Patrones de comportamiento con Java, Parte II

ABRAHAM SÁNCHEZ LÓPEZ
GRUPO MOVIS
FCC-BUAP

El patrón Memento, I

Descripción

- El patrón Memento tiene como objetivo salvaguardar y restablecer el estado de un objeto sin violar la encapsulación.

Ejemplo

- Durante la compra online de un vehículo nuevo, el cliente puede seleccionar opciones suplementarias que se agregarán a su carrito de la compra. No obstante, existen opciones incompatibles como por ejemplo, asientos deportivos frente a asientos en cuero o reclinables.
- La consecuencia de esta incompatibilidad es que si se han seleccionado asientos reclinables y a continuación se eligen asientos en cuero, la opción de los asientos reclinables se elimina del carrito de compra.
- Queremos incluir una opción para poder anular la última operación realizada en el vehículo. Suprimir la última opción agregada no es suficiente puesto que es necesario también restablecer las opciones presentes y que se han eliminado debido a la incompatibilidad.

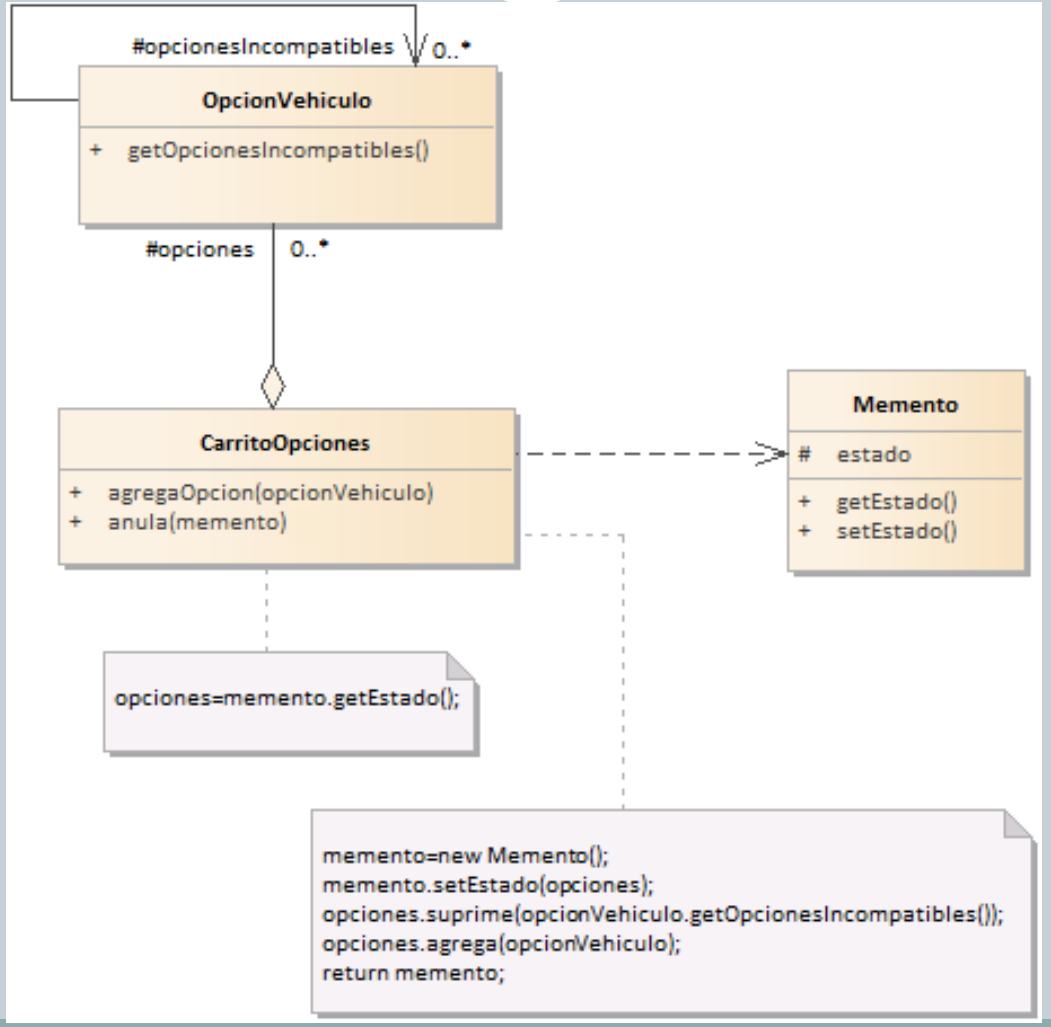
El patrón Memento, II

- Una solución consiste en memorizar el estado del carrito de la compra antes de agregar la nueva opción.
- Además, deseamos ampliar este comportamiento para gestionar un histórico de los estados del carrito de la compra y poder volver a cualquier estado anterior. Es preciso entonces, en este caso, memorizar todos los estados sucesivos del vehículo.
- Para preservar la encapsulación del objeto que representa el carrito de la compra, una solución consistiría en memorizar estos estados intermedios en el propio carrito. No obstante esta solución tendría como efecto un aumento inútil en la complejidad de este objeto.
- El patrón Memento proporciona una solución a este problema. Consiste en memorizar los estados del carrito de la compra en un objeto llamado memento (agenda o histórico).
- Cuando se agrega una nueva opción, el carrito crea un histórico, lo inicializa con su estado, retira las opciones incompatibles con la nueva opción, procede a agregar esta nueva opción y reenvía el memento así creado.

El patrón Memento, III

- Este se utilizará a continuación en caso de que se quiera anular la opción agregada y volver al estado anterior.
- Sólo el carrito de la compra puede memorizar su estado en el memento y restaurar un estado anterior: el memento es opaco de cara a los demás objetos.
- El diagrama de clases correspondiente se muestra en la figura del acetato 5.
- El carrito de la compra está representado por la clase CarritoOpciones y el memento por la clase Memento. El estado del carrito de la compra consiste en el conjunto de sus enlaces con las demás opciones.
- Las opciones están representadas mediante la clase OpcionVehiculo que incluye una asociación reflexiva para describir las opciones incompatibles.
- Conviene observar que las opciones forman un conjunto de instancias de la clase OpcionVehiculo.
- Estas instancias están compartidas entre todos los carritos de la compra.

