# Unidad 2: Diseño de sistemas de información orientado a objetos

## Introducción al diseño

En el desarrollo de software, el diseño es el proceso iterativo mediante el cual se aplican diversas técnicas y principios con el objetivo de definir un sistema con suficiente nivel de detalle para permitir su realización física, teniendo en cuenta las restricciones del negocio.

Características:

- modelo físico, porque es un plano de la implementación, con mayor detalle que el modelo conceptual que se obtiene del análisis
- específico para una implementación, no genérico
- más formal
- dinámico y centrado en la secuencia
- manifiesto del diseño del sistema, incluyendo la arquitectura

En un sistema de información se diseñan:

- Arquitectura: diseñar la arquitectura fundamental de los componentes del sistema, para dar respuesta a los requerimientos de calidad, con decisiones estratégicas.
- Datos: transformar los requerimientos en estructuras de datos necesarias para hacer persistente el software.
- Procesos: transformar elementos estructurales en procedimientos que lleven a cabo las tareas del sistema.
- Experiencia del usuario: cómo los usuarios interactúan con el sistema, creando interfaces de usuario intuitivas y atractivas.
- Formas de entrada/salida: cómo se ingresa la información al software y cómo se presentan las salidas.
- Procedimientos manuales: cómo se integra el software con procesos manuales en el entorno de negocio.

### Buen diseño

El diseño deberá implementar todos los requisitos del modelo de análisis y del cliente. Funciona como una guía para quienes generan código y quienes lo prueban y dan soporte. Para ello, debe proporcionar una imagen completa del software, teniendo en cuenta el comportamiento, aspectos funcionales y de datos, siempre desde una perspectiva de implementación.

Características de un buen diseño:

- Independencia de componentes: contar con bajo acoplamiento (que las relaciones entre clases sean las menores posibles) y alta cohesión (que todos los métodos y atributos de una clase esten altamente relacionados entre sí), para que los componentes no se vean afectados entre sí, sean reutilizables y de fácil evolución.
- Prevención y tolerancia de defectos: prevenir errores y hacer que causen el menor daño posible cuando no se pueden eliminar por completo.
- Identificación y tratamiento de excepciones: cómo se va a responder ante defectos que puedan ocurrir.

# Principios de diseño

Los principios de diseño en el desarrollo de software orientado a objetos son pautas y reglas que se siguen para crear sistemas de software más eficientes, mantenibles y flexibles. Estos principios guían la organización y estructura del código y promueven la creación de software de alta calidad.

- DRY - Don't repeat yourself: se trata de no duplicar código. Si una pieza de código se repite en diferentes lugares, se busca abstraerlo para escribirlo una sola vez y que sea fácil de corregir sin afectar al resto del código, logrando que cada requerimiento esté en un único lugar y quienes lo necesiten utilicen esta abstracción.
  ➔ Está relacionado a la alta cohesión (haciendo que una clase tenga una sola razón para cambiar) y evita inconsistencias.
- Separación de intereses: aislar la aplicación en secciones separadas, donde cada una aborda una preocupación distinta con poca superposición con el resto. Se puede aplicar a nivel arquitectónico separando en varias capas, o a nivel de programación separando servicios en diferentes niveles.
  ➔ Gestiona la complejidad y aumenta la mantenibilidad, permite la reutilización, mejora la cohesión.
- YAGNI - You ain't gonna need it - No lo vas a necesitar: no desarrollar código por suponer que va a ser útil en el futuro, es muy probable que las necesidades cambien con el tiempo y sea una pérdida de tiempo.
- KISS - Keep it simple: no aumentar infinitamente el nivel de abstracción ya que incrementa la complejidad. Los datos deben procesarse con la precisión necesaria y suficiente para una solución de alta calidad.
  ➔ Hace que sea más fácil de mantener, mejorar y ampliar el código.
- Menor conocimiento - No hables con extraños: cada unidad debe tener conocimiento limitado sobre otras unidades. El objetivo es tener bajo acoplamiento, y se logra manteniendo un número mínimo de dependencias, aprovechando las relaciones preexistentes y haciendo que los objetos "sepan poco" sobre los métodos de otras clases.
  ➔ Se logra mayor legibilidad y acoplamiento.
- Tell, don't ask: basado en el polimorfismo, en lugar de que un objeto solicite datos a otro objeto para aplicar la lógica sobre los mismos, se le indica directamente al objeto que realice una acción u operación.
  ➔ Se logra alta cohesión y bajo acoplamiento.
- Programar hacia la interfaz: una interfaz permite definir comportamiento que aplica a múltiples tipos y hacer foco en las clases que la utilizan. Las interfaces se usan si existen múltiples implementaciones concretas y para desacoplar el sistema de sistemas externos.
  Programar hacia la interfaz significa utilizar la interfaz de una clase en lugar de depender directamente de la implementación concreta de dicha clase. Es decir, en lugar de utilizar una clase específica, se utilizan las interfaces que definen el comportamiento esperado.
- Reutilización compuesta: promueve la reutilización de código a través de la composición en lugar de la herencia. En lugar de crear jerarquías de clases complejas mediante la herencia en tiempos de compilación, las clases deben lograr la reutilización de código combinando múltiples clases existentes en tiempos de ejecución. La composición permite que las relaciones entre clases se definan dinámicamente y no rompe el encapsulamiento, lo que facilita la creación de sistemas más flexibles y mantenibles.

