

Patrones Arquitectónicos

Aprendices del SENA

Julian David Vergara Ramirez

Instructor

Ing Nestor Montalo

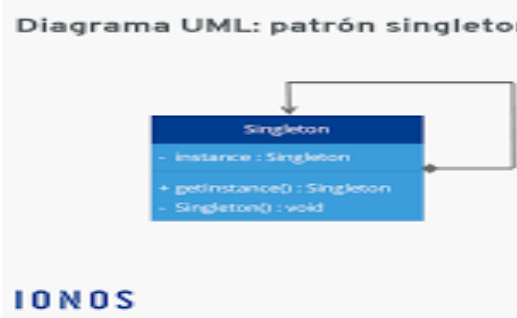
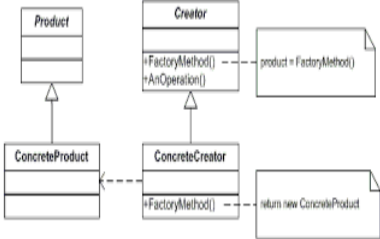
Tecnólogo de Análisis y Desarrollo de Software

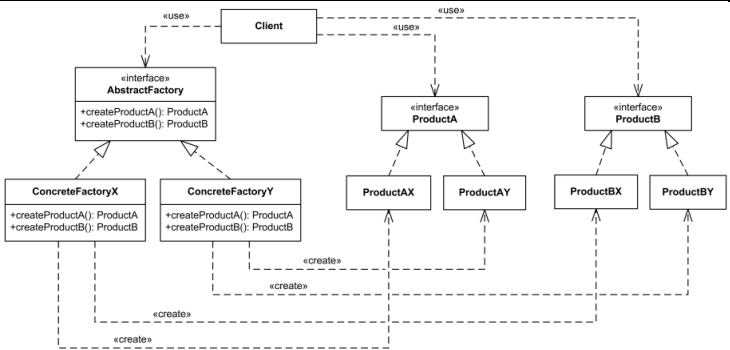
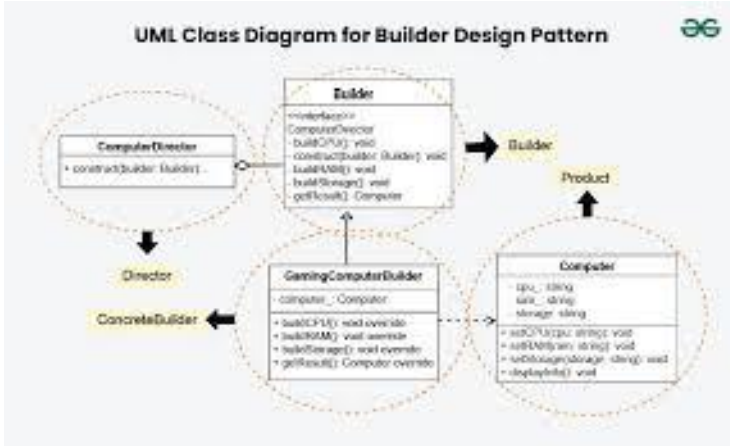
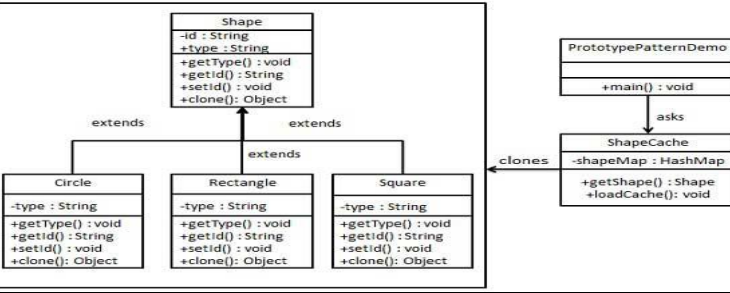
Centro de Diseño y Metrología - SENA

