Introducción a Objetos

Programación Orientada a Objetos

Método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son, todas ellas, miembros de una jerarquía de clases unidas mediante relaciones de herencia.

Un programa orientado a objetos es un conjunto de objetos que colaboran enviándose mensajes. Esta definición implica que sólo hay objetos que lo único que pueden hacer es enviar y recibir mensajes. Si algo se quiere hacer, se necesita un objeto que lo haga y otro que le envíe un mensaje.

Un lenguaje es orientado a objetos si y solo si:

- soporta objetos que son abstracciones de datos con una interfaz de operaciones con un nombre y un estado local oculto,
- los objetos tienen un tipo asociado (clase),
- los tipos (clase) pueden heredar atributos de los supertipos (superclases).

En este paradigma, el marco conceptual es el modelo de objetos, donde hay cuatro elementos fundamentales, que son:

- **Abstracción:** descripción simplificada de un sistema que enfatiza algunos de sus detalles o propiedades mientras suprime otros. Se enfoca sobre una vista externa del objeto, y sirve para separar su comportamiento esencial de su implementación.
- **Encapsulamiento:** Proceso de ocultamiento de todos los detalles de un objeto que no contribuyan a sus características esenciales.
- **Modularidad:** Proceso de partición de un programa en componentes individuales. Esto reduce la complejidad y crea un número de contornos bien documentados dentro del programa.
- **Jerarquía:** Ordenamiento de abstracciones. Hay dos tipos fundamentales: estructura de clases (“tipo de”) y de objetos (“parte de”).

Tipo Abstracto de Datos

Descripción abstracta y formal que define un número de operaciones aplicables, su sintaxis (modo en que puede invocarse cada operación, donde se define el nombre, el dominio y el rango de operaciones) y su semántica (cómo opera cada operación de acuerdo a un conjunto de axiomas). Esta noción matemática define un tipo de datos.

Todo TAD debe cumplir con los principios de encapsulamiento (ocultación de la información) y abstracción de datos (separación de las propiedades lógicas de los datos de su representación o implementación).

Los TADs permiten la creación de instancias con propiedades y comportamiento bien definidos. En POO, los TADs reciben el nombre de Clases.

Objeto

Entidad que representa un elemento, unidad o entidad individual e identificable, ya sea real o abstracto, con un papel bien definido en el dominio del problema. Un objeto modela alguna parte de la realidad, y es por tanto algo que existe en el tiempo y el espacio, tiene una frontera definida con nitidez.

Además, un objeto tiene estado, comportamiento e identidad:

- **Estado:** todas las propiedades del mismo más los valores actuales de cada propiedad.
- **Comportamiento:** métodos o procedimientos con los cuales éste puede operar, es decir, qué operaciones pueden realizarse con él, qué es lo que sabe hacer.
- **Identidad:** propiedad que lo diferencia de los demás objetos.

La estructura y comportamiento de objetos similares están definidos en su clase común.

Un objeto encapsula funcionalidad e información: retiene cierta información y sabe cómo realizar ciertas operaciones. En un objeto, las operaciones y la información están juntas y sólo se puede acceder a esa información a través de esas operaciones.

Clase

Declaración o abstracción de un objeto que oficia de “molde”. Una clase define las propiedades de objetos, los cuales son instancias en un entorno orientado a objetos. Básicamente, los objetos que comparten el mismo comportamiento pertenecen a la misma clase.

Tipos de clase

- **Clase abstracta:** Clase sin instancia.
- **Clase concreta/hoja:** Clase especializada.
- **Clase base:** Clase más generalizada de la estructura de clases.

Instancia

Objeto que se comporta de la manera descripta en una clase. Todo objeto es instancia de alguna clase. Una instancia se comporta de la misma manera que las demás instancias de esa clase y almacena su información en variables de instancia.

Mensajes

Un mensaje es una solicitud de servicios entre objetos. Está compuesto por tres partes: identidad del receptor, método solicitado y parámetros (información necesaria para el método).

Métodos

Algoritmos particulares con los cuales el objeto realiza operaciones. Al recibir un mensaje, un objeto lleva a cabo la operación mediante la ejecución de un método. Cada método especifica cómo se lleva a cabo la operación para responder a un determinado mensaje. Los métodos pueden acceder a la estructura interna de un objeto y enviarse mensajes a si mismo o a los demás objetos.

Un método está asociado a un mensaje, generalmente con el mismo nombre.
Hay dos tipos de métodos: de clase y de instancia.

- **Métodos de clase:** porciones de código asociadas exclusivamente con una clase. También se los llama métodos estáticos.
- **Métodos de instancia:** métodos relacionados con un objeto en particular. Estos métodos reciben una referencia oculta al objeto al que pertenecen (“this” o “self”) para que puedan acceder a los datos asociados con el mismo.

Variables

Las variables almacenan datos en memoria durante la ejecución del programa. Hay tres tipos de variables en la POO: variables de instancia, variables de clase y pseudovariables.

Variables de instancia

Representan el estado del objeto, y perduran durante toda la vida de éste. Dos objetos diferentes pueden tener valores diferentes en sus variables de instancia aunque pertenezcan a la misma clase.