## Principios SOLID

- **S**ingle Responsibility: Una clase debe tener una única razón para cambiar. Cada clase debe ser responsable de una única parte de la funcionalidad que brinda el software, y esa responsabilidad debe quedar encapsulada por la clase.
  ➔ Fomenta la alta cohesión y reduce la complejidad.
- **O**pen Close: Las clases deben estar abiertas para la extensión, pero cerradas para la modificación. Esto significa que el comportamiento de una clase debe poder ser extendido a través de una herencia o realización, sin necesidad de modificar el código fuente existente.
  ➔ Otorga flexibilidad y aumenta la cohesión.
  - Relacionado con este, tenemos también el principio de Encapsular lo que varía. Se trata de identificar los aspectos de la aplicación que pueden variar y separarlos de los que se mantienen inalterables, con el objetivo de minimizar las consecuencias de los cambios.
- Sustitución de **L**iskov: una clase derivada (o hija) debe poder ser usada en lugar de la clase base sin cambiar el comportamiento esperado del programa. Es decir, las clases hijas deben poder responder a los métodos que tiene la clase padre extendiendo su comportamiento.
  ➔ Así nos aseguramos de que la herencia está bien diseñada.
- Segregación de **I**nterfaces: la clase o el módulo que utiliza una interfaz no debe ser obligado a implementar métodos que no necesita o no utiliza. Se sugiere dividir las interfaces grandes en interfaces más pequeñas y específicas para cada cliente.
  ➔ Fomenta el bajo acoplamiento y alta cohesión, facilidad de implementación y prueba.
- Inversión de **D**ependencias: Las clases de alto nivel (las que contienen la lógica de negocio) no deben depender de clases de bajo nivel (las que implementan operaciones básicas, como acceder a una base de datos), ambas deben relacionarse con las abstracciones y no con concreciones. Separar la lógica de negocio de la lógica de implementación.
  ➔ Favorece al bajo acoplamiento, da flexibilidad, robustez y movilidad.

# Patrones de diseño

Los patrones de diseño son soluciones probadas y flexibles para problemas comunes en el diseño de software. Proporcionan un enfoque estructurado promoviendo la reutilización, eficiencia y claridad.
Son descripciones de clases y objetos relacionados que están particularizados para resolver un problema de diseño general en un determinado contexto.

## ¿Por qué usamos patrones?

Los patrones ayudan a resolver problemas que no podríamos resolver intuitivamente, cómo encontrar clases de fabricación pura, de granularidad fina, que favorezcan a principios del buen diseño, aplicando los principios de diseño.
Ayudan a encontrar objetos, especificar la implementación de las clases e interfaces, favorecen a la reutilización y la delegación de responsabilidades.

## Tipos de patrones según Gamma

### Patrones de creación

Abstraen el proceso de creación de instancias, disminuyendo la complejidad de este proceso. Ayudan a hacer un sistema independiente de cómo se crean, componen y representan sus objetos.

- Singleton

### Patrones estructurales

Se ocupan de cómo se combinan las clases y los objetos para formar estructuras más complejas. Explican cómo ensamblar objetos y clases manteniendo la flexibilidad y eficiencia de la estructura.

- Adapter

### Patrones de comportamiento

Se encargan de una comunicación efectiva y la distribución de responsabilidades entre objetos, apuntando a la alta cohesión y bajo acoplamiento. Describen el comportamiento y las interacciones entre los objetos y se utilizan para resolver problemas relacionados con la comunicación y la colaboración efectiva.

- Observer
- State
- Strategy
- Template Method
- Iterator

## State

### Síntesis

Patrón de diseño de comportamiento que permite a un objeto alterar su comportamiento cuando su estado interno cambia.

### Problema

El comportamiento de un objeto que depende de su estado implementado con operadores condicionales es difícil de mantener, ya que cualquier cambio en la lógica de transición entre estados puede requerir cambiar los condicionales de cada método.

### Solución

Sugiere que se creen clases nuevas para todos los estados posibles de un objeto y se extraigan todos los comportamientos específicos del estado para colocarlos dentro de esas clases.
En la clase padre se van a definir todos los métodos en su forma trivial, de forma que todos los estados concretos los hereden y aquellos que lo necesiten, van a redefinirlos.
Todas las clases de estado deben seguir la misma interfaz para que se puedan concretar las transiciones de estado.
El objeto original almacena una referencia a un objeto de estado y delega todo el trabajo relacionado con el estado a ese objeto.

