PATRONES ORIENTADOS AL COMPORTAMIENTO

MANUEL ALEJANDRO DOMINGUEZ

ALEJANDRO GUTIERREZ RESTREPO

CRISTIAN YESID PALACIOS VALENCIA

PAULO ANDRES ARIAS ESGUERRA

INGENIERIA DE SISTEMAS

INSTITUTO TECNOLOGICO METROPOLITANO

2019-1

MEDELLIN-ANTIOQUIA

**PATRONES ORIENTADOS AL COMPORTAMIENTO**

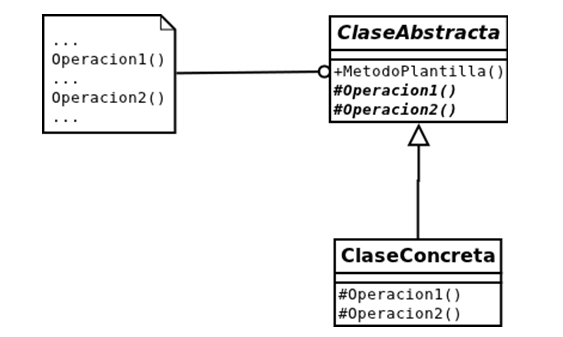
**PATRON MÉTODO PLANTILLA**

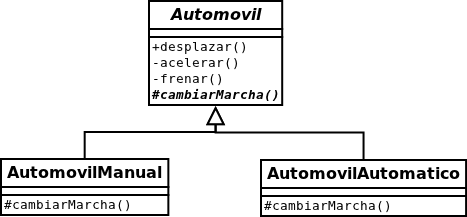
En [ingeniería de software](https://es.wikipedia.org/wiki/Ingenier%C3%ADa_de_software), el patrón de método de la plantilla es un [patrón de diseño](https://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o) de comportamiento que define el esqueleto de programa de un [algoritmo](https://es.wikipedia.org/wiki/Algoritmo) en un método, llamado método de plantilla, el cual difiere algunos pasos a las [subclases.](https://es.wikipedia.org/wiki/Herencia_(inform%C3%A1tica))[1](https://es.wikipedia.org/wiki/Patr%C3%B3n_de_m%C3%A9todo_de_la_plantilla#cite_note-1)​ Permite redefinir ciertos pasos seguros de un algoritmo sin cambiar la estructura del algoritmo.

**Uso**

El método de plantilla está diseñado para marcos, donde cada cual implementa las partes invariables de la arquitectura de un ámbito, dejando "placeholders" para personalizar las opciones. Esto es un ejemplo de [inversión de control](https://es.wikipedia.org/wiki/Inversi%C3%B3n_de_control). Algunas razones por la que se utiliza el método de plantilla son:

* Dejar que las subclases que se implementan (a través del método primordial) tengan un comportamiento que puede variar.
* Evitar duplicación en el código: la estructura general de flujo de trabajo, está implementada una vez en el [algoritmo](https://es.wikipedia.org/wiki/Algoritmo) de clase abstracta, y variaciones necesarias son implementadas en cada de las subclases.
* Control en qué punto(s) la [subclassing](https://es.wikipedia.org/wiki/Herencia_(inform%C3%A1tica)" \o "Herencia (informática)) está permitida. En oposición a una sencilla sobrecarga polimórfica, donde el método de base sería enteramente reescrito, permitiendo un cambio radical en el flujo. Sólo los detalles específicos del flujo se pueden cambiar.





**PATRON ITERADOR**

En [diseño de software](https://es.wikipedia.org/wiki/Dise%C3%B1o_de_software), el [patrón de diseño](https://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o) Iterador, define una interfaz que declara los métodos necesarios para acceder secuencialmente a un grupo de objetos de una colección. Algunos de los métodos que podemos definir en la interfaz Iterador son:

Primero(), Siguiente(), HayMas() y ElementoActual().

Este patrón de diseño permite recorrer una estructura de datos sin que sea necesario conocer la estructura interna de la misma.

**Propósito**

El patrón Iterador es un mecanismo de acceso a los elementos que constituyen una estructura de datos para la utilización de estos sin exponer su estructura interna.