Variables de clase

Variables compartidas por las instancias de una clase y sus subclases, manteniendo el mismo valor para todas las instancias. Se declaran en la definición de la clase.

Pseudovariables

Identificador que referencia a un objeto. La diferencia con las variables normales es que no se pueden asignar y siempre aluden al mismo objeto. Una pseudovariable, al igual que las constantes, no puede modificarse con una expresión de asignación.

Método Constructor

Método especial necesario para poder operar en un objeto, ya que es el encargado de crearlo. Los constructores suelen programarse de forma tal que hacen dos operaciones a la vez: crear el objeto y darle valores a sus datos.

Es habitual la existencia de más de un constructor, según la sobrecarga de parámetros. De esta forma, se pueden crear objetos e inicializarlos de distintas formas.

El constructor por defecto o “constructor argumento-cero” es el tipo de constructor que no recibe ningún parámetro. Según el lenguaje, es posible definir una clase sin crear un constructor para la misma. La ausencia de este constructor, en caso de que el lenguaje soporte tal cosa, implica que solamente se reserve espacio en memoria para almacenar los datos del objeto, los cuales van a estar sin ser inicializados, ya que el constructor por defecto no va a saber qué valores son apropiados para los datos de la clase.

También es posible construir objetos a partir de otros objetos, aunque lo que en realidad ocurra es que esos objetos son referenciados.

Método plantilla (Template)

Método que define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Esto permite la redefinición de ciertos pasos sin la necesidad de cambiar la estructura del algoritmo. Al usar este método, se define una estructura de herencia en la cual la superclase sirve de plantilla de los métodos en las subclases. Esto evita la repetición de código, y en consecuencia, la aparición de errores, y lo vuelve útil en algoritmos comunes para muchas clases, pero con pequeñas variaciones entre una

y otras.

Método abstracto y clase abstracta

Un método abstracto es un método declarado en una clase para el cual esa clase no proporciona la implementación. Este método permite construir el esqueleto de un algoritmo dejando los detalles a las subclases. Permite escribir las partes fijas de un algoritmo una sola vez y definir las partes que varían en las subclases. Esto permite refactorizar el código y definir los puntos de extensión de las subclases.

Esto ayuda a la reutilización de código, y permite el soporte de frameworks o armazones invirtiendo el flujo de control. Un método template puede llamar a operaciones concretas (de los clientes o la clase concreta), operaciones concretas de la clase abstracta (operaciones útiles para las subclases), y operaciones primitivas, entre otros.

Una clase abstracta define las operaciones primitivas que deben implementar las subclases concretas. Estas clases tienen al menos un método abstracto. Las subclases de la clase abstracta deben implementar el código para esos métodos abstractos o volver a declararlos como abstractos, convirtiendo a la subclase en una clase abstracta.

Encapsulamiento

También llamado ocultación u ocultamiento de la información. Proceso de almacenar en un mismo objeto los elementos de una abstracción que constituyen su estructura (los colaboradores) y su comportamiento. Sirve para separar el interfaz contractual de una abstracción (qué hace) de su implementación (cómo lo hace).

En el encapsulamiento sólo se puede acceder a los valores del tipo que define o modificarlos mediante las operaciones abstractas definidas sobre ellos.

Responsabilidades

Transmisiones del sentido del propósito de un objeto y de su lugar en el sistema. Incluye dos elementos clave: el conocimiento que un objeto mantiene y las acciones que puede llevar a cabo.

Un objeto sabe cumplir su rol y el de ningún otro objeto dentro del sistema. Fuera de los objetos no existen ni informaciones ni operaciones.

Jerarquía de Clases

Ordenación o clasificación de clases.

Herencia

Relación entre clases que permite que unos objetos puedan basarse en otros existentes, así una clase X puede heredar propiedades de una clase Y de modo que los objetos de la clase X tengan acceso a los atributos y operaciones de la clase sin necesidad de redefinirlos. En esta relación, una clase comparte la estructura de comportamiento definida en una o más clases, lo que permite declarar las abstracciones con economía de expresión. Suele identificarse la herencia mediante la regla “es un” o “es un tipo de”. Esta relación crea automáticamente una jerarquía de especialización-generalización.

La herencia, al igual que las clases, cumple un rol conceptual (el modelado de relaciones “es un” entre abstracciones) y uno implementativo (la reutilización de código).

Superclase y subclase

La superclase es la clase padre de la relación de herencia. Es una generalización de las subclases. Aquí se definen los atributos comunes y los métodos que se implementan de igual manera para todas las clases.

La subclase es la clase hija de la relación. Es una especialización de la superclase. Aquí se definen los atributos diferentes y los métodos que se implementan de manera diferente. Una clase hija extiende o amplía el comportamiento de la clase padre, pero también restringe o limita a la clase padre.

Herencia simple

Tipo de herencia en el cual cada subclase tiene exactamente una superclase. Esto limita la aplicabilidad de las clases predefinidas, haciendo necesario muchas veces duplicar el código.

Herencia múltiple

Tipo de herencia en el cual se plantea un estilo de clases pequeñas que se mezclan y combinan para construir clases con un comportamiento más sofisticado llamadas clases aditivas. Sintácticamente, una clase aditiva es idéntica a una clase normal, pero su intención es distinta, ya que su función es la de añadir funciones a otras clases.