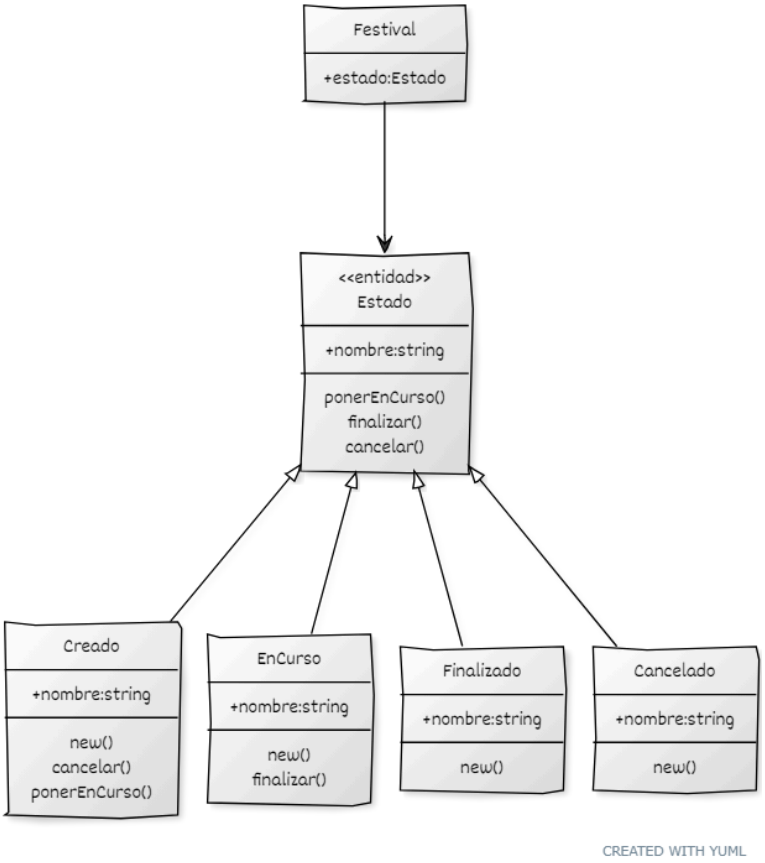
### Ventajas

- ★ Don't repeat yourself: promueve la reutilización en lugar de duplicar código relacionado con varios estados. El comportamiento "trivial" es heredado de la clase padre.
- ★ Tell don't ask: el objeto contexto no pregunta cual es su estado actual antes de realizar una acción, sino que notifica al objeto estado sobre algún evento y el estado maneja esa situación.
- ★ Responsabilidad única: organiza el código relacionado con estados concretos en clases separadas. Cada clase de estado concreto define el comportamiento en un estado específico.
- ★ Open - close: permite introducir nuevos estados sin cambiar los preexistentes o la clase contexto.
- ★ Sustitución de Liskov: Si se adhiere al contrato de interfaz del estado, cualquier implementación específica de estado puede reemplazarse sin problemas.
- ★ Inversión de Dependencias: el contexto depende de una abstracción de estado en lugar de sus clases hijas o estados concretos.

### Desventajas

- ★ Se rompe el encapsulamiento al aplicar herencia, ya que las clases hijas o estados concretos pueden tener accesos a partes del objeto de contexto que deberían estar ocultas.

## Ejemplo




En la clase padre se definen todos los métodos a los que un estado concreto podría responder en su forma más trivial, es decir la menos compleja. Así, aplicamos el principio **DRY**: en los estados en los que no se puede cancelar (en curso, finalizado y cancelado) no necesitamos escribir el mismo código una y otra vez, sólo debemos redefinirlo en el estado creado, para que efectivamente se cancele el pedido.

Cuando necesitemos efectuar una operación, el festival no va a averiguar cuál es su estado actual para saber cómo efectuar la operación, sino que le dirá al estado que efectúe la operación y este responderá de la forma en que se haya planteado según el estado que sea. De está forma se aplica el principio **Tell don't ask.**

Cada estado concreto tiene la responsabilidad de saber cómo actuar en ese único estado, en vez de que la clase festival tenga la responsabilidad de saber cómo actuar en cada uno de los estados posibles, lo cual corresponde con el principio de **responsabilidad única.**

Se aplica el principio **open close** ya que si deseamos agregar un nuevo estado posible, no necesitamos cambiar el código del resto de los estados o del festival, sino crear una nueva clase hija de Estado.

Se aplica el **principio de Liskov** ya que sin importar en qué estado esté el festival, su estado va a poder responder a cualquier método definido en la clase padre.

Como el festival se relaciona con la abstracción, la clase padre, no depende de las concreciones, aplicando **inversión de dependencias.**

## Strategy

### Síntesis

Patrón de diseño de comportamiento que define una familia de algoritmos, encapsula cada uno, y permite intercambiarlos. Permite que un algoritmo varíe independientemente de los clientes que los usan.

### Problema

Es útil cuando se tienen múltiples algoritmos para realizar una tarea y se desea seleccionar y cambiar entre ellos en tiempo de ejecución. Si todos los algoritmos están en una sola clase, un cambio en cualquiera de ellos aumenta las posibilidades de crear un error.