El patrón Memento, IV



El patrón Memento, V

Estructura

1. Diagrama de clases

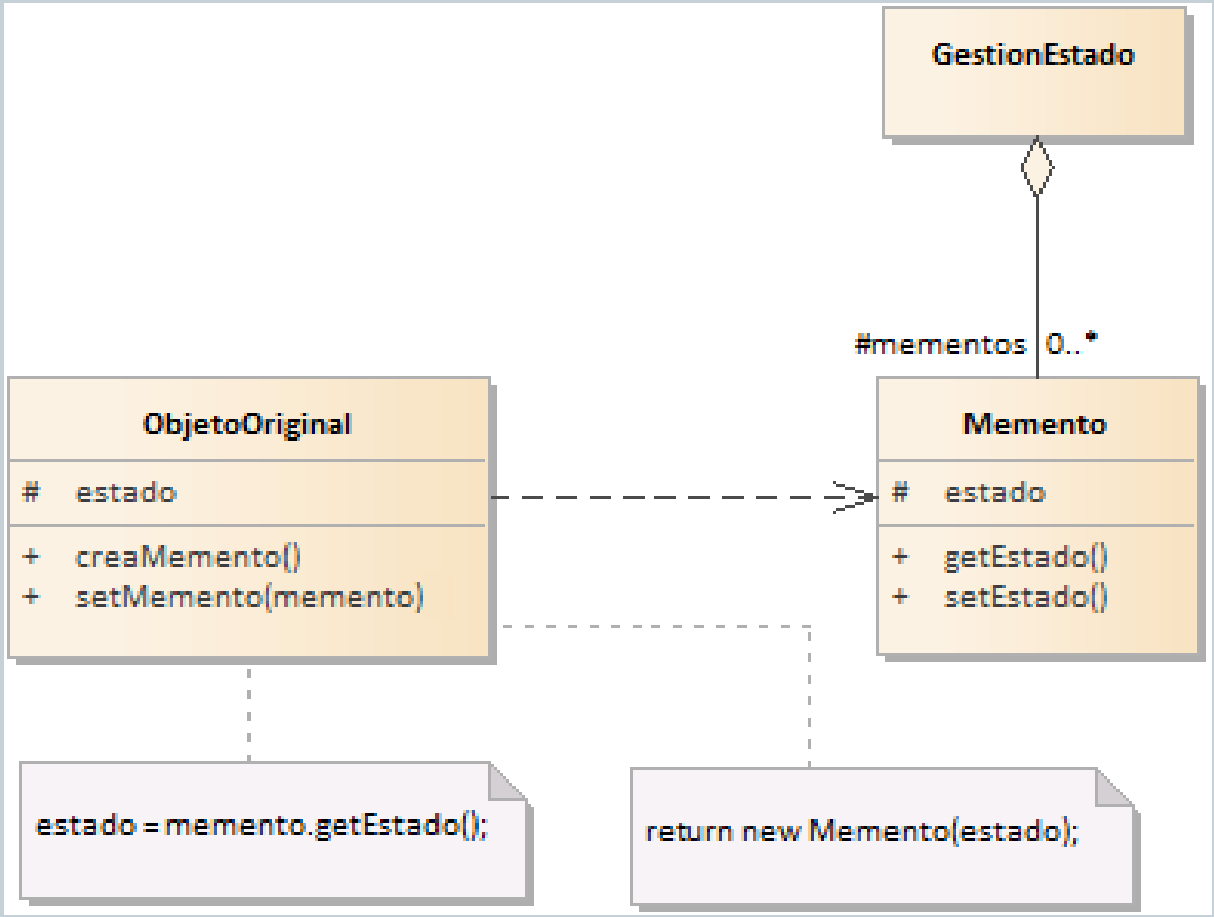
- La figura del acetato 7 detalla la estructura genérica del patrón.

2. Participantes

- Los participantes del patrón son los siguientes:
  - Memento es la clase de los mementos, que son los objetos que memorizan el estado interno de los objetos originales (o una parte de este estado).

    El memento posee dos interfaces: una interfaz completa destinada a los objetos originales que ofrece la posibilidad de memorizar y de restaurar su estado y una interfaz reducida para los objetos de gestión del estado que no tienen permisos para acceder al estado interno de los objetos originales.
  - ObjetoOriginal (CarritoOpciones) es la clase de los objetos que crean un memento para memorizar su estado interno y que pueden restaurarlo a partir de un memento.
  - GestiónEstado es el responsable de la gestión de los mementos y no accede al estado interno de los objetos originales.

El patrón Memento, VI



El patrón Memento, VII

3. Colaboraciones

- Una instancia de GestionEstado solicita un memento al objeto original llamando al método crearMemento, lo salvaguarda y, en caso de necesitar su anulación y retornar al estado memorizando en el memento, lo transmite de nuevo al objeto original mediante el método setMemento.

Dominios de aplicación

- El patrón se utiliza en el caso en que el estado interno de un objeto debe memorizarse (total o parcialmente) para poder ser restaurado posteriormente sin que la encapsulación de este objeto quede fragmentada.

Ejemplo en Java

- Comenzamos presentando este ejemplo escrito en Java con el memento. Este está descrito por la interfaz Memento y la clase MementoImpl. La clase incluye los métodos getEstado y setEstado cuya invocación está reservada al carrito de la compra. La interfaz es vacía, de modo que sólo sirve para determinar un tipo para los demás objetos que deben referenciar el memento sin poder acceder a los métodos getEstado y setEstado.
- El memento almacena el estado del carrito de opciones, a saber una lista que está constituida por un duplicado de la lista de opciones del carrito.

El patrón Observer, I

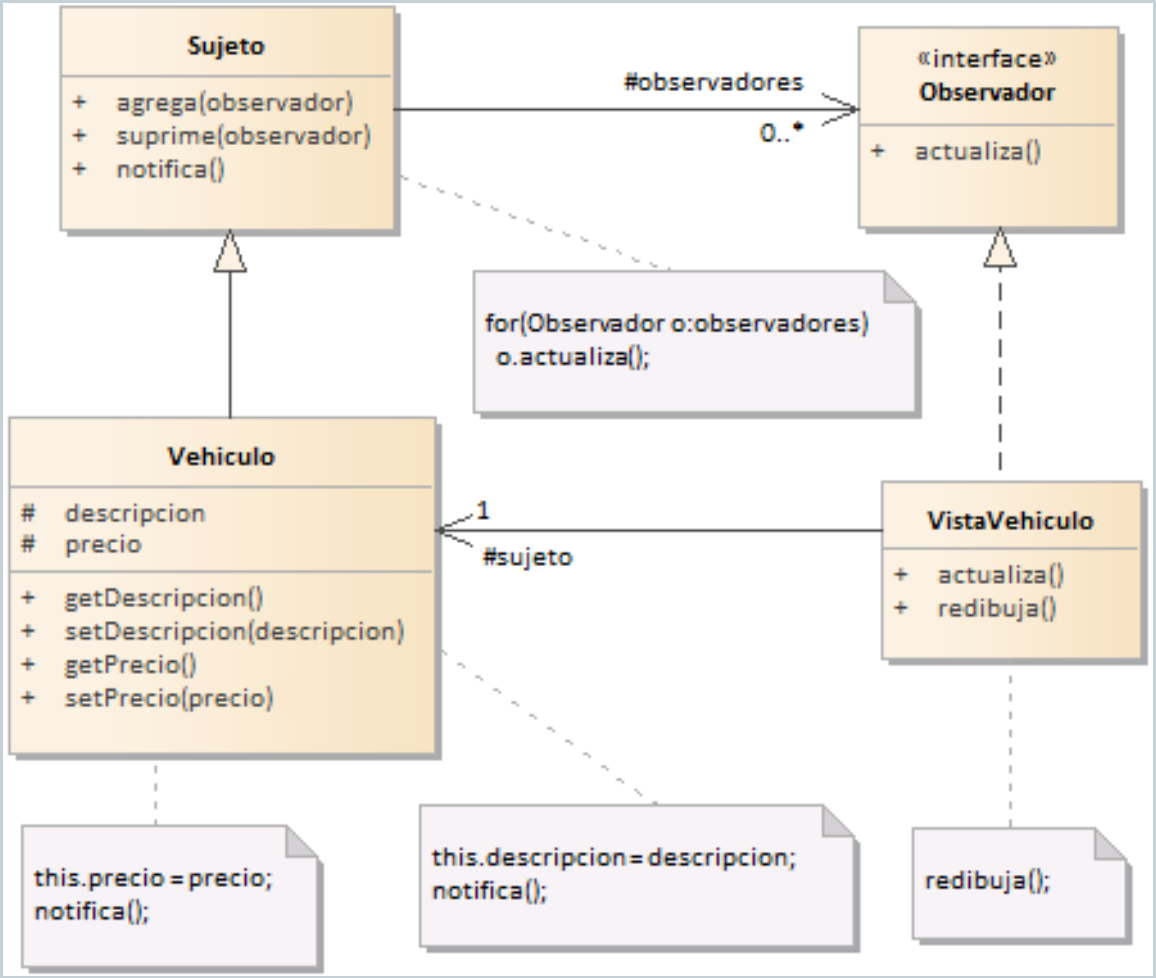
Descripción

- El patrón Observer tiene como objetivo construir una dependencia entre un sujeto y los observadores de modo que cada modificación del sujeto sea notificada a los observadores para que puedan actualizar su estado.