Clases abstractas y redefinición de métodos

La herencia permite que existan clases que nunca sean instanciadas directamente. Este tipo de clase recibe el nombre de clase abstracta. Su característica particular es la ausencia de instancias específicas, ya que para todo lo demás se la considera como una clase más.

Si en una clase en particular se invoca un método, y el método no está definido en la misma, es buscado en las clases superiores. Sin embargo, si existieran dos métodos con el mismo nombre y distinto código, uno en la clase y otro en la superclase, se ejecutaría el de la clase, no el de la superclase. Esta es la redefinición de métodos. Aún así siempre se puede acceder explícitamente al método de una superclase mediante una sintaxis específica, la cual varía según el lenguaje de programación empleado.

Estas herramientas ayudan a los programadores a ahorrar código y tiempo. Los objetos pueden ser construidos a partir de otros similares, aunque para ello es necesario que exista una clase base y una jerarquía de clases. La clase derivada, además, puede heredar código y datos de la clase base, añadiendo código o modificando lo heredado. Por último, las clases que heredan propiedades de otra clase pueden servir como clase base de otras clases.

Sobrecarga

Tipo especial de polimorfismo en el cual varios métodos (incluídos los métodos constructores) pueden tener el mismo nombre, siempre y cuando el tipo de parámetros recibidos (o su número) sea diferente.

Polimorfismo

Cualidad de los objetos para responder al mismo mensaje de distintas formas, haciendo que este tenga diferente comportamiento en diferentes objetos. El polimorfismo permite que el objeto emisor del mensaje se despreocupe de quién es su colaborador; al objeto emisor sólo le interesa que el objeto receptor sea responsable de llevar adelante la tarea encomendada a través del mensaje.

Colecciones

Estructuras que permiten almacenar objetos, pueden ser recorridas (iteradas), y pueden dar a conocer su tamaño. Las colecciones agrupan varios elementos en una sola unidad, son utilizadas para almacenar, recuperar y manipular los datos agregados, y representan generalmente elementos de datos que forman un grupo natural.

Framework

Arquitectura unificada para representar y manipular colecciones. Todos los frameworks contienen Interfaces, Implementaciones y Algoritmos.

- **Interfaces:** Tipos de datos abstractos que representan colecciones. Permiten manipular las colecciones de forma independiente de los detalles de su representación. En lenguajes orientados a objetos, generalmente forman una jerarquía.
- **Implementaciones:** Estructuras de datos reutilizables, implementaciones concretas de las interfaces de Collection.
- **Algoritmos:** Funcionalidad reutilizable. Métodos que realizan cálculos útiles, tales como búsqueda y selección, en los objetos que implementan interfaces de la colección. Son polimórficos, ya que el mismo método se puede utilizar en muchas implementaciones dependiendo de las diferentes interfaces que hayamos elegido.

Organización

Las interfaces y clases están relacionadas en un framework de colecciones para facilitar su uso.

- **Interfaces:** Manipulan los datos independientemente de los detalles de implementación
- **Clases:** Implementan las interfaces.

Para programar con colecciones se debe elegir una interfaz adecuada a la funcionalidad requerida, elegir una clase que implemente la interfaz y extender la clase si fuera necesario.

Tipos

Existen distintos tipos de colecciones: ordenadas, sin orden, indexadas, que permiten elementos repetidos, que no los permiten, y que almacenan un compuesto (clave,valor).

Iterator

Sirve para manipular los objetos/elementos. El iterador se usa para recorrer cualquier colección y eliminar un elemento. Tiene tres funciones:

- *boolean hasNext()* (devuelve true si la iteración tiene más elementos)
- *Object next()* (devuelve el siguiente elemento de la iteración)
- *void remove()* (elimina el último elemento devuelto por la iteración)

ListIterator

Extiende a Iterator y maneja un objeto List ordenado. Permite iterar hacia adelante y hacia atrás.

Collection (interfaz)

Representa un grupo de objetos. Sin implementaciones directas, define métodos para tratar una colección genérica de elementos.

Collections (clase)

Permite ordenar y buscar elementos en listas. Se usan los métodos sort() y binarySearch() junto a equals(), hashCode() y compareTo().

Implementa los siguientes métodos:

- *Object add(Object anObject)* – Agrega un objeto a la colección.
- *Collection addAll(Collection aCollection)* – Agrega cada elemento de la colección origen a la colección destino.
- *Object remove(Object anObject)* – Busca el elemento a quitar de la colección y lo quita. Si no lo encuentra, devuelve un error. Una variante de este método, en caso de no encontrar el elemento, lo busca en un bloque.
- *Collection removeAll(Collection aCollection)* – Quita todos los elementos de la colección origen en la colección destino.
- *int size()* – Devuelve el número de elementos contenidos en la colección.
- *boolean isEmpty()* – Retorna true si la colección no contiene elementos.
- *boolean notEmpty()* – Retorna true si la colección contiene elementos.
- Array asArray() – Retorna una instancia de Array conteniendo los elementos de la colección.
- Bag asBag() – Retorna una instancia de Bag conteniendo los elementos de la colección.
- OrderedCollection asOrderedCollection() – Retorna una instancia de OrderedCollection conteniendo los elementos de la colección.
- Set asSet() – Retorna una instancia de Set conteniendo los elementos de la colección.
- SortedCollection asSortedCollection() – Retorna una instancia de SortedCollection conteniendo los elementos de la colección. Una variante retorna la SortedCollection con los elementos ordenados de acuerdo a un bloque.
- boolean includes(Object anObject) – Retorna true si el receptor contiene un elemento igual al objeto pedido.
- int occurrencesOf(Object anObject) – Retorna la cantidad de elementos contenidos en la colección que son iguales al objeto pedido.
- Object detect(Block aBlock) – Retorna el primer elemento de la colección que causa que el bloque sea evaluado como true, con ese elemento como argumento. Si ningún elemento de la colección cumple con esa condición, se reporta un error. Una variante de este método evalúa un bloque de excepción.

