

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 { +crear() Animal } class CreadorPerro { +crear() Animal } class Animal { } class Perro { } CreadorAnimal < -- CreadorPerro CreadorPerro --> Perro : crea Perro ..> Animal </pre>
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 { +construir(builder ConstructorCasa) } class ConstructorCasa { +construirParedes() +construirPuertas() +construirVentanas() +obtenerCasa() Casa } class ConstructorCasaSimple { +construirParedes() +construirPuertas() +construirVentanas() +obtenerCasa() Casa } class Casa { +paredes +puertas +ventanas } Ingeniero --> ConstructorCasa : dirige construcción ConstructorCasa < .. ConstructorCasaSimple ConstructorCasaSimple --> Casa : crea </pre>

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<ul style="list-style-type: none">• Cache compartido	<pre>classDiagram class Configuracion { +instancia : Configuracion +modo : string +getInstancia() : Configuracion +mostrarMensaje() +establecerModo(modo : string) -_construct() } note for Configuracion "• Solo existe una instancia.\n• El constructor es privado.\n• getInstancia() controla el acceso.\n• Se puede leer/cambiar el 'modo'."</pre>

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 • Ejemplo: adaptar un celular viejo a una interfaz moderna 	<pre> classDiagram class Reproductor { <<interface>> +reproducirArchivo(nombre: string) } class ReproductorAAC { +reproducirAAC : ReproductorAACAntiguo +_construct(reproductorAAC : ReproductorAACAntiguo) +reproducirArchivo(nombre: string) } class ReproductorMP3 { +reproducirArchivo(nombre: string) } class ReproductorAACAntiguo { +reproducirAAC(archivo: string) } class AdaptadorAAC { +reproductorAAC : ReproductorAACAntiguo +_construct(reproductorAAC : ReproductorAACAntiguo) +reproducirArchivo(nombre: string) } Reproductor < -- ReproductorAAC Reproductor < -- ReproductorMP3 AdaptadorAAC --> ReproductorAAC AdaptadorAAC --> ReproductorAACAntiguo : usa </pre> <p>Convierte la llamada: reproducirArchivo() en: reproducirAAC()</p>
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 { <<interface>> +obtenerDescripcion(): string +obtenerCosto(): float } class CafeSimple { +obtenerDescripcion(): string +obtenerCosto(): float } class DecoradorLeche { +obtenerDescripcion(): string +obtenerCosto(): float } class DecoradorChocolate { +obtenerDescripcion(): string +obtenerCosto(): float } class BebidaDecorator { +bebida : Bebida +_construct(bebida : Bebida) +obtenerDescripcion(): string +obtenerCosto(): float } Bebida < -- CafeSimple Bebida < -- DecoradorLeche Bebida < -- DecoradorChocolate Bebida < -- BebidaDecorator DecoradorLeche --> Bebida : envuelve DecoradorChocolate --> Bebida : envuelve BebidaDecorator --> Bebida : envuelve </pre> <p>El decorador agrega funcionalidad sin modificar la clase original.</p>

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	<pre>classDiagram class FacadaCine { sonido : SistemaSonido pantalla : SistemaPantalla proyector : SistemaProyector __construct(sonido, pantalla, proyector) iniciarPelicula() terminarPelicula() } class SistemaSonido { encender() subirVolumen() apagar() } class SistemaPantalla { bajarPantalla() subirPantalla() } class SistemaProyector { encender() apagar() } FacadaCine --> SistemaSonido FacadaCine --> SistemaPantalla FacadaCine --> SistemaProyector</pre>