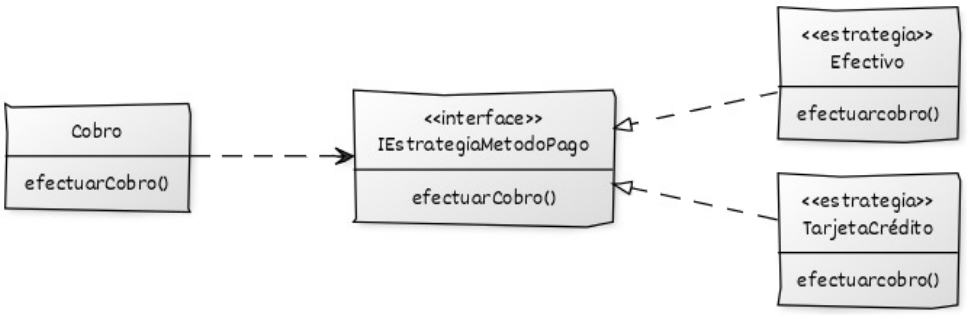
### Solución

Sugiere extraer los algoritmos de la clase y colocarlos en clases separadas llamadas estrategias. La clase contexto almacena una referencia a una de las estrategias y delega el trabajo a la misma. De esta forma, la estructura plantea un método polimórfico que implementa distintos algoritmos dependiendo de la situación.

### Ventajas

- ★ Single responsibility: cada estrategia tiene una única responsabilidad, que es implementar un algoritmo específico.
- ★ Open close: se pueden agregar nuevas estrategias sin cambiar las clases que las utilizan ni las estrategias preexistentes.
- ★ Tell don't ask: en lugar de que el contexto realice comprobaciones y tome decisiones, delega la responsabilidad de ejecutar un algoritmo u otro a la estrategia concreta.
- ★ Don't repeat yourself: evita la duplicación de código al encapsular algoritmos en clases separadas. En lugar de repetir el mismo código en múltiples lugares con variaciones, se reutiliza el código común; y no es necesario preguntar en cada lugar qué estrategia usar.
- ★ No lo vas a necesitar: este principio se cumple si creamos las estrategias concretas que necesitamos hoy, y no otras que "podrían llegar a utilizarse" en el futuro.

### Ejemplo




Cada estrategia concreta tiene la responsabilidad única de saber cómo efectuar el cobro según el método que le corresponda, en lugar de que el Cobro sepa cómo efectuar todos los tipos de cobros posibles, por lo que se aplica el principio de **responsabilidad única**.

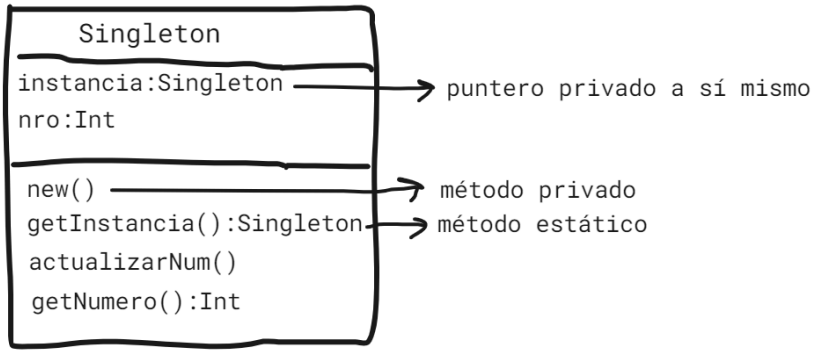
Si deseamos agregar un nuevo método de pago, no es necesario que modifiquemos los preexistentes ni la clase Cobro, simplemente agregamos una estrategia concreta nueva, aplicando el principio **open close**.

La clase Cobro pide a la estrategia que realice el cobro, en lugar de hacer comprobaciones para efectuar el cobro ella misma, por lo que se aplica **tell don't ask**.

## Singleton

### Síntesis

Patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.




### Problema

Existen clases que, de tener más de una instancia en el mismo momento, puede generar errores al utilizar un recurso compartido, como una base de datos o un archivo, por lo que se necesita asegurar que habrá una única instancia y controlar el acceso global a la misma.

### Solución

Extraer en una clase lo que deseamos que tenga una sola instancia, hacer privado el constructor para que solo la misma clase pueda crearlo y crear un método estático (al que solo puede responder la clase, no una instancia) que retorne la instancia única o la cree si no existe.

### Ventajas

- ★ Single responsibility: el singleton se encarga de crear y gestionar su única instancia, asegurando que la clase tenga la responsabilidad única de administrar su ciclo de vida y acceso.
- ★ Open close: se pueden tener subclases que agreguen funcionalidades al singleton sin alterar la clase original.
- ★ Inversión de dependencias: se proporciona una manera de acceder a la instancia sin exponer los detalles de cómo se crea la misma. La clase de alto nivel (el contexto) depende de una abstracción que representa la instancia.

*No creo que pidan ejemplo, es siempre igual*

## Iterator

### Síntesis

Patrón de diseño de comportamiento que proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.

### Problema

Independientemente de cómo se estructure una colección, debe aportar una forma de acceder a sus elementos de modo que otro código pueda utilizar dichos elementos. Se necesita una forma estandarizada de acceder secuencialmente a los elementos de una colección sin exponer su estructura.