Set

Conjunto. Colección que no puede tener objetos duplicados, es decir, que los elementos ocurren una sola vez. En un Set, el orden no es dato. Si bien es posible que existan Sets que nos aseguren un orden determinado cuando los recorremos, ese orden no es arbitrario y decidido por nosotros, ya que la interfaz no tiene ninguna funcionalidad para manipularlo, a diferencia de List.

Su ventaja es que preguntar si un elemento ya está contenido suele ser muy eficiente. Es conveniente usar esta colección cada vez que necesitemos una colección donde el orden no importe, pero que necesitemos preguntar si un elemento está o no.

Los Sets aprovechan una característica de Java: todos los objetos heredan de Object, por lo tanto, todos los objetos de Object están presentes en todos los objetos. Esto quiere decir que hay ciertas cosas que todo objeto en Java sabe hacer.

Desciende de la interfaz Collection. El polimorfismo aplica para todas las clases que implementan esta interfaz.

SortedSet

Set que mantiene los elementos ordenados según un criterio establecido. Desciende de la interfaz Collection. El polimorfismo aplica para todas las clases que implementan esta interfaz.

List

Colección ordenada (secuencia) en la que cada elemento ocupa una posición identificada por un índice. El primer índice es 0. Admite duplicados. Desciende de la interfaz Collection. El polimorfismo aplica para todas las clases que implementan esta interfaz.

Map

También llamado “diccionario” (Dictionary). Par de datos (clave,valor) en donde cada valor tiene un objeto extra asociado y puede ser accedido mediante su clave/key. No puede haber claves duplicadas, y cada clave se corresponde con al menos un valor. Internamente, una instancia de esta colección almacena pares clave/Valor como un conjunto de instancias de la clase Association. Suele estar asociada a la idea de “tabla de hashes”, aunque no se implemente necesariamente esta técnica.

SortedMap

Map que mantiene las claves ordenadas según un criterio establecido.

Queue

Maneja colas.

LinkedList

Maneja listas enlazadas. Los elementos de esta colección apuntan a su antecesor y al elemento que le sigue. Implementa la interfaz List.

ArrayList

Implementa la interfaz List.

HashSet

Colección que le pide a cada objeto que se añade el cálculo de su “hash”. Este hash es un número que va desde -2147483647 hasta 2147483648. Ese valor se guarda en una tabla. Más tarde, cuando se usa el método contains() para averiguar si un objeto está o no, hay que saber si está en esa tabla. Para un objeto determinado, el hash siempre va a tener el mismo valor.

La ventaja antes mencionada de Java en los Sets es relevante para los HashSets, ya que dos de estos métodos son el de saber si es igual a otro, con el método equals() y devolver

un número entero de modo tal que si dos objetos son iguales ese número también lo es (hash), mediante el método hashCode().

Implementa la interfaz Set.

LinkedHashSet

Implementa la interfaz Set.

TreeSet

Construye un árbol (forma de tener un conjunto de cosas todo el tiempo en orden, y permitir que se agreguen más cosas manteniendo el orden) con los objetos que se van agregando al conjunto. Una ventaja es que el orden de aparición de los elementos al recorrerlos es el orden natural de ellos. Esto tiene como desventaja que el mantenimiento del orden es costoso, haciendo que esta clase sea un poquito menos eficiente que HashSet.

Implementa la interfaz SortedSet.

Vector

Colección que ofrece un servicio similar a un Array, ya que puede almacenar y accesar valores y referencias a través de un índice. La diferencia es que, mientras un Array tiene un tamaño dado, un Vector puede crecer y decrecer dinámicamente conforme se vaya necesitando. Además, a diferencia de un Array, un Vector no está declarado para ser de un tipo en particular. En un Vector, un elemento puede insertarse y eliminarse de una posición específica a través de la invocación de un sólo método. Un objeto de tipo Vector maneja una lista de referencias a la clase Object. Sus métodos son:

- *Vector()* - Constructor, crea un vector inicialmente vacío.
- *void addElement(Object obj)* - Inserta el objeto especificado al final del vector.
- *void setElementAt(Object obj, int indice)* - Inserta el objeto especificado en el vector en la posición especificada.
- *Object remove(int indice)* - Elimina el objeto que se encuentra en la posición especificada y lo regresa.
- *boolean removeElement(Object obj)* - Elimina la primera ocurrencia del objeto especificado en el vector.
- *void removeElementAt(int indice)* - Elimina el objeto especificado en el índice del vector.
- *void clear()* - Elimina todos los objetos del vector.
- *boolean contains(Object obj)* - Retorna true si el objeto dado pertenece al vector.
- *int indexOf(Object obj)* - Retorna el índice del objeto especificado. Regresa -1 si no fue encontrado el objeto.
- *Object elementAt(int indice)* - Retorna el componente en el índice especificado.
- *boolean isEmpty()* - Retorna verdadero si el vector no contiene elementos.
- *int size()* - Retorna el número de elementos en el vector.