**Motivación**

El patrón surge del deseo de acceder a los elementos de un contenedor de objetos (por ejemplo, una lista) sin exponer su representación interna. Además, es posible que se necesite más de una forma de recorrer la estructura siendo para ello necesario crear modificaciones en la clase.

La solución que propone el patrón es añadir métodos que permitan recorrer la estructura sin referenciar explícitamente su representación. La responsabilidad del recorrido se traslada a un objeto iterador.

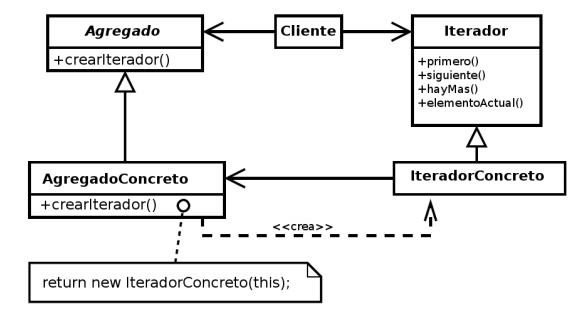
El problema de introducir este objeto iterador reside en que los clientes necesitan conocer la estructura para crear el iterador apropiado.

Esto se soluciona generalizando los distintos iteradores en una abstracción y dotando a las estructuras de datos de un [método de fabricación](https://es.wikipedia.org/wiki/Factory_Method_(patr%C3%B3n_de_dise%C3%B1o)) que cree un iterador concreto.

**Aplicabilidad**

El patrón iterator permite el acceso al contenido de una estructura sin exponer su representación interna. Además diferentes iteradores pueden presentar diferentes tipos de recorrido sobre la estructura (recorrido de principio a fin, recorrido con saltos...). Por otro lado los iteradores no tienen por qué limitarse a recorrer la estructura, sino que podrían incorporar otro tipo de lógica (por ejemplo, filtrado de elementos). Es más, dados diferentes tipos de estructuras, el patrón iterador permite recorrerlas todas utilizando una interfaz común uniforme.

**Estructura**

****

**Participantes**

Las entidades participantes en el diseño propuesto por el patrón iterador son:

* **Iterador (Iterator)** define la interfaz para recorrer el agregado de elementos y acceder a ellos, de manera que el cliente no tenga que conocer los detalles y sea capaz de manejarlos de todos modos.
* **Iterador Concreto (ConcreteIterator)** implementa la interfaz propuesta por el Iterador. Es el que se encarga de mantener la posición actual en el recorrido de la estructura.
* **Agregado (Aggregate)** define la interfaz para el método de fabricación de iteradores.
* **Agregado Concreto (ConcreteAggregate)** implementa la estructura de datos y el método de fabricación de iteradores que crea un iterador específico para su estructura.

**Implementación**

Para la creación del patrón iterador debe implementarse el control de la iteración (pudiendo ser un iterador externo que ofrece los métodos para que el cliente recorra la estructura paso a paso, o un iterador interno que ofrece un método de actuación sobre la estructura que, de manera transparente al cliente, la recorre aplicándose a todos sus elementos) y definirse el recorrido. A mayores se podrían implementar operaciones adicionales en el iterador o definir la estructura de éste de una manera más robusta ante cambios en la estructura. Hay que tener especial cuidado en la implementación de iteradores con accesos privilegiados, iteradores para estructuras compuestas o iteradores nulos.

**PATRON COMANDO**

Este patrón permite solicitar una operación a un objeto sin conocer realmente el contenido de esta operación, ni el receptor real de la misma. Para ello se encapsula la petición como un objeto, con lo que además facilita la parametrización de los métodos.

**Propósito**

* Encapsula un mensaje como un objeto, con lo que permite gestionar colas o registro de mensaje y deshacer operaciones.
* Soportar restaurar el estado a partir de un momento dado.
* Ofrecer una interfaz común que permita invocar las acciones de forma uniforme y extender el sistema con nuevas acciones de forma más sencilla.