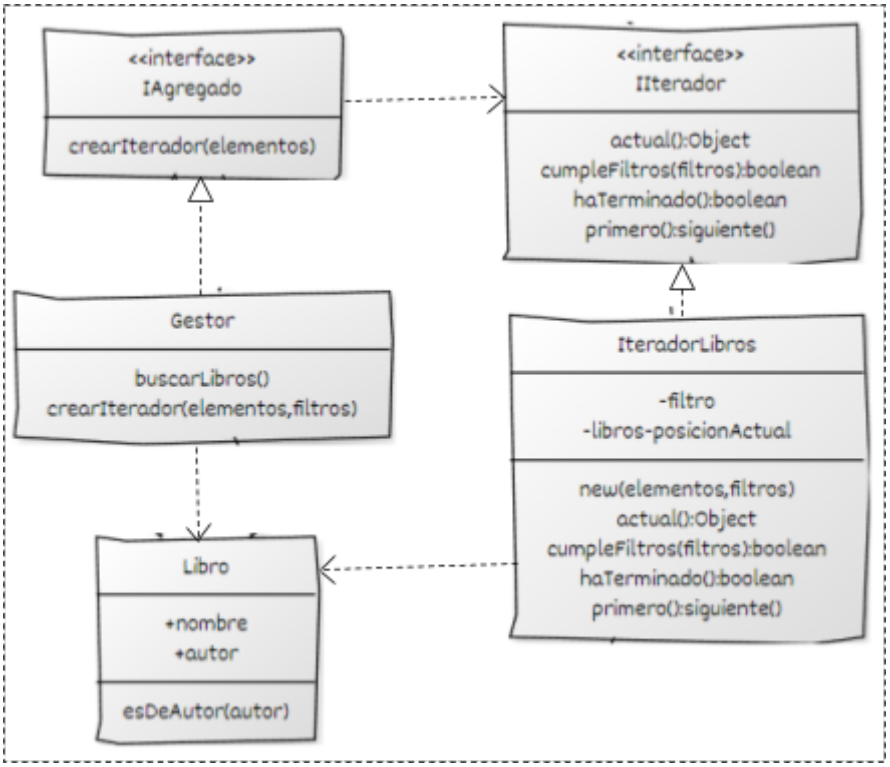
### Solución

La idea central es extraer el comportamiento de recorrido de una colección y colocarlo en un objeto independiente llamado iterador.
Además de implementar el algoritmo de iteración, el iterador encapsula los detalles como la posición actual o la cantidad de elementos que quedan por recorrer.
La interfaz Iteradora declara las operaciones necesarias para recorrer una colección: extraer el siguiente elemento, recuperar la posición actual, etc.
Los Iteradores Concretos implementan algoritmos específicos para recorrer una colección.
La Interfaz Agregado declara uno o varios métodos para obtener iteradores compatibles con la colección.
Los Agregados Concretos son los que anteriormente se encargaban de recorrer los elementos, ahora se encargan de crear el iterador que lo haga.

### Ventajas

- ★ Responsabilidad única: se extraen los algoritmos de recorrido del cliente y se los coloca en clases independientes que se encarguen exclusivamente de gestionar la iteración y acceso a los elementos.
- ★ Tell don’t ask: se delega la responsabilidad de iterar, de manera que el cliente indica que se debe iterar la colección pero no se preocupa por cómo lo hará el iterador.
- ★ Open close: se pueden agregar nuevos tipos de iteradores para colecciones nuevas sin necesidad de afectar a los existentes.
- ★ Inversión de dependencias: el cliente depende de una abstracción (la Interfaz Iterador) en vez de relacionarse directamente con los iteradores concretos.
- ★ Segregación de interfaces: se proporciona una interfaz simple y específica para recorrer colecciones, sin agregar métodos innecesarios.

Ejemplo




El gestor delega la responsabilidad de recorrer una lista de libros a una clase que se encarga específicamente de ello, respetando el principio de **responsabilidad única** y **tell don't ask**, ya que el gestor le indica al iterador que se debe recorrer la lista pero no se preocupa por los detalles de cómo este lo hará.

Si desearamos recorrer otras colecciones nuevas, como por ejemplo bibliotecas, simplemente deberíamos agregar un nuevo iterador concreto, sin necesidad de cambiar el código de nuestro iterador de libros o de la clase Biblioteca, lo que contribuye con el principio **open close.**

El gestor no depende de el iterador de libros, si no de la interfaz iteradores, por lo que, al relacionarse con una abstracción, sigue el principio de **inversión de dependencias**.

La interfaz iterador solo cuenta con los métodos esenciales para recorrer una colección, por lo que no obliga a los iteradores concretos a implementar métodos que no utilizan, aplicando el principio de **segregación de interfaces**.

## Template Method

### Síntesis

Patrón de diseño de comportamiento que define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura.

### Problema

La existencia de múltiples clases que resuelven un mismo problema de diferentes formas pero con pasos y secuencias similares, puede producir duplicación de código con mínimas diferencias, además de tener muchos condicionales que eligen un curso de acción dependiendo de la clase del objeto de procesamiento.