Implementa la interfaz List.

Array

Simple secuencia lineal que hace que el acceso a los elementos sea muy rápido, a costa de que, al momento de su creación, su tamaño es fijado y no se puede cambiar a lo largo de la vida del objeto. Una posible solución a este problema es crear un Array de tamaño determinado y luego, ya en tiempo de ejecución, crear otro más grande, mover todos los objetos al nuevo, y borrar el antiguo. Esto hace que Vector sea menos eficiente que un Array en cuestiones de velocidad. Es la forma más eficiente que Java proporciona para al-

macenar y acceder a una secuencia de objetos.

Metodologías de diseño Orientada a Objetos

Diseño orientado a objetos

Método de diseño que abarca el proceso de descomposición orientada a objetos y una notación para describir los modelos lógico y físico, así como los modelos estático y dinámico, del sistema que se diseña.

Análisis orientado a objetos

Método que enfatiza la construcción de modelos del mundo real, utilizando una visión orientada a objetos. Examina los requisitos desde la perspectiva de las clases y objetos que se encuentran en el vocabulario del dominio del problema.

Estilo de programación

Forma de organizar programas sobre las bases de algún modelo conceptual de programación y un lenguaje apropiado para que resulten claros los programas escritos en ese estilo.

Patrones

Soluciones recurrentes a problemas de diseño que se presentan comúnmente. Maneras convenientes de reutilizar código orientado a objetos entre proyectos y programadores. Se trata de anotar y catalogar aquellas interacciones comunes entre objetos que se encuentran frecuentemente y son útiles. Los patrones tratan problemas de diseño recurrente que aparecen en situaciones específicas del diseño y presentan soluciones al mismo.

Los buenos patrones deben solucionar un problema reiterativo, ser un concepto probado, describir participantes y sus relaciones, y no ser una solución obvia. Los patrones indican repetición, si algo no se repite, no es posible que sea un patrón. Un patrón se adapta para poder usarlo y es útil.

- *Repetición:* Característica cuantitativa pura. Es posible probarla demostrando la cantidad de veces que fue usada.
- *Adaptabilidad:* Característica cualitativa. (Cómo es exitoso el patrón)
- *Utilidad:* Característica cualitativa. (El patrón es exitoso y beneficioso)

Los patrones de diseño traen como ventajas la experiencia, el cómo sus nombres forman un vocabulario que ayuda a los desarrolladores a comunicarse mejor, la simplificación de lectura y comprensión de la documentación (en caso de que ésta use patrones) y, si están bien aplicados, hacen los diseños orientados a objetos más flexibles, elegantes y reutilizables.

La contracara del uso de patrones es que no garantizan nada por sí solos (ya que no buscan suplantar al humano en el proceso creativo) y que necesitamos seguir usando la creatividad para aplicarlos correctamente.

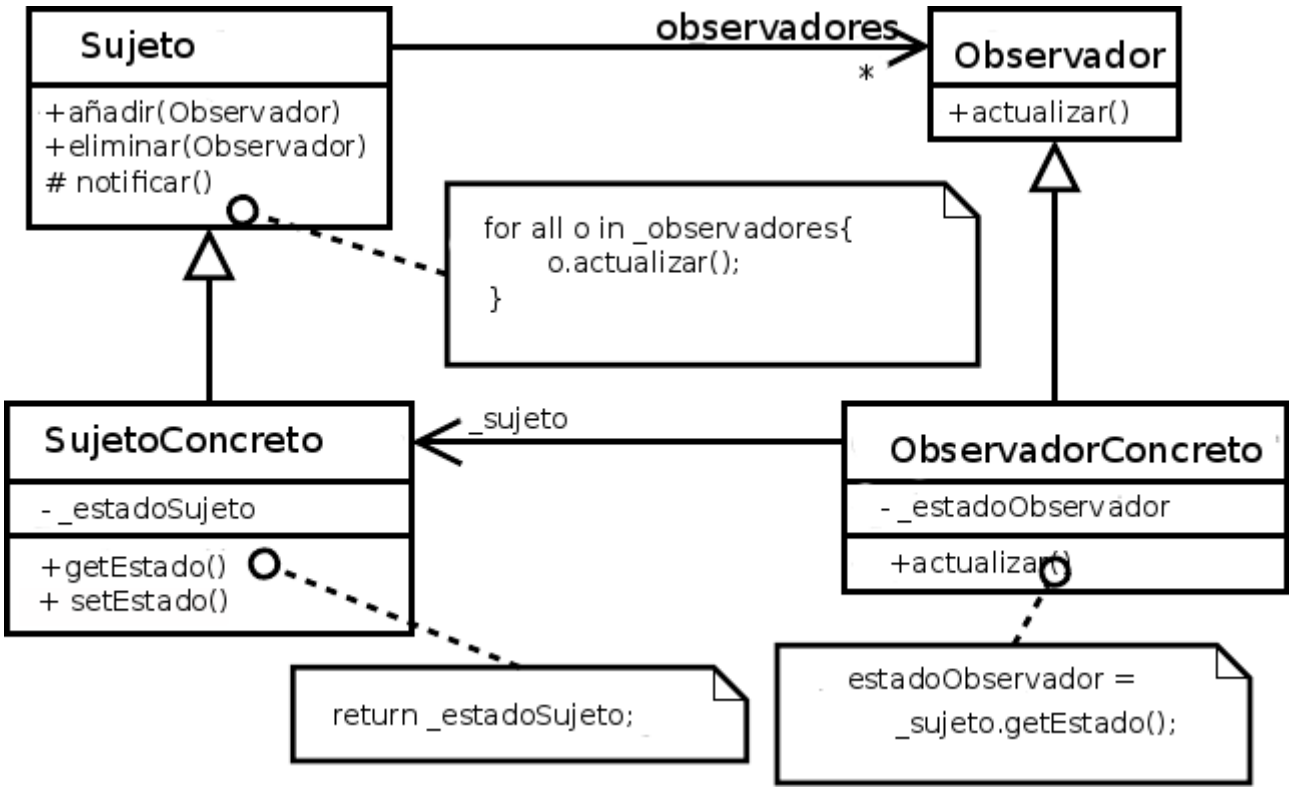
Los patrones ayudan a solucionar problemas de diseño encontrando objetos apropiados, determinando el número y tamaño de los objetos, especificando el protocolo de los objetos y motivando el diseño pensando en las interfaces y no en la implementación.

