno de un objeto.	Comportamiento		class Memento { constructor(estado){ this.estado=estado; } }
Observer	Actualiza automáticamente a los observadores cuando el sujeto cambia.	Comportamiento		class Subject { constructor(){ this.obs=[]; } attach(o){ this.obs.push(o); } notify(v){ this.obs.forEach(o=>o.update(v)); } }
State	Permite cambiar el comportamiento de un objeto según su estado interno.	Comportamiento		class EstadoA { manejar(){ console.log("Estado A"); } }

Strategy	Define algoritmos intercambiables.	Comportamiento		class A {
				ejecutar(){
				console.log("A");
				} }

Template Method	Define el esqueleto de un algoritmo, dejando pasos a subclases.	Comportamiento		class Receta {
				cocinar(){
				this.preparar();
				this.servir();
				}
				}

Visitor	Permite agregar operaciones a objetos sin modificar sus clases.	Comportamiento		class Visitor {
				visitar(e){
				e.aceptar(this); }
				}