Ficha: 3064241

Bogotá D.C

Patrones creacionales

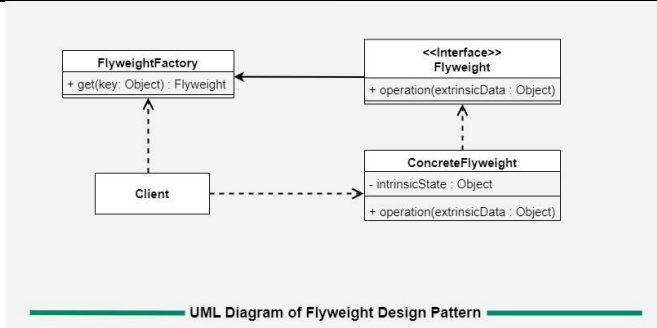
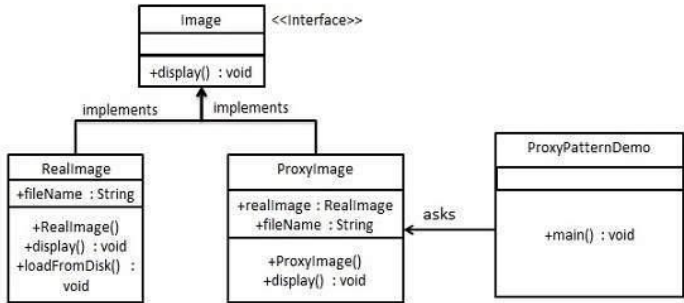
Nombre del Patrón	Definición	Clasificación	Ejemplo en UML	Implementación en JavaScript
<i>singleton</i>	Garantiza que una clase tenga una sola instancia y proporciona un acceso global a ella.	Patrón Creacional		<pre> class Singleton { constructor() { if (Singleton.instancia) { return Singleton.instancia; } 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> 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	 <p>The diagram shows an AbstractFactory interface with methods <code>+createProductA(): ProductA</code> and <code>+createProductB(): ProductB</code>. It has two concrete implementations: ConcreteFactoryX and ConcreteFactoryY. There are two product interfaces: ProductA and ProductB, each with two concrete implementations: ProductAX, ProductAY, ProductBX, and ProductBY. A Client depends on the AbstractFactory and the two product interfaces. Concrete factories implement the AbstractFactory interface and are associated with specific product implementations (e.g., ConcreteFactoryX with ProductAX and ProductBX).</p>	<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>The diagram is titled "UML Class Diagram for Builder Design Pattern". It shows a Builder interface with methods <code>+buildCPU(): void</code>, <code>+buildRAM(): void</code>, <code>+buildStorage(): void</code>, and <code>+getResult(): Computer</code>. There are two concrete builders: ComputerDirector and GamingComputerBuilder. ComputerDirector implements the Builder interface. GamingComputerBuilder implements the Builder interface and has a <code>-computer: Computer</code> attribute. A Computer class has attributes <code>-cpu: string</code>, <code>-ram: string</code>, <code>-storage: string</code>, and methods <code>+addCPU(cpu: string): void</code>, <code>+addRAM(ram: string): void</code>, <code>+addStorage(storage: string): void</code>, and <code>+displayInfo(): void</code>. A Product class is also shown. The diagram illustrates how the Builder interface is used to construct a Computer object.</p>	<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>The diagram shows a Shape interface with attributes <code>-id: String</code> and <code>-type: String</code>, and methods <code>+getType(): void</code>, <code>+getId(): String</code>, <code>+setId(): void</code>, and <code>+clone(): Object</code>. It has three concrete implementations: Circle, Rectangle, and Square. A PrototypePatternDemo class has a <code>+main(): void</code> method. A ShapeCache class has a <code>-shapeMap: HashMap</code> attribute and methods <code>+getShape(): Shape</code> and <code>+loadCache(): void</code>. The diagram illustrates how the Shape interface is used to create and clone objects, and how the ShapeCache class is used to store and retrieve objects.</p>	<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 class Target { +Request() } class Adapter { +Request() } class Adaptee { +SpecificRequest() } Client --> Target : target Adapter -- > Target Adapter --> Adaptee : adaptee note for Adapter "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 Shape { +drawAPI : DrawAPI +Shape() : void +draw() : String } class DrawAPI { <<interface>> +drawCircle() : void } class Circle { -x, y, radius : int +Circle() : void +draw() : String } class RedCircle { +drawCircle() : void } class GreenCircle { +drawCircle() : void } class BridgePatternDemo { +main() : void } Shape -- > DrawAPI : extends Circle -- > Shape : extends RedCircle -- > DrawAPI : implement GreenCircle -- > DrawAPI : implement BridgePatternDemo --> Shape : uses BridgePatternDemo --> Circle : uses BridgePatternDemo --> RedCircle : uses BridgePatternDemo --> 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 StoreManager class StandardCustomer class MemberCustomer Order "1" *-- "*" Item 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 *-- Component : component Decorator *-- ConcreteDecoratorA Decorator *-- ConcreteDecoratorB ConcreteDecoratorA *-- Decorator ConcreteDecoratorB *-- Decorator ConcreteDecoratorB ..> Component : base.Operation(), 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 { <<interface>> +draw() : void } class Circle { +draw() : void } class Rectangle { +draw() : void } class Square { +draw() : void } class ShapeMaker { -circle : Shape -rectangle : Shape -square : Shape +ShapeMaker() +drawCircle() : void +drawRectangle() : void +drawSquare() : void } class FacadePatternDemo { +main() : void } Shape < -- Circle Shape < -- Rectangle Shape < -- Square ShapeMaker --> Circle : creates ShapeMaker --> Rectangle : creates ShapeMaker --> Square : creates FacadePatternDemo --> ShapeMaker : asks </pre>	<pre> class Subsistema { operacion(){ return "Operación interna"; } } class Facade { constructor(){ this.s = new Subsistema(); } operar(){ return this.s.operacion(); } } </pre>

