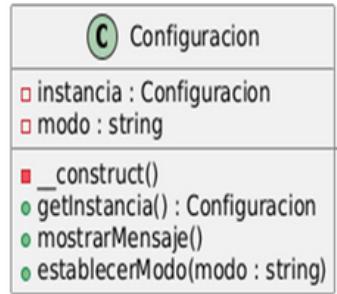


Patrones Creacionales

Patrón	Descripción	Cuando Usarlo	Diagrama UML
Factory Method	Crea objetos sin especificar la clase exacta a usar	<ul style="list-style-type: none"> • Cuando no sabes qué tipo de objeto necesitas hasta el momento de ejecución • Para evitar dependencias directas con clases concretas • Ejemplo: crear diferentes tipos de notificaciones (email, SMS, push) 	<pre> classDiagram class CreadorAnimal { <<A>> crear() : Animal } class Animal { <<I>> } class CreadorPerro { <<C>> crear() : Animal } class Perro { <<C>> } CreadorAnimal < -- CreadorPerro CreadorPerro --> Animal CreadorPerro --> Perro </pre> <p>Este diagrama UML ilustra el patrón Factory Method. Se muestra una clase abstracta CreadorAnimal que define un método <code>crear()</code> que retorna un objeto Animal. Una clase concreta CreadorPerro hereda de CreadorAnimal y implementa su propio método <code>crear()</code> que retorna un objeto Perro. Existe una dependencia entre CreadorPerro y Animal, y otra entre CreadorPerro y Perro.</p>
Builder	Construye objetos paso a paso, evita constructores con muchos parámetros	<ul style="list-style-type: none"> • Cuando un objeto tiene muchos atributos opcionales • Para evitar constructores con 5+ parámetros • Ejemplo: construir una hamburguesa personalizada, configurar un PC 	<pre> classDiagram class Ingeniero { <<C>> construir(builder : ConstructorCasa) } class ConstructorCasa { <<I>> construirParedes() construirPuertas() construirVentanas() obtenerCasa() : Casa } class ConstructorCasaSimple { <<C>> construirParedes() construirPuertas() construirVentanas() obtenerCasa() : Casa } class Casa { <<C>> paredes puertas ventanas } Ingeniero --> ConstructorCasa ConstructorCasa --> ConstructorCasaSimple ConstructorCasaSimple --> Casa </pre> <p>Este diagrama UML ilustra el patrón Builder. Se muestra una clase Ingeniero que llama al método <code>construir()</code> de una clase ConstructorCasa. La clase ConstructorCasa es una interfaz (I) que define métodos para construir paredes, puertas y ventanas, y para obtener la casa final. Una clase concreta ConstructorCasaSimple implementa esta interfaz y llama al método <code>construir()</code> de la clase Casa, que contiene los atributos <code>paredes</code>, <code>puertas</code> y <code>ventanas</code>.</p>

Patrones Creacionales

Patrón	Descripción	Cuando Usarlo	Diagrama UML
Singleton	Solo una instancia de la clase en toda la app	<ul style="list-style-type: none">• Conexión a base de datos• Configuración global de la app• Logger o sistema de logs• Cache compartido	 <p>The diagram shows a class named 'Configuracion' with the following details:<ul style="list-style-type: none">Attributes:<ul style="list-style-type: none">instancia : Configuracionmodo : stringOperations:<ul style="list-style-type: none">_construct()getInstancia() : ConfiguracionmostrarMensaje()establecerModo(modo : string)</p> <p>A callout box points to the operations and contains the following notes:<ul style="list-style-type: none">• Solo existe una instancia.• El constructor es privado.• getInstancia() controla el acceso.• Se puede leer/cambiar el "modo".</p>

Patrones Estructurales

Patrón	Descripción	Cuando Usarlo	Diagrama UML
Adapter	Hace que dos interfaces incompatibles trabajen juntas	<ul style="list-style-type: none"> • Integrar código viejo con código nuevo • Usar librerías externas con interfaces diferentes <ul style="list-style-type: none"> • Ejemplo: adaptar un celular viejo a una interfaz moderna 	<pre> classDiagram class Reproductor { <<I>> <<reproducirArchivo(nombre : string)>> } class AdaptadorAAC { <<reproductorAAC : ReproductorAACAntiguo>> <<__construct(reproductorAAC : ReproductorAACAntiguo)>> <<reproducirArchivo(nombre : string)>> } class ReproductorAACAntiguo { <<reproducirAAC(archivo : string)>> } class ReproductorMP3 { <<reproducirArchivo(nombre : string)>> } Reproductor "1" --> "1" AdaptadorAAC : reproducirArchivo(nombre : string) AdaptadorAAC "1" --> "1" ReproductorAACAntiguo : usa AdaptadorAAC "1" --> "1" ReproductorMP3 : reproducirArchivo(nombre : string) note over AdaptadorAAC: Convierte la llamada: reproducirArchivo() en: reproducirAAC() </pre>
Decorator	Agrega funcionalidad a objetos envolviéndolos en capas	<ul style="list-style-type: none"> • Agregar funcionalidades sin modificar la clase original • Combinaciones dinámicas de características • Ejemplo: agregar seguro, notificaciones y límites a una cuenta bancaria 	<pre> classDiagram class Bebida { <<obtenerDescripcion() : string>> <<obtenerCosto() : float>> } class BebidaDecorador { <<bebida : Bebida>> <<__construct(bebida : Bebida)>> <<obtenerDescripcion() : string>> <<obtenerCosto() : float>> } class DecoradorLeche { <<obtenerDescripcion() : string>> <<obtenerCosto() : float>> } class DecoradorChocolate { <<obtenerDescripcion() : string>> <<obtenerCosto() : float>> } class CafeSimple { <<obtenerDescripcion() : string>> <<obtenerCosto() : float>> } Bebida --> "1" BebidaDecorador : envuelve BebidaDecorador --> "1" DecoradorLeche BebidaDecorador --> "1" DecoradorChocolate note over BebidaDecorador: El decorador agrega funcionalidad sin modificar la clase original. </pre>