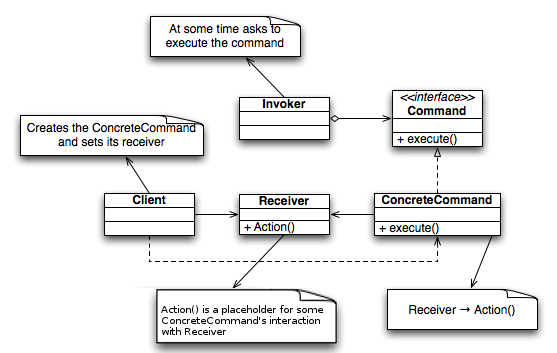
**Motivo**

* El concepto de "orden" puede ser ambiguo y complejo en los sistemas actuales y al mismo tiempo muy extendido: intérpretes de órdenes del sistema operativo, lenguajes de macros de paquetes ofimáticos, gestores de bases de datos, protocolos de servidores de Internet, etc.
* Este patrón presenta una forma sencilla y versátil de implementar un sistema basado en comandos facilitándose su uso y ampliación.

**Aplicaciones**

* Facilitar la parametrización de las acciones a realizar.
* Independizar el momento de petición del de ejecución.
* Implementar CallBacks, especificando que órdenes queremos que se ejecuten en ciertas situaciones de otras órdenes. Es decir, un parámetro de una orden puede ser otra orden a ejecutar.
* Soportar el "deshacer".
* Desarrollar sistemas utilizando órdenes de alto nivel que se construyen con operaciones sencillas (primitivas).

**Estructura**

****

**Participantes**

* AbstractCommand.

Clase que ofrece una interfaz para la ejecución de órdenes. Define los métodos do y undo que se implementarán en cada clase concreta.

* ConcreteCommand.

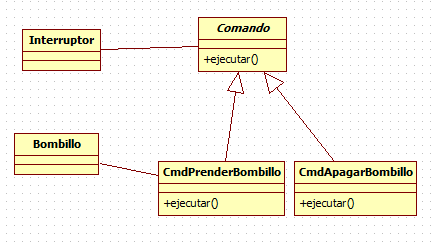
Clase que implementa una orden concreta y sus métodos do y undo. Su constructor debe inicializar los parámetros de la orden.

* Invoker.

Clase que instancia las órdenes, puede a su vez ejecutarlas inmediatamente (llamando a do) o dejar que el CommandManager lo haga.

* CommandManager.

Responsable de gestionar una colección de objetos orden creadas por el Invoker. Llamará a los métodos do y unDo. Gestionará su secuenciación y reordenación (sobre la base de prioridades por ejemplo).



**PATRON OBSERVADOR**

Es un [patrón de diseño](https://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o) de [software](https://es.wikipedia.org/wiki/Software) que define una dependencia del tipo uno a muchos entre objetos, de manera que cuando uno de los objetos cambia su estado, notifica este cambio a todos los dependientes. Se trata de un patrón de comportamiento, por lo que está relacionado con algoritmos de funcionamiento y asignación de responsabilidades a [clases](https://es.wikipedia.org/wiki/Clase_(inform%C3%A1tica)) y [objetos](https://es.wikipedia.org/wiki/Objeto_(programaci%C3%B3n)).

Los patrones de comportamiento describen no solamente estructuras de relación entre objetos o clases sino también esquemas de comunicación entre ellos y se pueden clasificar en función de que trabajen con clases u objetos.

La variación de la encapsulación es la base de muchos patrones de comportamiento, por lo que cuando un aspecto de un programa cambia frecuentemente, estos patrones definen un objeto que encapsula dicho aspecto. Los patrones definen una clase abstracta que describe la encapsulación del objeto.

Los observadores a su vez están obligados a implementar unos métodos determinados mediante los cuales el Sujeto es capaz de notificar a sus observadores suscritos los cambios que sufre para que todos ellos tengan la oportunidad de refrescar el contenido representado. De manera que cuando se produce un cambio en el Sujeto, ejecutado, por ejemplo, por alguno de los observadores, el objeto de datos puede recorrer la lista de observadores avisando a cada uno. Este patrón suele utilizarse en los [entornos de trabajo](https://es.wikipedia.org/wiki/Framework) de interfaces gráficas orientados a objetos, en los que la forma de capturar los eventos es suscribir listeners a los objetos que pueden disparar eventos.