Ejemplo

- Queremos actualizar la visualización de un catálogo en tiempo real. Cada vez que la información relativa a un vehículo se modifica, queremos actualizar la visualización de la misma. Puede haber varias visualizaciones simultáneas.
- La solución recomendada por el patrón Observer, consiste en establecer un enlace entre cada vehículo y sus vistas para que el vehículo pueda indicarles que se actualicen cuando su estado interno haya sido modificado.
- Esta solución se ilustra en la figura de acetato 10.
- El diagrama contiene las cuatro clases siguientes:
  - Sujeto es la clase abstracta que incluye todo objeto que notifica a los demás objetos de las modificaciones en su estado interno.
  - Vehiculo es la subclase concreta de Sujeto que describe a los vehículos. Gestiona dos atributos: descripción y precio.

El patrón Observer, II

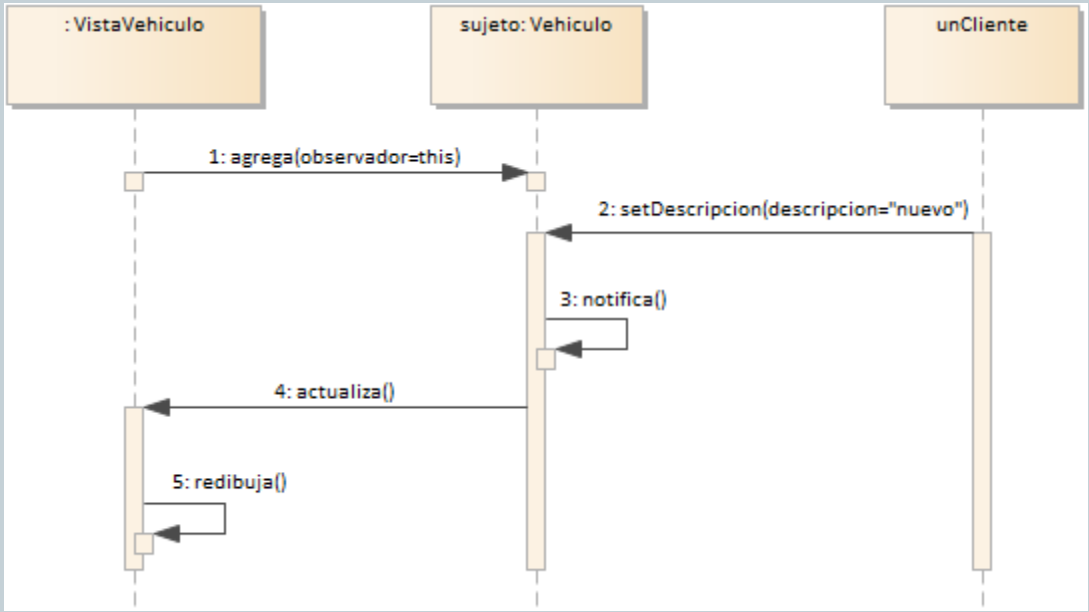


El patrón Observer, III

- Observador es la interfaz de todo objeto que necesite recibir las notificaciones de cambio de estado provenientes de los objetos a los que se ha inscrito previamente.
- VistaVehiculo es la subclase concreta correspondiente a la implementación de Observador cuyas instancias muestran la información de un vehículo.

- El funcionamiento es el siguiente: cada nueva vista se inscribe como observador de su vehículo mediante el método agrega. Cada vez que la descripción o el precio se actualizan, se invoca el método *notifica*. Este solicita a todos los observadores que se actualicen, invocando a su método *actualiza*.
- En la clase VistaVehiculo, éste último método se llama *redibuja*. Este funcionamiento se ilustra en la figura del acetato 12 mediante un diagrama de secuencia.
- La solución implementada mediante el patrón Observer es genérica. En efecto, todo el mecanismo de observación está implementado en la clase Sujeto y en la interfaz Observador, que puede tener otras subclases distintas de Vehiculo y VistaVehiculo.

El patrón Observer, IV



Estructura

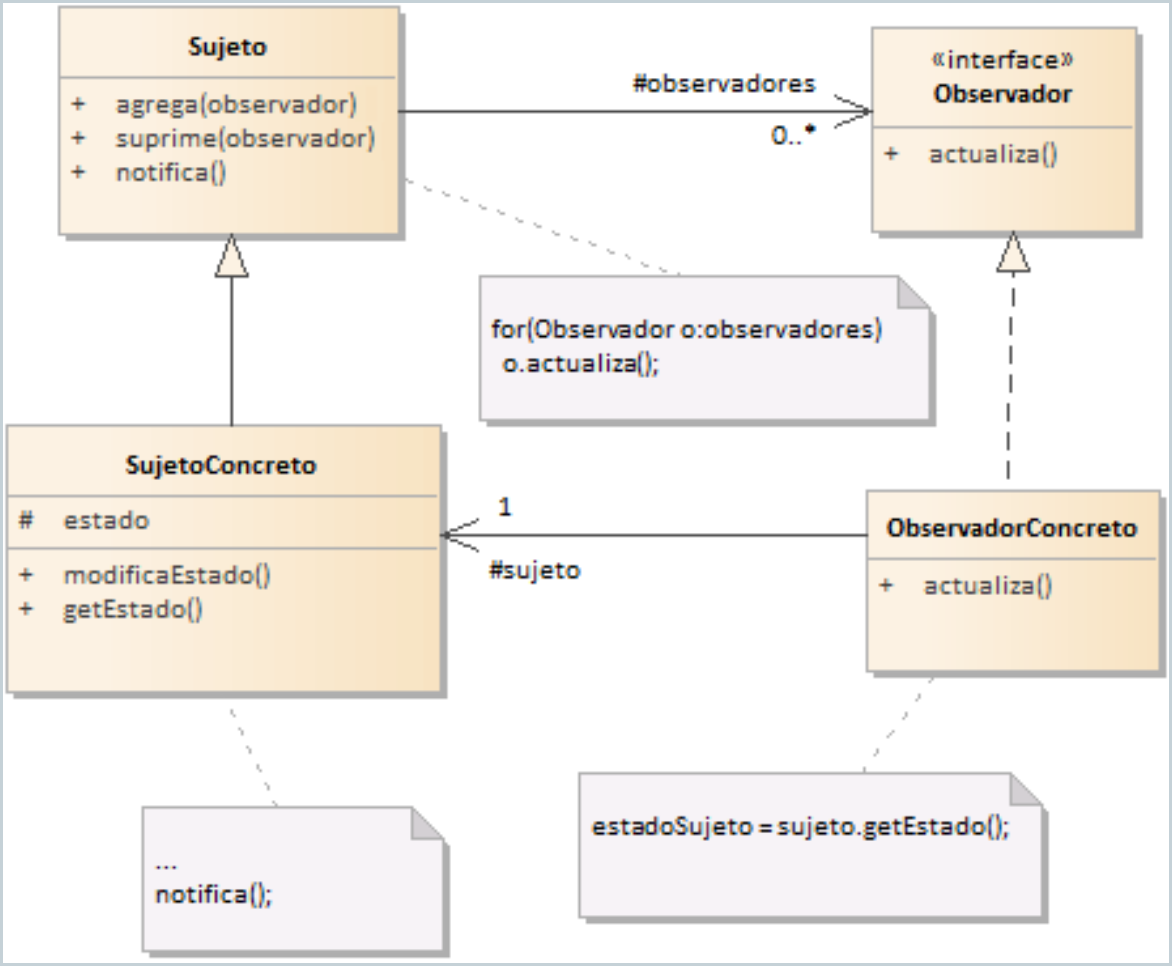
1. Diagrama de clases

- La figura del acetato 13 detalla la estructura genérica del patrón.

2. Participantes

- Los participantes del patrón son los siguientes:

El patrón Observer, V



El patrón Observer, VI

- Sujeto es la clase abstracta que incluye la asociación con los observadores así como los métodos para agregar o suprimir observadores.
- Observador es la interfaz que es necesario implementar para recibir las notificaciones (método *actualiza*).
- SujetoConcreto (Vehiculo) es una clase correspondiente a la implementación de un sujeto. Un sujeto envía una notificación cuando su estado se ha modificado.
- ObservadorConcreto (VistaVehiculo) es una clase de implementación de un observador. Mantiene una referencia hacia el sujeto e implementa el método actualiza. Solicita a su sujeto información que forma parte de su estado durante las actualizaciones invocando al método *getEstado*.