Patrones Estructurales



Patrón	Descripción	Cuando Usarlo	Diagrama UML
Facade	Interfaz simple para un sistema complejo	<ul style="list-style-type: none">• Simplificar sistemas complejos con muchas clases• Ocultar complejidad interna• Ejemplo: un control remoto que maneja motor, luces y aire del auto	<p>The diagram illustrates the Facade pattern. At the top is the FachadaCine class, which has three private attributes: sonido : SistemaSonido, pantalla : SistemaPantalla, and proyector : SistemaProyector. It contains three public methods: _construct(sonido, pantalla, proyector), iniciarPelicula(), and terminarPelicula(). Below FachadaCine are three subclasses: SistemaSonido, SistemaPantalla, and SistemaProyector. Arrows point from FachadaCine to each of these three classes. A callout box to the right of the diagram states: "La fachada simplifica el uso de varios subsistemas internos."</p>

Patrones de Comportamiento

Patrón	Descripción	Cuando Usarlo	Diagrama UML
Strategy	Cambia algoritmos fácilmente sin if/switch	<ul style="list-style-type: none"> Múltiples formas de hacer algo (pagar, ordenar, calcular) Evitar if/switch gigantes <ul style="list-style-type: none"> Ejemplo: diferentes métodos de pago (tarjeta, efectivo, Nequi) 	<pre> classDiagram class CarritoCompras { strategia : EstrategiaEnvio __construct(strategia : EstrategiaEnvio) cambiarEstrategia(strategia : EstrategiaEnvio) procesarEnvio(distancia : int) } interface EstrategiaEnvio { calcularCosto(distancia : int) : float } class EnvioNormal { calcularCosto(distancia : int) : float } class EnvioRapido { calcularCosto(distancia : int) : float } class EnvioEconomico { calcularCosto(distancia : int) : float } CarritoCompras --> EstrategiaEnvio EstrategiaEnvio < -- EnvioNormal EstrategiaEnvio < -- EnvioRapido EstrategiaEnvio < -- EnvioEconomico </pre> <p>CarritoCompras</p> <p>EstrategiaEnvio</p> <p>EnvioNormal</p> <p>EnvioRapido</p> <p>EnvioEconomico</p> <p>Usa la estrategia seleccionada para calcular el costo de envío.</p>
Observer	Un objeto notifica a varios cuando cambia	<ul style="list-style-type: none"> Sistema de suscripciones <ul style="list-style-type: none"> Notificaciones automáticas Ejemplo: canal de YouTube notifica a suscriptores cuando sube video 	<pre> classDiagram interface Sujeto { adjuntar(obs : Observador) desadjuntar(obs : Observador) notificar() } class CanalYouTube { suscriptores : List<Observador> ultimoVideo : string adjuntar(obs : Observador) desadjuntar(obs : Observador) subirVideo(titulo : string) notificar() } interface Observador { actualizar(mensaje : string) } class SuscriptorMovil { nombre : string actualizar(mensaje : string) } class SuscriptorPC { nombre : string actualizar(mensaje : string) } Sujeto < -- CanalYouTube Sujeto < -- Observador CanalYouTube --> Observador SuscriptorMovil SuscriptorPC </pre> <p>Sujeto</p> <p>CanalYouTube</p> <p>Observador</p> <p>SuscriptorMovil</p> <p>SuscriptorPC</p> <p>Guarda lista de suscriptores. Cuando sube video, llama a notificar().</p> <p>notifica a</p> <p>Cada observador implementa actualizar()</p>

Patrones de Comportamiento



Patrón	Descripción	Cuando Usarlo	Diagrama UML
Command	<p>Encapsula acciones como objetos</p>	<ul style="list-style-type: none"> Deshacer/rehacer acciones Encolar operaciones Sistema de macros Ejemplo: control remoto con botones programables 	<pre> classDiagram class ControlRemoto { boton : Comando setComando(c : Comando) presionarBoton() } class Comando { ejecutar() } class Luz { encender() apagar() } ControlRemoto --> Comando : ejecuta Comando --> Luz : usa </pre> <p>The diagram illustrates the Command pattern. It features four classes: ControlRemoto, Comando, Luz, and Controlador. ControlRemoto has an attribute boton of type Comando and methods setComando and presionarBoton. Comando has a method ejecutar. Luz has methods encender and apagar. A directed association labeled ejecuta connects ControlRemoto to Comando. Another directed association labeled usa connects Comando to Luz. Callouts provide context: 'Invocador: llama a ejecutar() sin conocer los detalles.' points to the ControlRemoto class; 'Define la acción ejecutar() que todos los comandos comparten.' points to the ejecutar() method in the Comando interface; and 'Receptor real que sabe encender y apagar.' points to the Luz class.</p>