El patrón observador es la clave del patrón de arquitectura [Modelo Vista Controlador](https://es.wikipedia.org/wiki/Modelo_Vista_Controlador) (MVC).[1](https://es.wikipedia.org/wiki/Observer_(patr%C3%B3n_de_dise%C3%B1o)#cite_note-jont-1)​ De hecho el patrón fue implementado por primera vez en el MVC de [Smalltalk](https://es.wikipedia.org/wiki/Smalltalk" \o "Smalltalk) basado en un entorno de trabajo de interfaz.[2](https://es.wikipedia.org/wiki/Observer_(patr%C3%B3n_de_dise%C3%B1o)#cite_note-gang-2)​ Este patrón está implementado en numerosos [bibliotecas](https://es.wikipedia.org/wiki/Biblioteca_(inform%C3%A1tica)) y sistemas, incluyendo todos los toolkits de [GUI](https://es.wikipedia.org/wiki/GUI).

**Objetivo**

Definir una dependencia uno a muchos entre objetos, de tal forma que cuando el objeto cambie de estado, todos sus objetos dependientes sean notificados automáticamente. Se trata de desacoplar la clase de los objetos clientes del objeto, aumentando la modularidad del lenguaje, creando las mínimas dependencias y evitando bucles de actualización ([espera activa](https://es.wikipedia.org/w/index.php?title=Espera_espera_activa&action=edit&redlink=1) o [sondeo](https://es.wikipedia.org/wiki/Polling)). En definitiva, normalmente, se usará el patrón observador cuando un elemento quiere estar pendiente de otro, sin tener que estar comprobando de forma continua si ha cambiado o no.

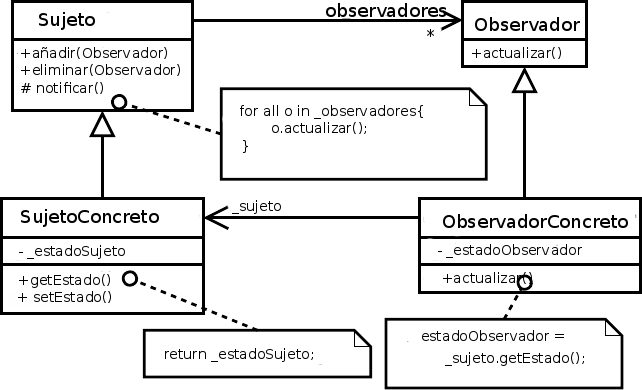
**Motivación**

Si se necesita consistencia entre clases relacionadas, pero con independencia, es decir, con un bajo [acoplamiento](https://es.wikipedia.org/wiki/Acoplamiento_inform%C3%A1tico).

**Ámbito De Aplicación**

Puede pensarse en aplicar este patrón cuando una modificación en el estado de un objeto requiere cambios de otros, y no se desea que se conozca el número de objetos que deben ser cambiados. También cuando se quiere que un objeto sea capaz de notificar a otros objetos sin hacer ninguna suposición acerca de los objetos notificados y cuando una abstracción tiene dos aspectos diferentes, que dependen uno del otro; si se encapsulan estos aspectos en objetos separados se permitirá su variación y reutilización de modo independiente.

**Estructura**

****

**Participantes**

Habrá sujetos concretos cuyos cambios pueden resultar interesantes a otros y observadores a los que al menos les interesa estar pendientes de un elemento y en un momento dado, reaccionar ante sus notificaciones de cambio. Todos los sujetos tienen en común que un conjunto de objetos quieren estar pendientes de ellos. Cualquier elemento que quiera ser observado tiene que permitir indicar:

1. “Estoy interesado en tus cambios”.
2. “Ya no estoy interesado en tus cambios”.

El observable tiene que tener, además, un mecanismo de aviso a los interesados.

A continuación se detallan a los participantes de forma desglosada:

* Sujeto (subject):

El sujeto proporciona una interfaz para agregar (attach) y eliminar (detach) observadores. El Sujeto conoce a todos sus observadores.