### Solución

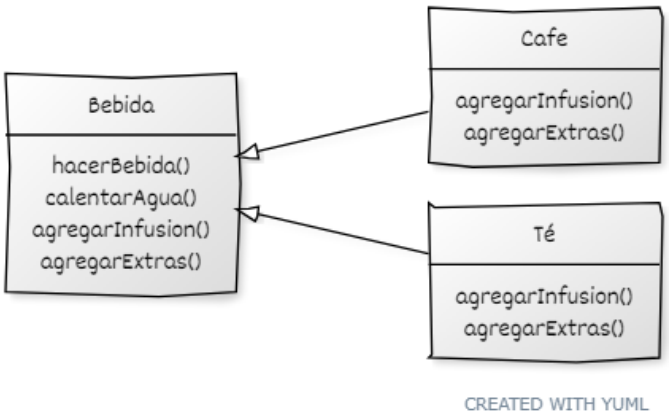
Se sugiere dividir un algoritmo en una serie de pasos, convertir cada paso en un método y colocar las llamadas a los mismos en un método plantilla.
Las operaciones que son similares para todos los métodos se definen en la clase base (métodos hook), eliminando la duplicación de código, mientras que los pasos que varían se redefinen en cada clase concreta, siendo métodos polimórficos.

### Ventajas

- ★ Inversión de dependencias: permite que las clases concretas implementen pasos específicos en el algoritmo sin depender directamente de la clase base (abstracta) que define la estructura del algoritmo.
- ★ Open close: se pueden agregar nuevas formas de resolver el problema sin afectar a las preexistentes, lo que permite agregar nuevas funcionalidades sin cambiar la clase base.
- ★ Sustitución de Liskov: se respeta siempre que las clases concretas, al proporcionar sus implementaciones para los pasos del algoritmo, cumplan con el contrato definido por la clase base.

### Ejemplo

El método plantilla en este caso es hacerBebida(), que llama a cada uno de los pasos necesarios. Como calentar agua es necesario tanto para el café como para el té, este método hook será escrito en la clase padre. En cambio, agregar infusión y extras varían dependiendo del tipo de bebida, por lo que deberá ser redefinido en cada clase hija, siendo métodos polimórficos.




De esta forma, las clases concretas café y té implementan pasos específicos sin depender de lo escrito en la clase padre, aplicando **inversión de dependencias**.
Si deseamos agregar otra bebida que cumpla con esta estructura de pasos (para que se aplique la **sustitución de Liskov**), solo debemos asegurarnos que siga el contrato definido por la interfaz y no debemos cambiar otra de las clases ni la clase padre, lo que cumple con el principio **open close**.

## Observer

### Síntesis

Patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.

### Problema

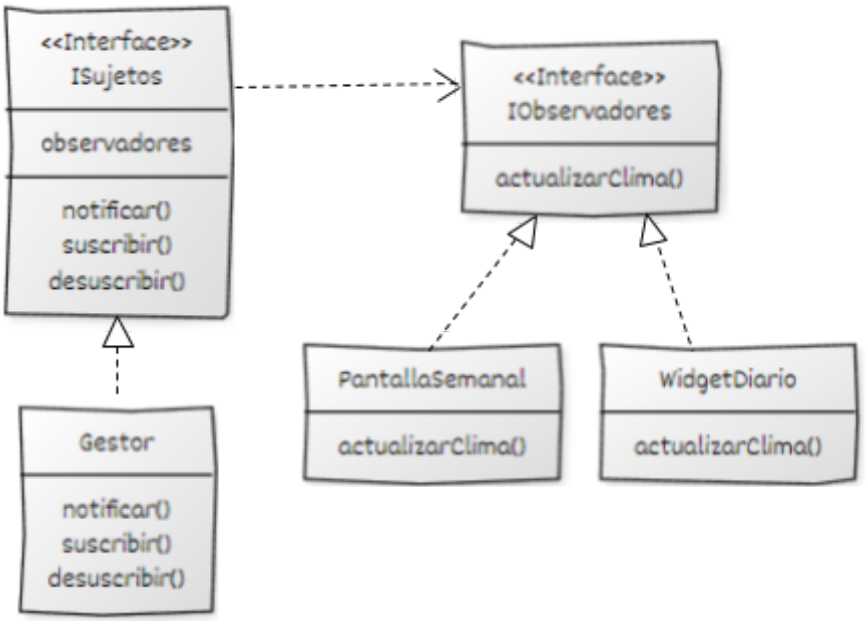
Aborda la necesidad de establecer una relación de notificación eficiente y desacoplada entre objetos, sin necesidad de hacer verificaciones constantes y enviando la información a quienes realmente están interesados en ella.

### Solución

Sugiere añadir un mecanismo de suscripción a la clase cuyos cambios son de interés (el sujeto) para que otros objetos (observadores) puedan suscribirse a un flujo de eventos.
El cambio consiste en que el sujeto tenga una lista de referencias a sus observadores, un método que permita recorrerlos y llame al método de los observadores que utilizan la información, y otros que puedan suscribirlos o eliminarlos.

### Ventajas

- ★ Open close: permite que los sujetos y observadores puedan cambiar y evolucionar independientemente, sin requerir modificaciones en el otro. Los sujetos no necesitan conocer detalles de los observadores y viceversa.
- ★ Separación de intereses: permite que los sujetos se centren en su propia lógica mientras que los observadores se centran en sus acciones en respuesta a los cambios del sujeto.
- ★ Inversión de dependencias: los observadores dependen de una abstracción (la interfaz de observador), y los sujetos se comunican con observadores sin conocer las implementaciones específicas de estos observadores.