3. Colaboraciones

- El sujeto notifica a sus observadores cuando su estado interno ha sido modificado. Cuando un observador recibe esta notificación, se actualiza en consecuencia. Para realizar esta actualización, puede invocar a los métodos del sujeto que dan acceso a su estado.

El patrón Observer, VII

Dominios de aplicación

- El patrón se utiliza en los siguientes casos:
  - Una modificación en el estado de un objeto genera modificaciones en otros objetos que se determinan dinámicamente.
  - Un objeto quiere avisar a otros objetos sin tener que conocer su tipo, es decir sin estar fuertemente acoplado a ellos.
  - No se desea fusionar dos objetos en uno solo

Ejemplo en Java

- Retomamos el ejemplo de la figura del acetato 10. El código fuente de la clase Sujeto aparece a continuación.
- Los observadores se gestionan mediante una lista.
- Revisar la carpeta del patrón.

El patrón State, I

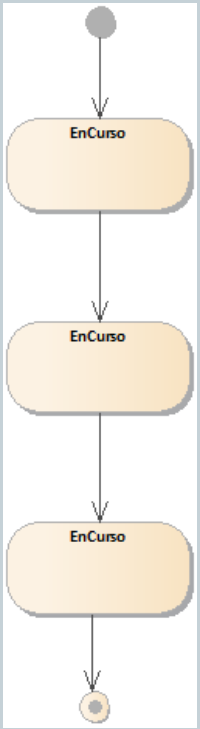
Descripción

- El patrón State permite a un objeto adaptar su comportamiento en función de su estado interno.

Ejemplo

- Nos interesamos a continuación por los pedidos de productos en nuestro sitio de venta en línea.
- Están descritos mediante la clase Pedido. Las instancias de esta clase poseen un ciclo de vida que se ilustra en el diagrama de estados y transiciones de la figura del acetato 17.
- El estado EnCurso se corresponde con el estado en el que el pedido está en curso de creación: el cliente agrega los productos.
- El estado Validado es el estado en el que el pedido ha sido validado y aprobado por el cliente.
- Por último, el estado Entregado es el estado en el que los productos han sido entregados.

El patrón State, II

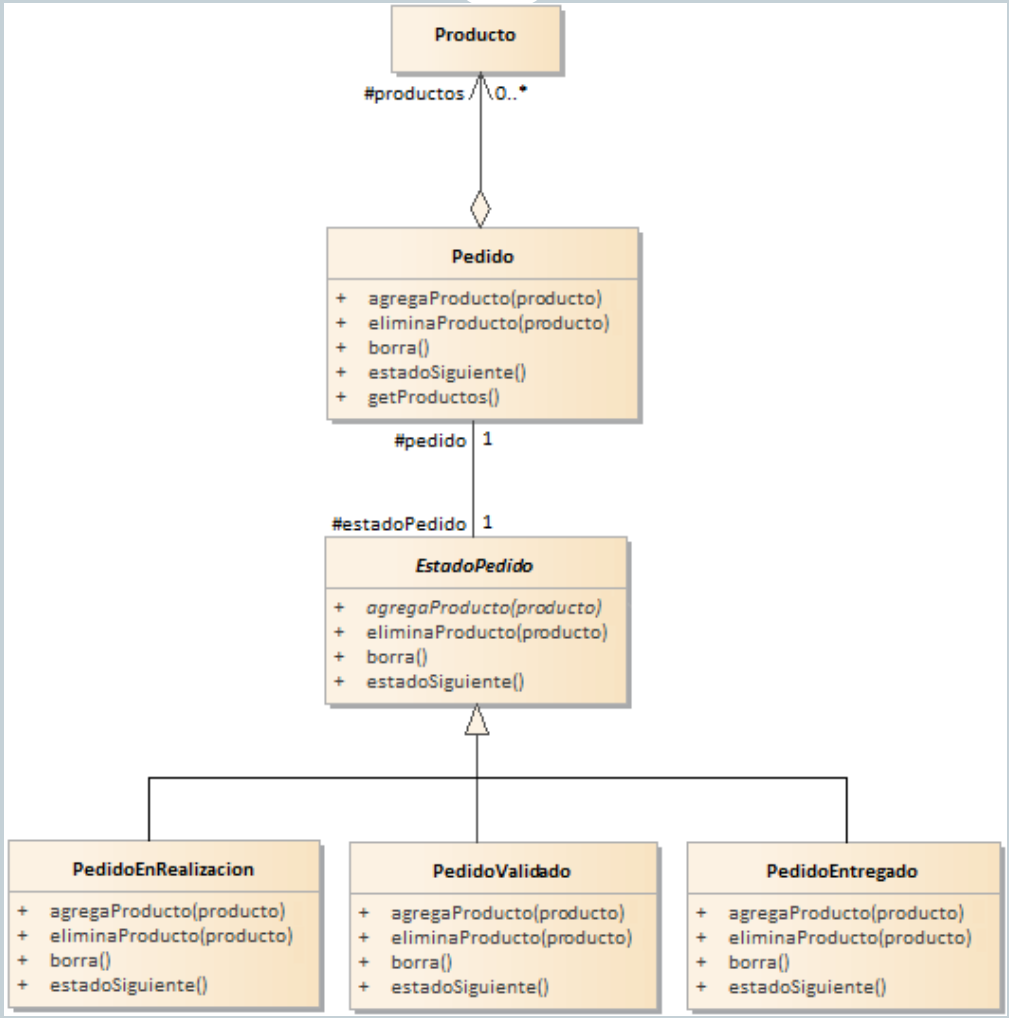


- La clase Pedido posee dos métodos cuyo comportamiento difiere en función de este estado. Por ejemplo, el método agregaProducto añade los productos si el pedido está en el estado EnRealizacion. El método borra no tiene comportamiento en el estado Entregado.

El patrón State, III

- El enfoque tradicional para resolver estas diferencias de comportamiento consiste en utilizar condiciones en el cuerpo de los métodos.
- Este enfoque conduce con frecuencia a métodos complejos de escribir y de aprender.
- El patrón State propone otra solución que consiste en transformar cada estado en una clase. Esta clase presenta los estados dependientes de los métodos de la clase Pedido utilizando su propio comportamiento para este estado.
- La figura del acetato 19 ilustra el diagrama de clases que corresponden a este enfoque.
- Las tres subclases que corresponden a los estados son PedidoEnRealizacion, PedidoValidado y PedidoEntregado. Estas son las subclases de la clase abstracta EstadoPedido que mantiene la asociación con la clase Pedido.
- La clase EstadoPedido introduce igualmente las firmas de los métodos donde el comportamiento depende del estado actual, los métodos implantados en sus subclases.

El patrón State, IV



El patrón State, V

- Una instancia de la clase Pedido posee una referencia hacia una instancia de la subclase de EstadoPedido que corresponde a su estado actual. En esta clase Pedido, los métodos que dependen del estado actual delegan sus invocaciones a esta instancia.