* Observador (observer):

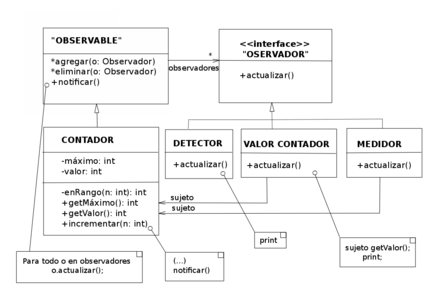
Define el método que usa el sujeto para notificar cambios en su estado (update/notify).

* Sujeto concreto (concrete subject):

Mantiene el estado de interés para los observadores concretos y los notifica cuando cambia su estado. No tienen por qué ser elementos de la misma jerarquía.

* Observador concreto (concrete observer):

Mantiene una referencia al sujeto concreto e implementa la interfaz de actualización, es decir, guardan la referencia del objeto que observan, así en caso de ser notificados de algún cambio, pueden preguntar sobre este cambio.



**PATRON MEDIADOR**

El patrón mediador define un objeto que encapsula cómo un conjunto de objetos interactúan. Este [patrón de diseño](https://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o) está considerado como un patrón de comportamiento debido al hecho de que puede alterar el comportamiento del programa en ejecución.

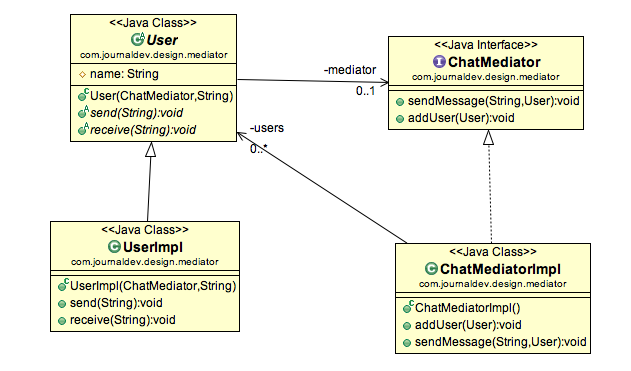
Habitualmente un programa está compuesto de un número de [clases](https://es.wikipedia.org/wiki/Clase_(inform%C3%A1tica)) (muchas veces elevado). La [lógica](https://es.wikipedia.org/wiki/L%C3%B3gica) y [computación](https://es.wikipedia.org/wiki/Computaci%C3%B3n) es distribuida entre esas clases. Sin embargo, cuantas más clases son desarrolladas en un programa, especialmente durante [mantenimiento](https://es.wikipedia.org/wiki/Mantenimiento_de_software) y/o [refactorización](https://es.wikipedia.org/wiki/Refactorizaci%C3%B3n), el problema de [comunicación](https://es.wikipedia.org/wiki/Comunicaci%C3%B3n) entre estas clases quizás llegue a ser más complejo. Esto hace que el programa sea más difícil de leer y [mantener](https://es.wikipedia.org/wiki/Mantenimiento_de_software). Además, puede llegar a ser difícil cambiar el programa, ya que cualquier cambio podría afectar código en muchas otras clases.

Con el patrón mediador, la comunicación entre objetos es encapsulada con un objeto mediador. Los objetos no se comunican de forma directa entre ellos, en lugar de ello se comunican mediante el mediador. Esto reduce las dependencias entre los objetos en comunicación, reduciendo entonces la [Dependencia de código](https://es.wikipedia.org/w/index.php?title=Dependencia_de_c%C3%B3digo&action=edit&redlink=1).

La esencia del Patrón Mediador es "Definir un objeto que encapsula cómo un conjunto de objetos interactúan. El mediador busca reducir la [dependencia](https://es.wikipedia.org/w/index.php?title=Dependencia_de_c%C3%B3digo&action=edit&redlink=1) evitando que los objetos se relacionen entre ellos de forma explícita, y permitiendo variar cualquier interacción independientemente".