*Está clarito como se aplica cada principio no me hagan escribir lo mismo dos veces.*

# Adapter

## Síntesis

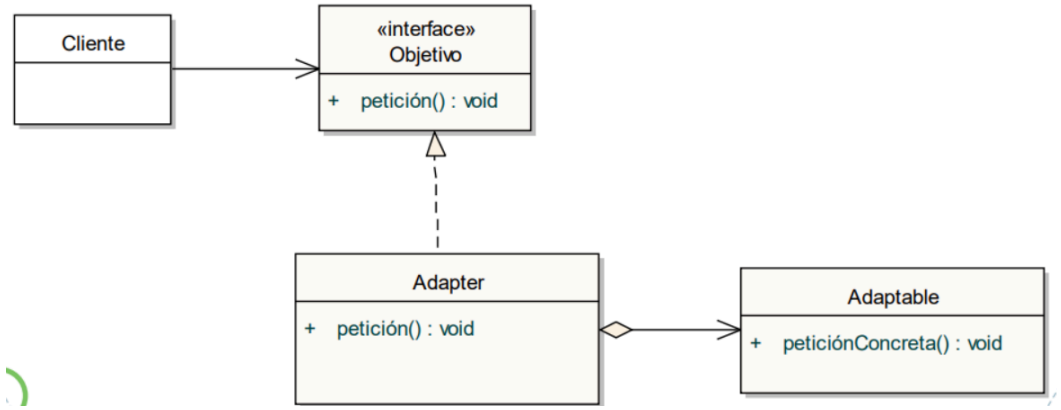
Patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles. Convierte la interfaz de una clase en otra interfaz que es la que esperan los clientes.

## Problema

Es común que una clase tenga que interactuar con otra que no es compatible con ella debido a diferencias en la interfaz, métodos o estructura de datos.

## Solución

La idea es crear un objeto especial que convierte la interfaz de un objeto, de manera que otro objeto pueda comprenderla. Este adaptador crea una capa adicional alrededor del objeto para esconder la complejidad de la conversión.



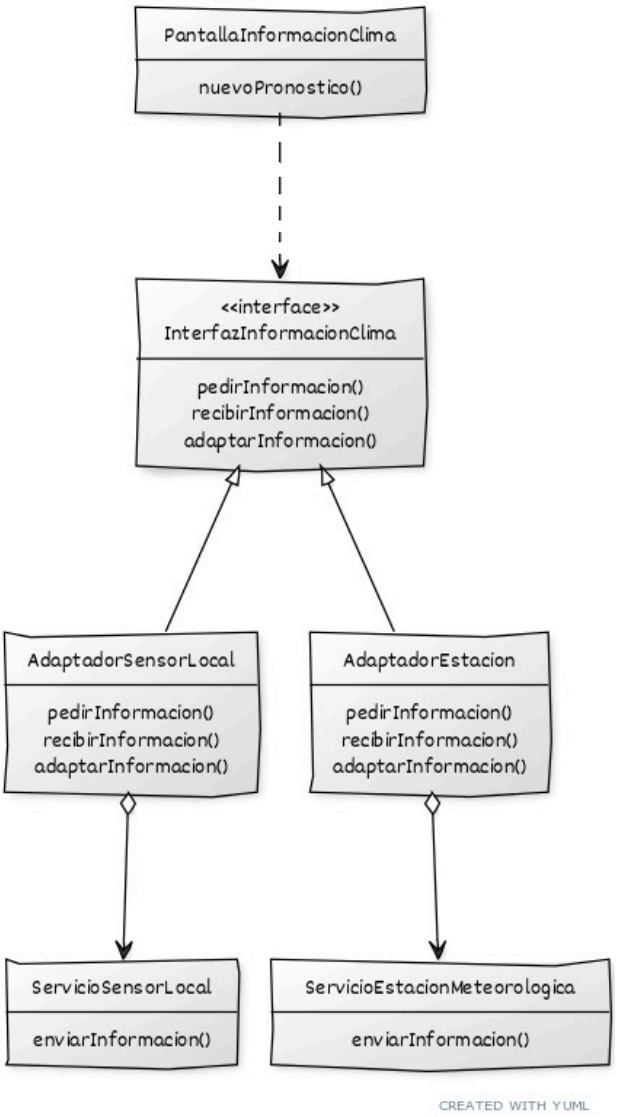

## Ventajas

- ★ Responsabilidad única: se crea una clase que se encargue de la conversión, en lugar de que esta responsabilidad caiga sobre el cliente.
- ★ Open close: se pueden introducir nuevos tipos de adaptadores sin descomponer el código cliente existente, siempre y cuando trabajen con los adaptadores a través de la interfaz con el cliente.
- ★ Tell don't ask: en lugar de buscar los datos para realizar la conversión, el cliente indica al adaptador qué acción realizar y no necesita preocuparse por cómo se lleva a cabo internamente.