Estructura

1. Diagrama de clases

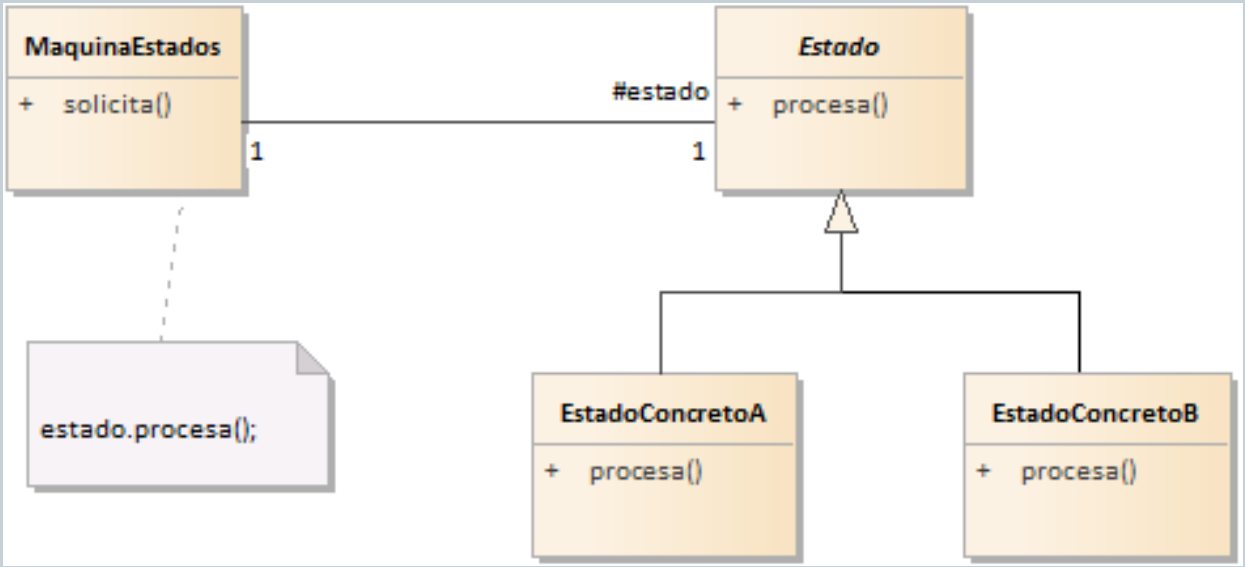
- La figura del acetato 21 ilustra la estructura genérica del patrón.

2. Participantes

- Los participantes del patrón son los siguientes:
  - MaquinaEstados (Pedido) es una clase concreta que describe los objetos que son máquinas de estados, es decir que poseen un conjunto de estados que pueden describirse mediante un diagrama de estados y transiciones. Esta clase mantiene una referencia hacia una instancia de una subclase de Estado que define el estado en curso.
  - Estado (EstadoPedido) es una clase abstracta que incluye los métodos ligados al estado y que gestionan la asociación con la máquina de estados.

El patrón State, VI

- EstadoConcretoA y EstadoConcretoB (PedidoEnCurso, PedidoValidado y PedidoEntregado) son subclases concretas que implementan el comportamiento de los métodos relativos a cada estado.



El patrón State, VII

3. Colaboraciones

- La máquina de estados delega las llamadas a los métodos dependiendo del estado en curso hacia un objeto de estado.
- La máquina de estados puede transmitir al objeto de estado una referencia hacia sí misma si es necesario. Esta referencia puede pasarse durante la delegación o en el propia inicialización del objeto de estado.

Dominios de aplicación

- El patrón se utiliza en los siguientes casos:
  - El comportamiento de un objeto depende de su estado.
  - La implementación de esta dependencia del estado mediante instrucciones condicionales se vuelve muy compleja.

Ejemplo en Java

- A continuación presentamos el ejemplo de la figura del acetato 19 escrito en Java. La clase Pedido se describe a continuación.
- Los métodos agregaProducto, suprimeProducto y borra dependen del estado. Por consiguiente, su implementación consiste en llamar al método correspondiente de la instancia referenciada mediante estadoPedido.

El patrón State, VIII

- El constructor de la clase inicializa el atributo estadoPedido con una instancia de la clase PedidoEnCurso. El método estadoSiguiente pasa al estado siguiente asociando una nueva instancia al atributo estadoPedido.
- Ver la carpeta del patrón.

El patrón Strategy, I

Descripción

- El patrón Strategy tiene como objetivo adaptar el comportamiento y los algoritmos de un objeto en función de una necesidad sin cambiar las interacciones de este objeto con los clientes.
- Esta necesidad puede ponerse de relieve en base a aspectos tales como la presentación, la eficacia en tiempo de ejecución o en memoria, la elección de algoritmos, la representación interna, etc.
- Aunque evidentemente no se trata de una necesidad funcional de cara a los clientes del objeto, pues las interacciones entre el objeto y sus clientes deben permanecer inmutables.

Ejemplo

- En el sistema de venta en línea de vehículos, la clase VistaCatalogo dibuja la lista de vehículos destinados a la venta. Se utiliza un algoritmo de diseño gráfico para calcular la representación gráfica en función del navegador. Existen dos versiones de este algoritmo:

El patrón Strategy, II

- Una primera versión que sólo muestra un vehículo por línea (un vehículo ocupa todo el ancho disponible) y que muestra toda la información posible así como cuatro fotografías.
- Una segunda versión que muestra tres vehículos por línea pero que muestra menos información y una única fotografía.

- La interfaz de la clase VistaCatalogo no depende de la elección del algoritmo de representación gráfica. Esta elección no tiene impacto alguno en relación de una vista de catálogo con sus clientes. Sólo se modifica la representación.
- Una primera solución consiste en transformar la clase VistaCatalogo en una interfaz o en una clase abstracta y en incluir dos subclases de implementación diferentes según la elección del algoritmo. Esto presenta el inconveniente de complicar de manera inútil la jerarquía de las vistas del catálogo.
- Otra posibilidad consiste en implementar ambos algoritmos en la clase VistaCatalogo y en apoyarse en instrucciones condicionales para realizar la elección. No obstante esto consiste en desarrollar una clase relativamente pesada donde el código de los métodos es difícil de comprender.

El patrón Strategy, III

- El patrón Strategy proporciona otra solución incluyendo una clase por algoritmo. El conjunto de las clases así creadas posee una interfaz común que se utiliza para dialogar con la clase VistaCatalogo.
- La figura del acetato 27 muestra el diagrama de clases de la aplicación del patrón Strategy.
- Este diagrama muestra las dos clases de algoritmo: DibujaUnVehiculoPorLinea y DibujaTresVehicuosPorLinea que implementan la interfaz DibujaCatalogo.
- La nota que detalla el método dibuja la clase VistaCatalogo muestra cómo se utilizan ambos métodos.

Estructura

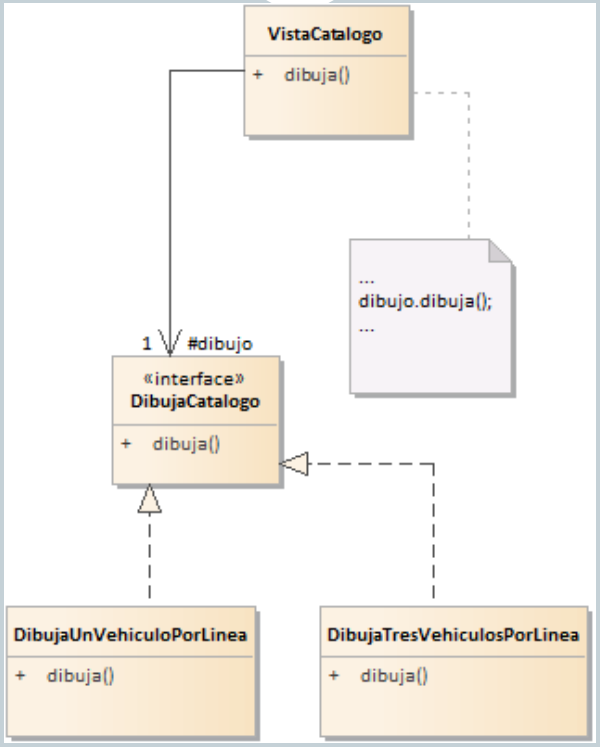
1. Diagrama de clases

- La figura del acetato 28 muestra la estructura genérica del patrón.