**Participantes**

* Mediador - define la interfaz para la comunicación entre objetos amigos
* MediadorConcreto - implementa la interfaz Mediador y coordina la comunicación entre objetos amigos. Es consciente de todos los amigos y su propósito en lo que concierne a la comunicación entre ellos.
* AmigoConcreto - se comunica con otros amigos a través de su Mediador

****

**PATRON ESTADO**

El [patrón de diseño](https://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o) estado se utiliza cuando el comportamiento de un objeto cambia dependiendo del estado del mismo. Por ejemplo: una alarma puede tener diferentes estados, como desactivada, activada, en configuración. Definimos una interfaz Estado\_Alarma, y luego definimos los diferentes estados.

En determinadas ocasiones, cuando el contexto en el que se está desarrollando requiere que un objeto tenga diferentes comportamientos según el estado en que se encuentra, resulta complicado poder manejar el cambio de comportamientos y los estados de dicho objeto, todos dentro del mismo [bloque de código](https://es.wikipedia.org/wiki/Bloque_de_c%C3%B3digo). El patrón estado propone una solución a esta complicación, creando básicamente, un objeto por cada estado posible del objeto que lo llama.

**Propósito**

Permite a un objeto alterar su comportamiento dependiendo de su estado interno.

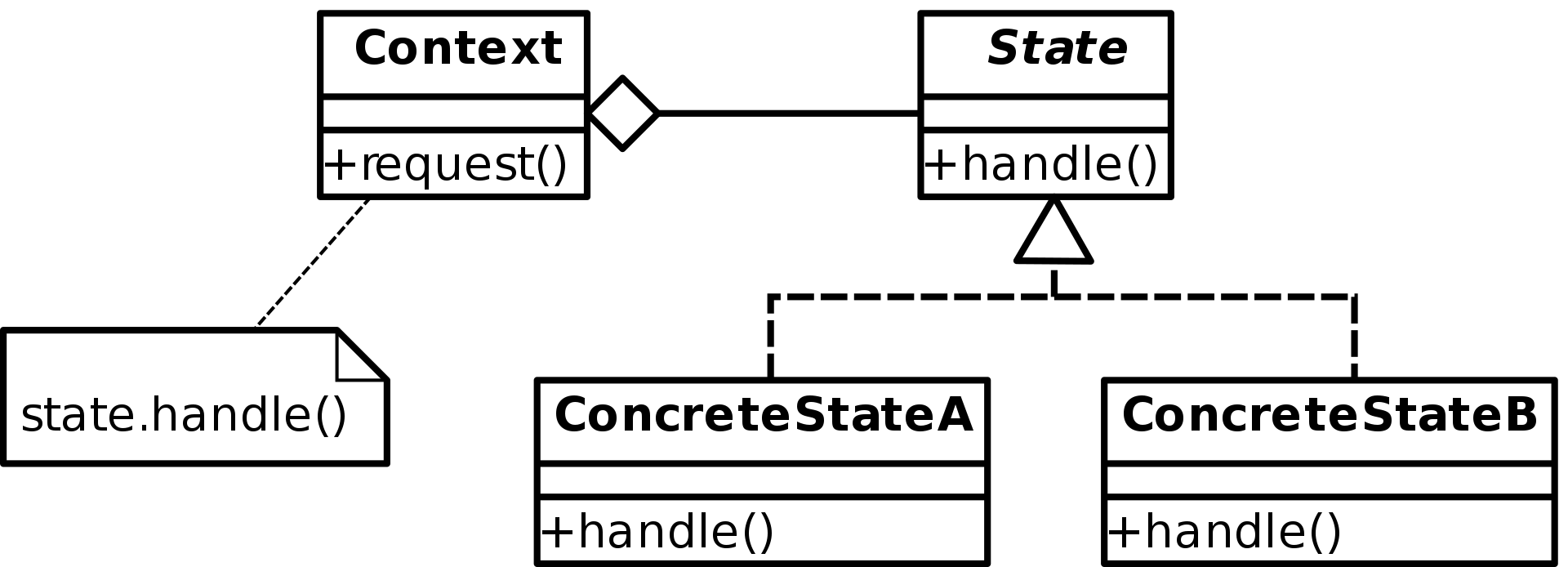
**Motivación**

El patrón State está motivado por aquellos objetos en que, según su estado actual, varía su comportamiento ante los diferentes mensajes. Como ejemplo se toma una clase TCPConnection que representa una conexión de red, un objeto de esta clase tendrá diferentes respuestas según su estado (Listening, Closed o Established). Por ejemplo la invocación al método Open de un objeto de la clase TCPConnection diferirá su comportamiento si la conexión se encuentra en Closed o en Established.