### Ejemplo

La pantalla que va a crear un pronóstico del clima, delega la responsabilidad de convertir los distintos formatos en los que llega la información desde diferentes fuentes. De esta forma los adaptadores concretos son los nuevos encargados de esta tarea, aplicando el principio de **responsabilidad única** y **tell don't ask** ya que la pantalla no se preocupa por cómo será convertida la información.

Si se desea recibir información de una nueva fuente con otro formato, no es necesario cambiar el código de la pantalla que la recibirá ni de los adaptadores existentes, lo cual se alinea con el principio **open close**.




## Diseño de Persistencia

La persistencia es la capacidad de almacenar objetos mayores que una ejecución de programa, es decir que el objeto sobreviva a la ejecución que lo creó. La persistencia frecuentemente significa que los objetos se copian desde una memoria primaria rápida y volátil a una memoria secundaria lenta y persistente.

### Problema de impedancia

En una base de datos relacional, que son la mayoría, la información se almacena en tablas con tipos de datos primitivos tales como caracteres, enteros, etc. Esto trae algunos problemas a nuestra necesidad de almacenar objetos:

- solo los datos pueden almacenarse y no el comportamiento
- solo los tipos de datos primitivos pueden almacenarse y no las estructuras complejas y relaciones de nuestros objetos.

En nuestro sistema toda la información se almacena en los objetos, por lo tanto necesitamos transformar nuestra estructura de información de objeto a una estructura orientada a tablas.
Por otro lado, el problema de impedancia crea un fuerte acoplamiento entre la aplicación y el gesto de bases de datos. Para hacer que el diseño sea mínimamente afectado por el DBMS, tan pocas partes de nuestro sistema como sea posible deberían saber sobre la interfaz del DBMS.

## Relaciones entre clases

### Asociación

Las bases de datos relacionales permiten asociar tablas a través de claves foráneas, representando relaciones uno a uno, uno a muchos, muchos a uno, o con tablas intermedias para relaciones muchos a muchos.

### Herencia

Existen tres posibilidades para mapear la herencia a una base de datos relacional:

- simular la herencia: se mantienen todas las tablas
- eliminar la herencia eliminando al padre
- eliminar la herencia eliminando a los hijos

## Modelos de persistencia

Los modelos de persistencia son un conjunto de patrones y estrategias que se utilizan para almacenar objetos de un programa en una base de datos. Se trata de módulos encargados de materializar y desmaterializar objetos y tablas y comunicarse con la base de datos.

- Super Objeto Persistente: se define un objeto que tiene la capacidad de ser almacenado en una base de datos, y todos los objetos persistentes heredan su comportamiento. Hoy en día no tiene mucho uso, ya que es poco flexible y extensible. No hace que crezca la estructura de clases.
- Esquema de persistencia: se trata de una capa que define cómo se mapean las clases y objetos del programa en tablas y estructuras de datos. La ventaja más importante es su extensibilidad, además de la reusabilidad y la cohesión entre las clases. No hace que crezca la estructura de clases.

## Frameworks

Es un conjunto extensible de clases e interfaces que colaboran para proporcionar servicios de un subsistema lógico.
Contiene clases concretas y (especialmente) abstractas que definen las interfaces a las que ajustarse, interacciones de objetos en las que participar, y otras variantes.
Confía en el Principio de Hollywood: “No nos llame, nosotros lo llamaremos”.
Ofrecen un alto grado de reutilización, mucho más que con clases individuales.

# Unidad 3: Diseño de arquitectura de software

El diseño arquitectónico es la asignación de modelos de requerimientos esenciales a una tecnología específica.
La arquitectura de software puede entenderse como el conjunto de decisiones que condicionan la escalabilidad, mantenibilidad, robustez del software a partir de los requerimientos no funcionales.
Es el diseño de más alto nivel de la estructura de un sistema, el cual consiste en un conjunto de patrones y abstracciones que proporcionan un marco claro para la implementación del sistema.
Diseñar la arquitectura del software es esencial ya que proporciona una estructura organizada y comprensible del sistema, facilitando su desarrollo, mantenimiento y escalabilidad. Una arquitectura sólida también promueve la reutilización de componentes y contribuye a un rendimiento eficiente y una mayor seguridad. Además, sirve como base para la toma de decisiones informadas y garantiza que el software cumpla con los requisitos y expectativas del proyecto, lo que es fundamental para su éxito a largo plazo.

## Documentación

El arquitecto de software revisa y negocia los requerimientos, documenta, comunica y verifica el cumplimiento de la arquitectura. Produce la arquitectura que conducirá al sistema.
La documentación de la arquitectura de software es una parte esencial del proceso de diseño y desarrollo de sistemas, ya que proporciona una guía detallada sobre cómo está estructurado y organizado el sistema. Incluye diagramas, descripciones y detalles técnicos que permiten a los miembros del equipo entender la lógica interna del sistema, las relaciones entre componentes y cómo se cumplen los requisitos.
Las razones para documentar y comunicar a cada miembro del equipo la parte de la arquitectura en la que se debe interesar son:

- escalabilidad y mantenibilidad
- asegurar que se respeten las decisiones tomadas
- que cada miembro esté enterado de cómo afecta la arquitectura a su trabajo.