La descripción de un patrón está compuesta por su nombre, su clasificación, su intención (el objetivo del patrón explicado en términos abstractos), su nombre alternativo, su motivación (contar un problema concreto, la solución y luego la generalización), su aplicabilidad (cuando se debe aplicar el patrón), su estructura (esquema de objetos), sus participantes, sus colaboraciones, sus consecuencias (efectos adversos sobre el modelo OO), su implementación (un ejemplo de la solución), su código de ejemplo (un ejemplo en pseudocódigo), sus usos conocidos (ejemplos de aplicabilidad) y sus patrones relacionados.

Los patrones son soluciones concretas y técnicas, se aplican en situaciones muy comunes, son soluciones simples y facilitan la reutilización del código y el diseño. Como contracara, su uso no se refleja claramente en el código, y es difícil reutilizar su implementación.

Los patrones pueden pertenecer a una de tres categorías: creación, estructurales y de comportamiento. Estas clasificaciones, además, pueden subdividirse en patrones de diseño de clase y de objeto.

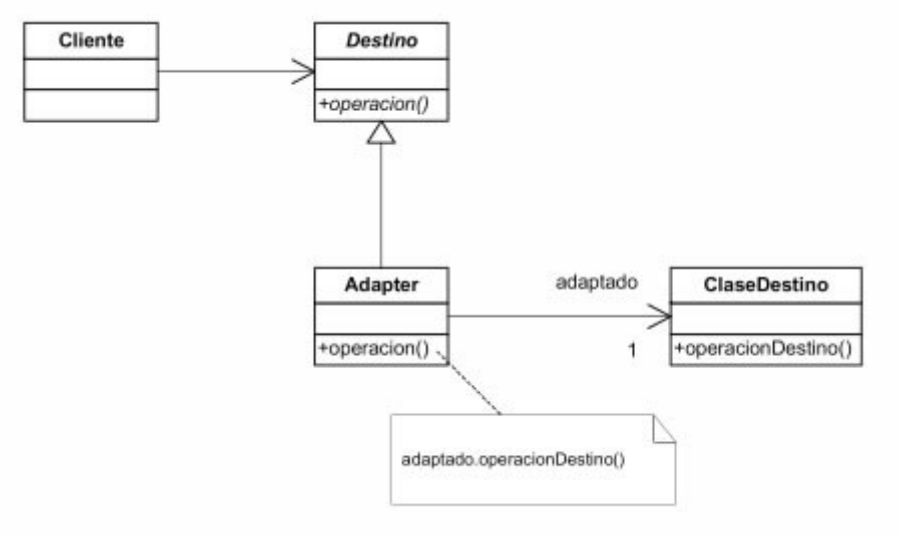
Observer



Clasificación	Comportamiento de Objeto
Intención	Define una dependencia uno a muchos entre objetos de modo que, cuando un objeto cambia su estado, todos sus dependientes son notificados y actualizados automáticamente.
Nombre alternativo	Dependants Publish-Subscribe
Motivación	Un efecto secundario común al particionar