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 { +EstrategiaEnvio estrategia +_construct(estrategia : EstrategiaEnvio) +CambiarEstrategia(estrategia : EstrategiaEnvio) +procesarEnvio(distancia : int) } class EstrategiaEnvio { <<interface>> +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>UML Diagram for Strategy Pattern:</p> <ul style="list-style-type: none"> CarritoCompras (Concrete Class): <ul style="list-style-type: none"> Attributes: <code>estrategia : EstrategiaEnvio</code> Operations: <code>_construct(estrategia : EstrategiaEnvio)</code>, <code>CambiarEstrategia(estrategia : EstrategiaEnvio)</code>, <code>procesarEnvio(distancia : int)</code> EstrategiaEnvio (Interface): <ul style="list-style-type: none"> Operation: <code>calcularCosto(distancia : int) : float</code> EnvioNormal, EnvioRapido, EnvioEconomico (Concrete Classes): <ul style="list-style-type: none"> Operation: <code>calcularCosto(distancia : int) : float</code> <p>Relationships: CarritoCompras uses EstrategiaEnvio. EstrategiaEnvio is implemented by EnvioNormal, EnvioRapido, and EnvioEconomico.</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 class Sujeto { <<interface>> +adjuntar(obs : Observador) +desadjuntar(obs : Observador) +notificar() } class CanalYouTube { +suscriptores : List<Observador> +ultimoVideo : string +adjuntar(obs : Observador) +desadjuntar(obs : Observador) +subirVideo(titulo : string) +notificar() } class Observador { <<interface>> +actualizar(mensaje : string) } class SuscriptorMovil { +nombre : string +actualizar(mensaje : string) } class SuscriptorPC { +nombre : string +actualizar(mensaje : string) } Sujeto < .. CanalYouTube Observador < .. SuscriptorMovil Observador < .. SuscriptorPC CanalYouTube --> Sujeto CanalYouTube --> Observador : notifica a </pre> <p>UML Diagram for Observer Pattern:</p> <ul style="list-style-type: none"> Sujeto (Interface): <ul style="list-style-type: none"> Operations: <code>adjuntar(obs : Observador)</code>, <code>desadjuntar(obs : Observador)</code>, <code>notificar()</code> CanalYouTube (Concrete Class): <ul style="list-style-type: none"> Attributes: <code>suscriptores : List<Observador></code>, <code>ultimoVideo : string</code> Operations: <code>adjuntar(obs : Observador)</code>, <code>desadjuntar(obs : Observador)</code>, <code>subirVideo(titulo : string)</code>, <code>notificar()</code> Observador (Interface): <ul style="list-style-type: none"> Operation: <code>actualizar(mensaje : string)</code> SuscriptorMovil, SuscriptorPC (Concrete Classes): <ul style="list-style-type: none"> Attributes: <code>nombre : string</code> Operation: <code>actualizar(mensaje : string)</code> <p>Relationships: CanalYouTube implements Sujeto. Observador is implemented by SuscriptorMovil and SuscriptorPC. CanalYouTube notifies Observador.</p>

Patrones de Comportamiento

Patrón	Descripción	Cuando Usarlo	Diagrama UML
Command	Encapsula acciones como objetos	<ul style="list-style-type: none">• Deshacer/rehacer acciones• Encolar operaciones• Sistema de macros• Ejemplo: control remoto con botones programables	<pre>classDiagram class ControlRemoto { +Comando boton +setComando(c : Comando) +presionarBoton() } class Comando { +ejecutar() } class ComandoEncenderLuz { +Luz luz +ejecutar() } class ComandoApagarLuz { +Luz luz +ejecutar() } class Luz { +encender() +apagar() } ControlRemoto --> Comando : ejecuta Comando < .. ComandoEncenderLuz Comando < .. ComandoApagarLuz ComandoEncenderLuz --> Luz : usa ComandoApagarLuz --> Luz : usa</pre> <p>The UML diagram illustrates the Command pattern for a remote control system. It features four classes: ControlRemoto, Comando, ComandoEncenderLuz, and ComandoApagarLuz, along with a Luz class. ControlRemoto (Invoker) contains a Comando object and methods setComando(c : Comando) and presionarBoton(). A note indicates it calls ejecutar() without knowing details. Comando (Interface) defines the ejecutar() method. ComandoEncenderLuz and ComandoApagarLuz (Concrete Commands) implement ejecutar() by calling encender() or apagar() on a Luz object. Luz (Receiver) contains encender() and apagar() methods. A note describes it as the real receiver that knows how to turn the light on or off. Relationships include a solid arrow from ControlRemoto to Comando labeled "ejecuta", dashed arrows from Comando to the concrete command classes, and solid arrows from the concrete command classes to Luz labeled "usa".</p>