**Problema**

Existe una extrema complejidad en el código cuando se intenta administrar comportamientos diferentes según una cantidad de estados diferentes. Asimismo el mantenimiento de este código se torna dificultoso, e incluso se puede llegar en algunos casos puntuales a la incongruencia de estados actuales por la forma de implementación de los diferentes estados en el código (por ejemplo con variables para cada estado).

**Estructura**



**Participantes**

1. **Context(Contexto):** Este integrante define la interfaz con el cliente. Mantiene una instancia de ConcreteState (Estado Concreto) que define su estado actual
2. **State (Estado):** Define una interfaz para el encapsulamiento de las responsabilidades asociadas con un estado particular de Context.
3. **Subclase ConcreteState:** Cada una de estas subclases implementa el comportamiento o responsabilidad de Context.

El Contexto (Context) delega el estado específico al objeto ConcreteState actual Un objeto Context puede pasarse a sí mismo como parámetro hacia un objeto State. De esta manera la clase State puede acceder al contexto si fuese necesario. Context es la interfaz principal para el cliente. El cliente puede configurar un contexto con los objetos State. Una vez hecho esto, los clientes no tendrán que tratar con los objetos State directamente. Tanto el objeto Context como los objetos de ConcreteState pueden decidir el cambio de estado.

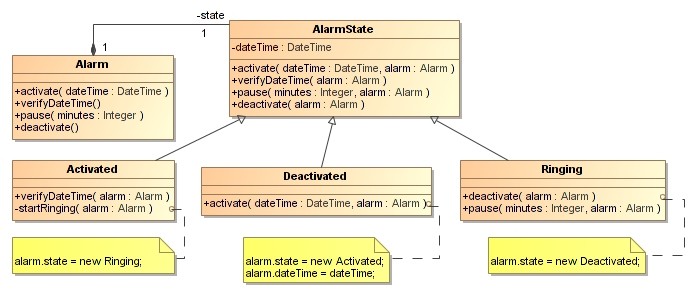
**Ventajas y Desventajas**

Se encuentran las siguientes ventajas:

* Se localizan fácilmente las responsabilidades de los estados específicos, dado que se encuentran en las clases que corresponden a cada estado. Esto brinda una mayor claridad en el desarrollo y el mantenimiento posterior. Esta facilidad la brinda el hecho que los diferentes estados están representados por un único atributo (state) y no envueltos en diferentes variables y grandes condicionales.
* Hace los cambios de estado explícitos puesto que en otros tipos de implementación los estados se cambian modificando valores en variables, mientras que aquí al estar representado cada estado.
* Los objetos State pueden ser compartidos si no contienen variables de instancia, esto se puede lograr si el estado que representan está enteramente codificado en su tipo. Cuando se hace esto estos estados son Flyweights sin estado intrínseco.
* Facilita la ampliación de estados
* Permite a un objeto cambiar de clase en tiempo de ejecución dado que al cambiar sus responsabilidades por las de otro objeto de otra clase la herencia y responsabilidades del primero han cambiado por las del segundo.

Se encuentran la siguiente desventaja:

* Se incrementa el número de subclases.

****

**PATRON ESTRATEGIA**

El patrón Estrategia es un [patrón de diseño](https://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o) para el desarrollo de software. Se clasifica como patrón de comportamiento porque determina cómo se debe realizar el intercambio de mensajes entre diferentes objetos para resolver una tarea. El patrón estrategia permite mantener un conjunto de algoritmos de entre los cuales el objeto cliente puede elegir aquel que le conviene e intercambiarlo dinámicamente según sus necesidades.