2. Participantes

- Los participantes del patrón son los siguientes:

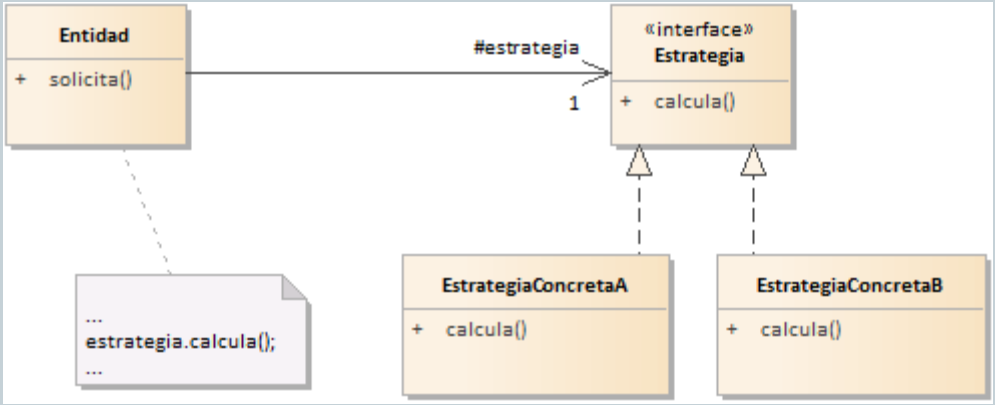
El patrón Strategy, IV



- Estrategia (DibujaCatalogo) es la interfaz común a todos los algoritmos. Esta interfaz se utiliza en Entidad para invocar al algoritmo.
- EstrategiaConcretaA y EstrategiaConcretaB (DibujaUnVehiculoPorLinea y DibujaTresVehiculosPorLinea) son las subclases concretas que implementan los distintos algoritmos.

El patrón Strategy, V

- Entidad es la clase que utiliza uno de los algoritmos de las clases que implementan la Estrategia. Por consiguiente, posee una referencia hacia una de estas clases. Por último, si fuera necesario, puede exponer sus datos internos a las clases de implementación.



3. Colaboraciones

- La entidad y las instancias de las clases de implementación de la Estrategia interactúan para implementar los algoritmos. En el caso más sencillo, los datos que necesita el algoritmo se pasan como parámetro.

El patrón Strategy, VI

- Si fuera necesario, la clase Entidad implementaría los métodos necesarios para dar acceso a sus datos internos.
- El cliente inicializa la entidad con una instancia de la clase de implementación de Estrategia. El mismo selecciona esta clase y, por lo general, no la modifica a continuación. La entidad puede modificar a continuación esta elección.
- La entidad redirige las peticiones provenientes de sus clientes hacia la instancia referenciada por su atributo estrategia.

Dominios de aplicación

- El patrón se utiliza en los casos siguientes:
  - El comportamiento de una clase puede estar implementado mediante distintos algoritmos siendo alguno de ellos más eficaz en términos de ejecución o de consumo de memoria o incluso contener mecanismos de decisión.
  - La implementación de la elección del algoritmo mediante instrucciones condicionales se vuelve demasiado compleja.
  - Un sistema posee numerosas clases idénticas salvo una parte correspondiente a su comportamiento.

El patrón Strategy, VII

- En el último caso, el patrón Strategy permite reagrupar estas clases en una sola, lo que simplifica la interfaz para los clientes.

Ejemplo en Java

- Nuestro ejemplo escrito en Java está basado en la visualización del catálogo de vehículos, simulado aquí simplemente mediante salidas por pantalla.
- La interfaz DibujaCatalogo incluye el método dibuja que recibe como parámetro una lista de instancias de VistaVehiculo.
- Ver la carpeta del patrón.

El patrón Template Method, I

Descripción

- El patrón Template Method permite delegar en las subclases ciertas etapas de una de las operaciones de un objeto, estando estas etapas descritas en las subclases.

Ejemplo

- En el sistema de venta en línea de vehículos, queremos gestionar pedidos de clientes de México y de USA. La diferencia entre ambas peticiones concierne principalmente al cálculo del IVA.
- Si bien en México la tasa de IVA es siempre fija del 21%, en el caso de USA es variable (12% para los servicios, 15% para el material). El cálculo del IVA requiere dos operaciones de cálculo distintas en función del país.
- Una primera solución consiste en implementar dos clases distintas sin superclase común: PedidoMexico y PedidoUSA.
- Esta solución presenta el inconveniente importante de que tiene código idéntico que no ha sido factorizado, como por ejemplo la visualización de la información del pedido (método visualiza).

El patrón Template Method, II

- Podría incluirse una clase abstracta Pedido para factorizar los métodos comunes, como el método visualiza.
- El patrón Template Method permite ir más lejos proponiendo factorizar el código común en el interior de los métodos.
- Tomemos el ejemplo del método calculaImporteConIVA cuyo algoritmo es el siguiente para Mexico (escrito en pseudocódigo).

  calculaImporteConIVA:

  importeIVA = importeSinIVA*0.21;

  importeIVA = importeSinIVA + importeIVA;
- El algoritmo para USA tiene el siguiente pseudocódigo.

  calculaImporteConIVA:

  importeIVA = (importeServiciosSinIVA*0.12) + (importeMaterialSinIVA*0.15);

  importeIVA = importeSinIVA + importeIVA;

El patrón Template Method, III

- Veamos en este ejemplo que la última línea del método es común a ambos países (en este ejemplo, sólo hay una línea común aunque en un caso real la parte común puede ser mucho más importante).
- Reemplazamos la primera línea por una llamada a un nuevo método llamado calculaIVA. De este modo el método calculaImporteConIVA se describe en delante de la siguiente forma:

  calculaImporteConIVA:

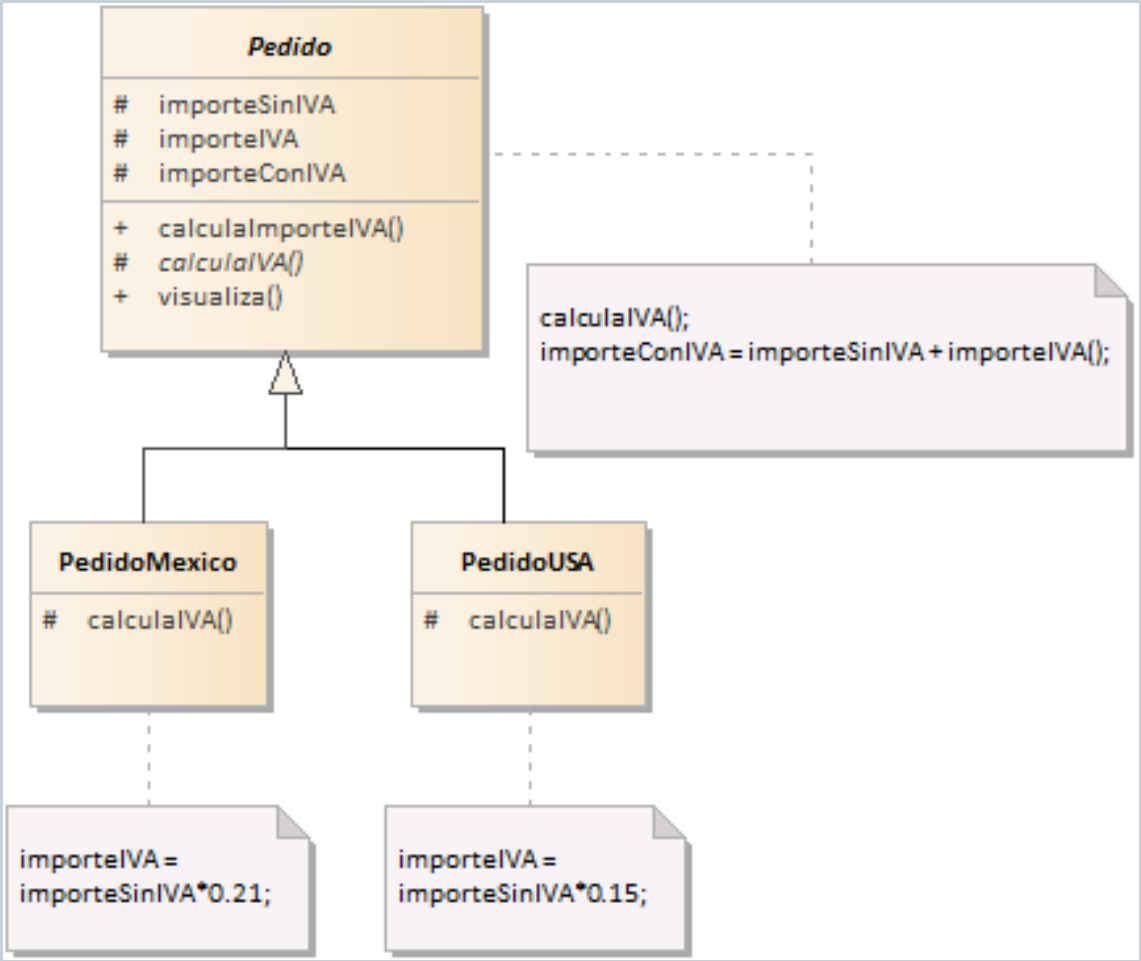
  calculaIVA();