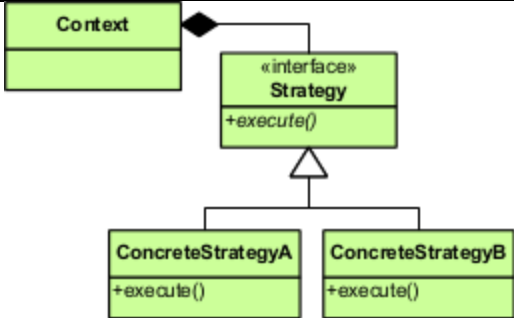
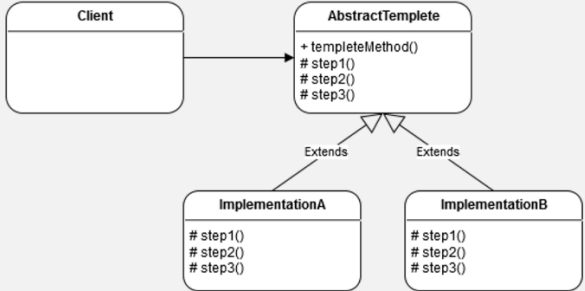
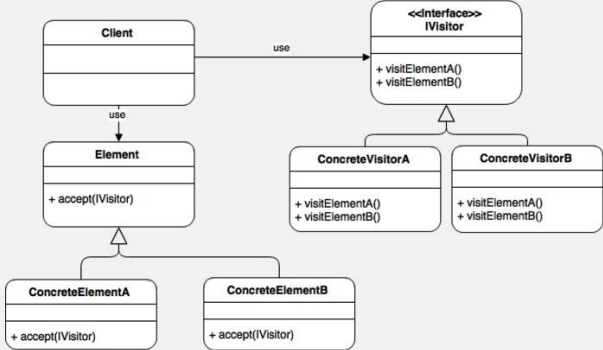
Nombre del Patrón	Definición	Clasificación	Ejemplo en UML	Implementación en JavaScript
Flyweight	Minimiza el uso de memoria compartiendo objetos de bajo peso.	Estructural	 <pre> classDiagram class FlyweightFactory { +get(key: Object) Flyweight } class Client class Flyweight { <<interface>> +operation(extrinsicData: Object) } class ConcreteFlyweight { -intrinsicState: Object +operation(extrinsicData: Object) } FlyweightFactory ..> Flyweight Client ..> FlyweightFactory Client ..> ConcreteFlyweight Flyweight < .. ConcreteFlyweight </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 class Image { <<interface>> +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 < .. RealImage Image < .. ProxyImage ProxyImage --> RealImage : implements ProxyPatternDemo --> ProxyImage : asks </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> classDiagram class AbstractLogger { <<abstract class>> -write() void +logMessage() void +getMessage() void +execute() void +level int +ERROR int +DEBUG int +INFO int } class ConsoleLogger { -write() void +logMessage() void +getMessage() void +consoleLog() void } class ErrorLogger { -write() void +logMessage() void +getMessage() void +consoleLog() void } class FileLogger { -write() void +logMessage() void +getMessage() void +consoleLog() void } class ChainHandlerDemo { +main() void } AbstractLogger < -- ConsoleLogger AbstractLogger < -- ErrorLogger AbstractLogger < -- FileLogger ChainHandlerDemo --> AbstractLogger : use </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 { +action() } class Command { +execute(r: Receiver) } Invoker *-- Command Client --> Receiver Client --> Command </pre>	<pre> class EncenderLuz { ejecutar(){ console.log("Luz encendida"); } } </pre>