**Motivación**

Suponiendo un editor de textos con diferentes algoritmos para particionar un texto en líneas (justificado, alineado a la izquierda, etc.), se desea separar las clases clientes de los diferentes algoritmos de partición, por diversos motivos:

* Incluir el código de los algoritmos en los clientes hace que estos sean demasiado grandes y complicados de mantener y/o extender.
* El cliente no va a necesitar todos los algoritmos en todos los casos, de modo que no queremos que dicho cliente los almacene si no los va a usar.
* Si existiesen clientes distintos que usasen los mismos algoritmos, habría que duplicar el código, por tanto, esta situación no favorece la reutilización.

La solución que el patrón estrategia supone para este escenario pasa por encapsular los distintos algoritmos en una jerarquía y que el cliente trabaje contra un objeto intermediario contexto. El cliente puede elegir el algoritmo que prefiera de entre los disponibles, o el mismo contexto puede ser el que elija el más apropiado para cada situación.

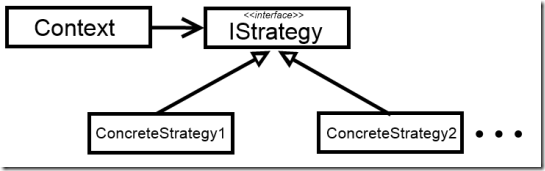
**Aplicabilidad**

Cualquier programa que ofrezca un servicio o función determinada, que pueda ser realizada de varias maneras, es candidato a utilizar el patrón estrategia. Puede haber cualquier número de estrategias y cualquiera de ellas podrá ser intercambiada por otra en cualquier momento, incluso en tiempo de ejecución. Si muchas clases relacionadas se diferencian únicamente por su comportamiento, se crea una superclase que almacene el comportamiento común y que hará de interfaz hacia las clases concretas.

Si un algoritmo utiliza información que no deberían conocer los clientes, la utilización del patrón estrategia evita la exposición de dichas estructuras. Aplicando el patrón a una clase que defina múltiples comportamientos mediante instrucciones condicionales, se evita emplear estas instrucciones, moviendo el código a clases independientes donde se almacenará cada estrategia.

Efectivamente, como se comenta anteriormente, este patrón de diseño nos sirve para intercambiar un sin número de estrategias posibles.

**Estructura**

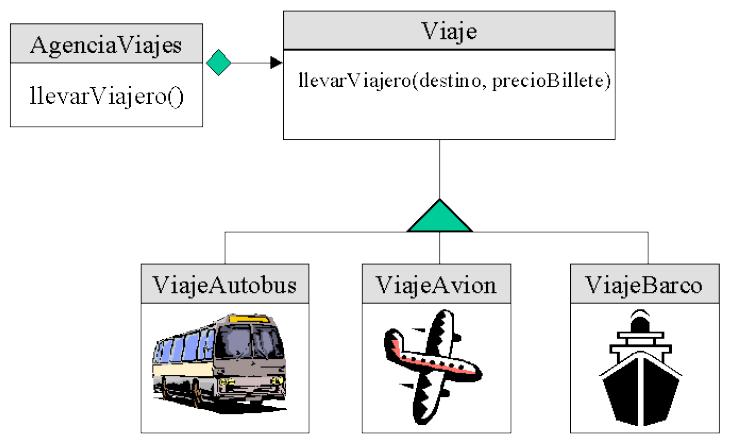
****

**Participantes**

**Contexto (Context):** Es el elemento que usa los algoritmos, por tanto, delega en la jerarquía de estrategias. Configura una estrategia concreta mediante una referencia a la estrategia necesaria. Puede definir una interfaz que permita a la estrategia el acceso a sus datos en caso de que fuese necesario el intercambio de información entre el contexto y la estrategia. En caso de no definir dicha interfaz, el contexto podría pasarse a sí mismo a la estrategia como parámetro.

**Estrategia (Strategy):** Declara una interfaz común para todos los algoritmos soportados. Esta interfaz será usada por el contexto para invocar a la estrategia concreta.

**EstrategiaConcreta (ConcreteStrategy):** Implementa el algoritmo utilizando la interfaz definida por la estrategia.

****