  importeIVA = importeSinIVA + importeIVA;
- El método calculaImporteConIVA puede ahora factorizarse. El código específico ha sido desplazado en el método calculaIVA, cuya implementación es específica para cada país.
- El método calculaIVA se incluye en la clase Pedido como método abstracto.

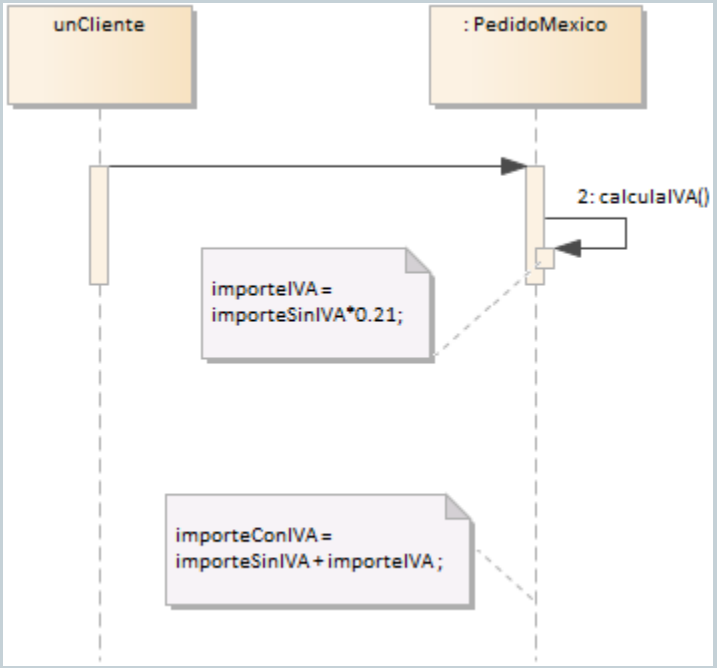
El patrón Template Method, IV

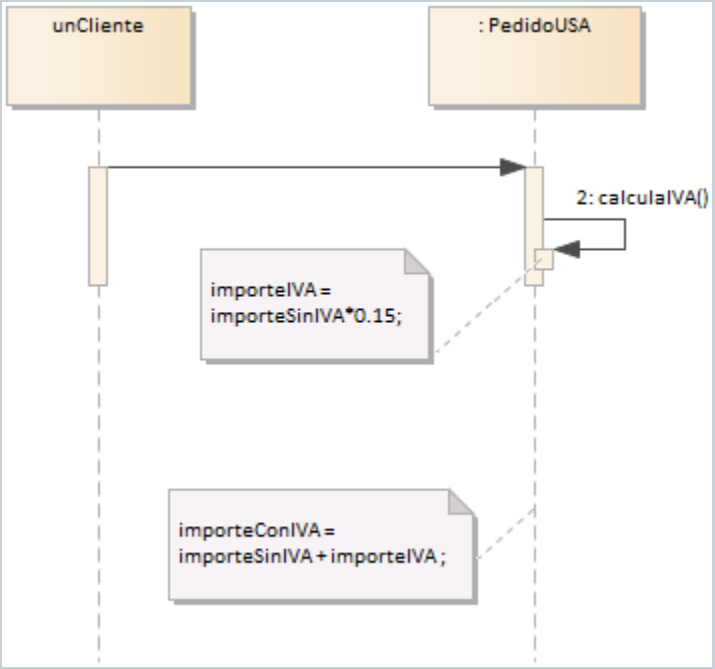
- El método calculaImporteConIVA se llama un método “modelo” (template method). Un método “modelo” incluye la parte común de un algoritmo que está complementado por partes específicas.
- Esta solución se ilustra en el diagrama de clases de la figura del acetato 35 donde, por motivos de simplicidad, el cálculo del IVA de Guatemala se ha configurado con una tasa única del 15%.
- Cuando un cliente invoca al método calculaImporteConIVA de un pedido, éste invoca al método calculaIVA.
- La implementación de este método depende de la clase concreta del pedido:
  - Si esta clase es PedidoMexico, el diagrama de secuencia se describe en la figura del acetato 36 (arriba). El IVA se calcula basado en la tasa del 21%.
  - Si esta clase es PedidoUSA, el diagrama de secuencia se describe en la figura del acetato 36 (abajo). El IVA se calcula basado en la tasa del 15%.

El patrón Template Method, V



El patrón Template Method, VI





El patrón Template Method, VII

Estructura

1. Diagrama de clases

- La figura del acetato 38 muestra la estructura genérica del patrón.

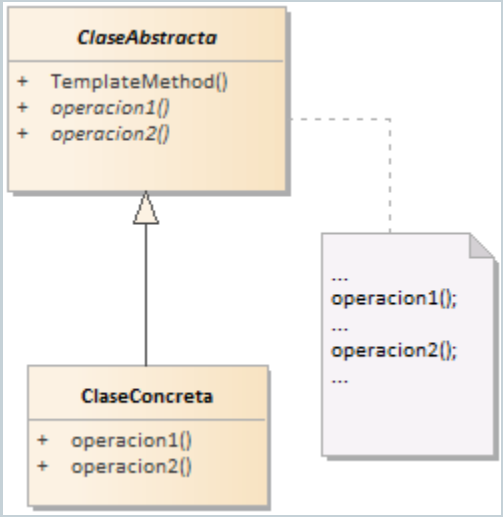
2. Participantes

- Los participantes del patrón son los siguientes:
  - La clase abstracta ClaseAbstracta (Pedido) incluye el método "modelo" así como la firma de los métodos abstractos que invoca este método.
  - La subclase concreta ClaseConcreta (PedidoMexico y PedidoUSA) implementa los métodos abstractos utilizados por el método "modelo" de la clase abstracta. Puede haber varias clases concretas.

3. Colaboraciones

- La implementación del algoritmo se realiza mediante la colaboración entre el método "modelo" de la clase abstracta y los métodos de una subclase concreta que complementa el algoritmo.

El patrón Template Method, VIII



Dominios de aplicación

- El patrón se utiliza en los siguientes casos:
  - Una clase compartida con otra u otras clases con código idéntico que puede factorizarse siempre que las partes específicas a cada clase hayan sido desplazadas a nuevos métodos.
  - Un algoritmo posee una parte invariable y partes específicas a distintos tipos de objetos.

Ejemplo en Java

- La clase abstracta Pedido incluye el método “modelo” calculaImporteConIVA que invoca al método abstracto calculaIVA.