Interpreter	Define una gramática y un intérprete para evaluar expresiones.	Comportamiento	<pre> classDiagram class Context class Client class AbstractExpression { +interpret(c : Context) } class TerminalExpression { +interpret(c : Context) } class NonterminalExpression { +interpret(c : Context) } Client --> Context Client --> AbstractExpression AbstractExpression < -- TerminalExpression AbstractExpression < -- NonterminalExpression NonterminalExpression o-- Context </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 Aggregate { +createIterator() : Iterator } class Client class Iterator { +first() +next() +isDone() +currentItem() } class ConcreteAggregate { <<PTN Cloneable>> +createIterator() : Iterator } class ConcreteIterator { <<PTN Cloneable>> +first() +next() +isDone() +currentItem() } Client --> Aggregate Client --> Iterator Aggregate < -- ConcreteAggregate Iterator < -- ConcreteIterator ConcreteAggregate ..> ConcreteIterator ConcreteAggregate <.. 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 Colleague class ConcreteMediator class ConcreteColleague1 class ConcreteColleague2 Mediator < -- ConcreteMediator Colleague < -- ConcreteColleague1 Colleague < -- ConcreteColleague2 ConcreteMediator --> ConcreteColleague1 ConcreteMediator --> ConcreteColleague2 ConcreteColleague1 --> Mediator ConcreteColleague2 --> Mediator </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 ..> Memento : memento Caretaker o-- Memento Note for Originator: state = m.GetState() Note for 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 Subject { <<PTN Members Creatable>> +Attach(o: Observer) +Detach(o: Observer) +Notify() } class ConcreteSubject { <<PTN Cloneable>> -subjectstate +GetState() +SetState(state) } class Observer { <<PTN Members Creatable>> +Update() } class ConcreteObserver { <<PTN Cloneable>> -observerstate +Update() } Subject < -- ConcreteSubject Observer < -- ConcreteObserver Subject --> Observer </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] Closed --> Opened : Open/ Closed --> Locked : Lock/ Locked --> Closed : Unlock/ Locked --> Locked : </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 o--> 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	<p>Template Method – Class diagram</p>  <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>	<pre> class Receta { cocinar(){ this.preparar(); this.servir(); } } </pre>
Visitor	Permite agregar operaciones a objetos sin modificar sus clases.	Comportamiento	<p>Visitor pattern – Class diagram</p>  <pre> classDiagram class Client class Element { +accept(Visitor) } class ConcreteElementA { +accept(IVisitor) } class ConcreteElementB { +accept(IVisitor) } class IVisitor { <<interface>> +visitElementA() +visitElementB() } class ConcreteVisitorA { +visitElementA() +visitElementB() } class ConcreteVisitorB { +visitElementA() +visitElementB() } Client --> Element : use Client --> IVisitor : use Element < -- ConcreteElementA Element < -- ConcreteElementB IVisitor < -- ConcreteVisitorA IVisitor < -- ConcreteVisitorB </pre>	<pre> class Visitor { visitar(e){ e.aceptar(this); } } </pre>