	el sistema en una colección de clases que cooperan para realizar cierta tarea es la necesidad de mantener la consistencia entre estos objetos relacionados. Se quiere lograr consistencias, pero no a costa de atentar contra la flexibilidad y el reuso.
Aplicabilidad	Cuando una abstracción tiene dos aspectos, uno dependiente del otro. Cuando el cambio en un objeto requiere la actualización de otros, sin saber cuantos son. Cuando un objeto debería ser capaz de notificar a otros objetos sobre su cambio, sin necesidad de conocer las características de esos objetos.
Participantes	Subject, Observer, ConcreteSubject, ConcreteObserver (Sujeto, Observador, Sujeto Concreto, Observador Concreto)
Colaboraciones	El Sujeto Concreto notifica a sus dependientes cada vez que ocurre un cambio. Luego de ser informado de la ocurrencia de un cambio, un Observador Concreto puede llegar a preguntar al Sujeto sobre su estado.
Consecuencias	Bajo acoplamiento entre Sujetos y Observadores, entonces, es posible hacerlos interactuar aunque pertenezcan a diferentes capas de un sistema (model-view). Se pueden reusar los Sujetos sin necesidad de reusar los Observadores. Se pueden agregar Observadores el tiempo real sin necesidad de modificar el Sujeto ni los otros Observadores. Dependiendo de la implementación, puede haber actualizaciones inesperadas.
Patrones relacionados	**Mediator:** Puede utilizarse para encapsular la lógica de actualización. **Singleton:** Puede utilizarse para manejar los Sujetos.

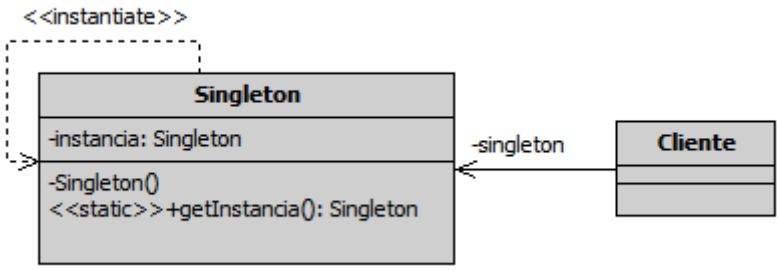
Adapter



Clasificación	Estructural de Clase (usa herencia para componer interfaces o implementaciones) Estructural de Objeto (compone objetos en tiempo de ejecución)
Intención	Convertir la interfaz de una clase en otra que esperan los clientes.
Nombre alternativo	Wrapper (envolvente) Class Adapter (para el EdC) Object Adapter (para el EdO)
Motivación	Reutilizar una clase de una biblioteca aunque su interfaz no correspondiera exactamente con el que requiere una aplicación concreta. Añadir funcionalidad que la clase reutilizada no proporciona.
Aplicabilidad	**Ambos:** Usar una clase existente cuya interfaz no se corresponde con el que se necesita. Crear una clase reutilizable que coopera con clases imprevistas (que no tienen necesariamente interfaces compatibles). **Object Adapter:** utilizar varias subclases existentes para las que sería poco práctico adaptar su interfaz heredando de cada una. Un adaptador de objeto puede adoptar la interfaz de su clase padre.
Participantes	Client, Target, Adapter, Adaptee (Cliente, Objetivo, Adaptador, Adaptado)
Colaboraciones	Los clientes llaman a las operaciones de un objeto Adaptador. A su vez, el Adaptador llama a las operaciones heredadas de la clase Adaptada que tratan la petición.

Consecuencias	**Class Adapter:** Adapta una clase Adaptada a una interfaz Objetivo reutilizando los métodos de la clase Adaptada. No funcionará cuando se quieran adaptar la clase adaptada y todas sus subclases. Además, la clase Adaptadora puede redefinir algunos de los métodos de la clase Adaptada. Por último, sólo se introduce un objeto, y no hace falta delegar en otro adaptado. **Object Adapter:** Permite trabajar un sólo adaptador con muchos adaptados (de la clase adaptada y sus subclases). Se puede añadir funcionalidad a todos los adaptados de una vez. Es más difícil si se necesita redefinir el comportamiento del adaptado. **Ambos:** Es posible que se requiera desde cambiar el nombre de los métodos hasta añadir nuevas operaciones para la adaptación que hace el adaptador.
Patrones relacionados	

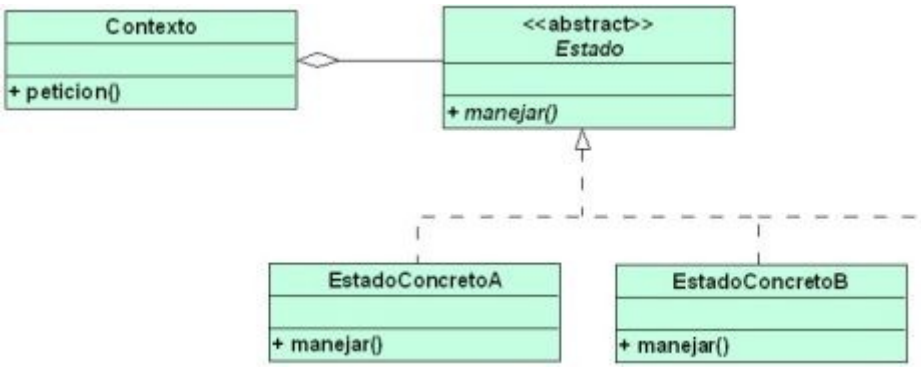
Singleton



Clasificación	Creación de Objeto
Intención	Asegurar que una clase sólo tiene un ejemplar y proporcionar un punto de acceso global a éste.
Nombre alternativo	
Motivación	Algunas clases sólo necesitan un ejemplar. En lugar de tener una variable global para acceder a ese ejemplar único, la clase se encarga de proporcionar un método de acceso.
Aplicabilidad	Cuando sólo puede haber un ejemplar de la clase, y debe ser accesible a los clientes desde un punto de acceso bien conocido. Cuando el único ejemplar pudiera ser extensible por herencia, y los clientes deberían usar el ejemplar de una subclase

	sin modificar su código.
Participantes	Clase Singleton.
Colaboraciones	Los clientes acceden al ejemplar de Singleton únicamente a través del método Instance de la clase Singleton.
Consecuencias	Controla el acceso a un ejemplar único. Evita la necesidad de utilizar variables globales. Permite refinar las operaciones y la representación. Permite un número de ejemplares variable. Es más flexible que las operaciones de clase. (static)
Patrones relacionados	**Abstract Factory:** muchas veces implementados a partir de Singleton, ya que deben ser accesibles públicamente y debe haber una única instancia que controle la creación de objetos. **Monostate:** Similar a Singleton, pero en lugar de controlar el instanciado de una clase, asegura que todas las instancias tengan un estado en común, haciendo que todos sus miembros sean de esa clase.

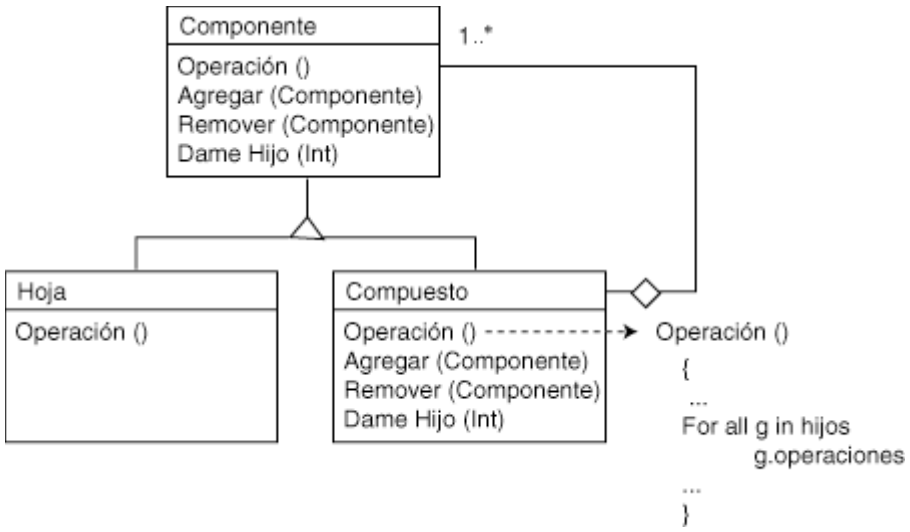
State



Clasificación	Comportamiento de Objeto
Intención	
Nombre alternativo	
Motivación	Cambiar el comportamiento dependiendo del estado. Evitar la complejidad de código derivada de la creación de objetos que cambian su comportamiento según cambia su estado.
Aplicabilidad	Se utiliza cuando el objeto tiene diferentes estados y cambia su comportamiento para cada estado. También se utiliza como máquina de

	estados. Si las operaciones tienen largas sentencias con múltiples ramas que dependen del estado del objeto.
Participantes	Context, State, ConcreteState<1..X> (Contexto, Estado, Estado Concreto <1..X>)
Colaboraciones	
Consecuencias	Localiza el comportamiento dependiente del estado y divide dicho comportamiento en diferentes estados. Esto ayuda a la eliminación de bloques if...then...else y case...switch, ya que las transiciones entre estados se reparte entre las subclases. Hace explícitas las transiciones entre estados, introduciendo objetos separados para los diferentes estados, haciéndolas más explícitas. Permite que los Objetos Estado puedan compartirse.
Patrones relacionados	

Composite



Clasificación	Estructural de Objeto
Intención	Construir objetos complejos mediante la composición recursiva de objetos similares de manera similar a un árbol.
Nombre alternativo	
Motivación	Para crear componentes que pueden agruparse para formar componentes mayores (contenedores)
Aplicabilidad	Representar jerarquías de objetos parte-todo. Para que los clientes puedan manejar indistintamente objetos individuales o

	composiciones de objetos.
Participantes	Client, Component, Leaf, Composite (Cliente, Componente, Hoja, Compuesto)
Colaboraciones	Los clientes usan la clase Componente para interactuar con los objetos de la estructura Compuesta. Si el recipiente es una Hoja, la petición se maneja directamente. Si se trata de un Compuesto, se pasa a sus componentes hijos, pudiéndose realizar operaciones adicionales antes o después.
Consecuencias	Define jerarquías de clases que tienen objetos primitivos y compuestos (la composición puede ser recursiva). Hace el cliente simple tratando la estructura y los objetos individuales uniformemente. Facilita la adición de nuevas clases de componentes. Puede hacer que el diseño sea demasiado general, haciendo más difícil restringir los componentes de un compuesto y requiriendo la codificación de las comprobaciones para que se realicen en tiempo de ejecución, en caso de que se quiera hacer que un compuesto sólo tenga ciertos componentes.
Patrones relacionados	