El patrón Visitor, I

Descripción

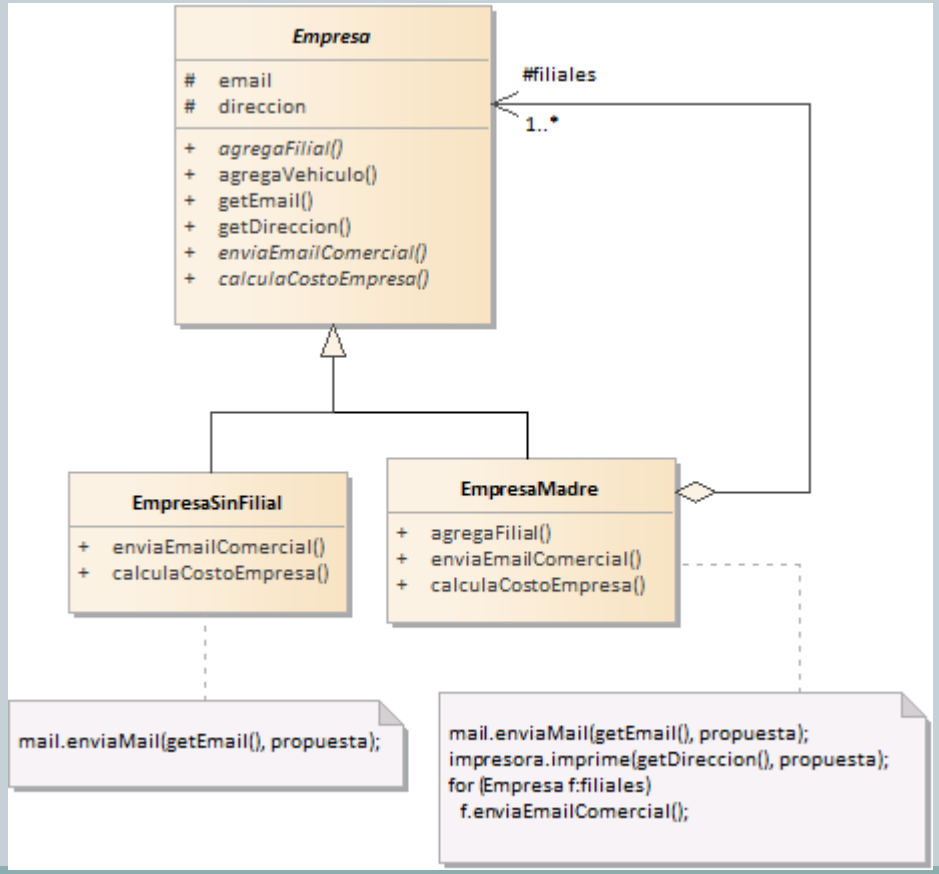
- El patrón Visitor construye una operación que debe realizarse sobre los elementos de un conjunto de objetos. Esto permite agregar nuevas operaciones sin modificar las clases de estos objetos.

Ejemplo

- Consideremos la figura del acetato 40 que describe los clientes de nuestro sistema organizados bajo la forma de objetos compuestos según el patrón Composite. A excepción del método agregaFilial, específico a la gestión de la composición, las dos subclases poseen dos métodos con el mismo nombre: calculaCostoEmpresa y enviaEmailComercial.
- Cada uno de estos métodos se corresponde con una funcionalidad cuya implementación está bien adaptada en función de la clase.
- Podrían implementarse muchas otras funcionalidades como, por ejemplo, el cálculo de la cifra de negocios de un cliente (incluyendo o no sus filiales), etc.

El patrón Visitor, II

- En el diagrama, el cálculo del costo de mantenimiento no está detallado. El detalle se encuentra en la parte dedicada al patrón Composite.



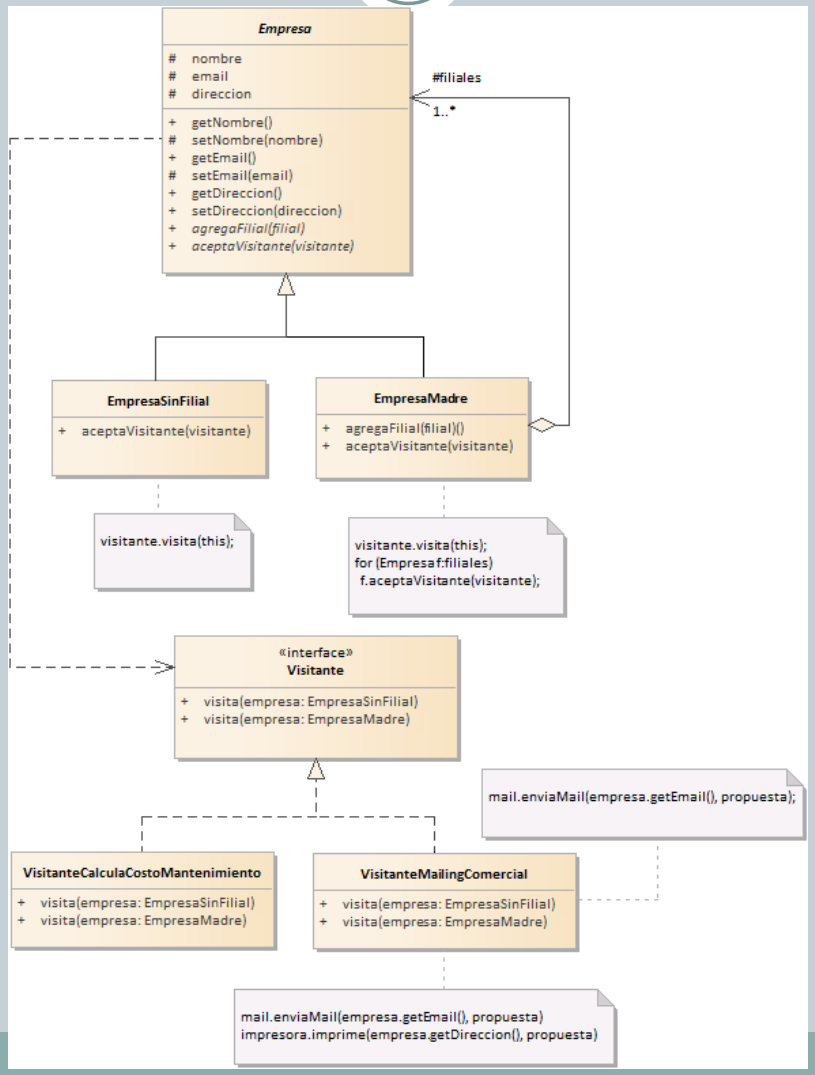
El patrón Visitor, III

- Este enfoque puede utilizarse siempre y cuando el número de funcionalidades sea pequeño. En cambio, si se vuelve importante, obtendremos clases con muchos métodos, difíciles de comprender y de mantener.
- Además, estas funcionalidades darán lugar a métodos (calculaCostoEmpresa y enviaEmailComercial) sin relación entre ellos y sin relación entre el núcleo de los objetos con la diferencia, por ejemplo, del método agregaFilial que contribuye a componer los objetos.
- El patrón Visitor permite implementar las nuevas funcionalidades en un objeto separado llamado visitante.
- Cada visitante establece una funcionalidad para varias clases incluyendo para cada una de ellas un método de implementación llamado visita y cuyo parámetro está tipado según la clase a visitar.
- A continuación, el visitante se transmite al método aceptaVisitante de estas clases. Este método invoca al método del visitante correspondiente a su clase. Sea cual sea el número de funcionalidades a implementar en un conjunto de clases, sólo debe escribirse el método aceptaVisitante.

El patrón Visitor, IV

- Puede ser necesario ofrecer la posibilidad al visitante de acceder a la estructura interna del objeto visitado (preferentemente mediante accesos en modo lectura como en este caso los métodos getNombre, getEmail y getDireccion).
- Si los objetos son compuestos entonces su método aceptaVisitante llama al método aceptaVisitante de sus componentes.
- Es el caso aquí para cada instancia de la clase EmpresaMadre que llama al método aceptaVisitante de sus filiales.
- El diagrama de clases de la figura del acetato 43 ilustra la implementación del patrón Visitor.
- La interfaz visitante incluye la firma de los métodos que implementan la funcionalidad para cada clase a visitar.
- Esta interfaz posee dos subclases de implementación, una por cada funcionalidad.

El patrón Visitor, V



El patrón Visitor, VI

Estructura

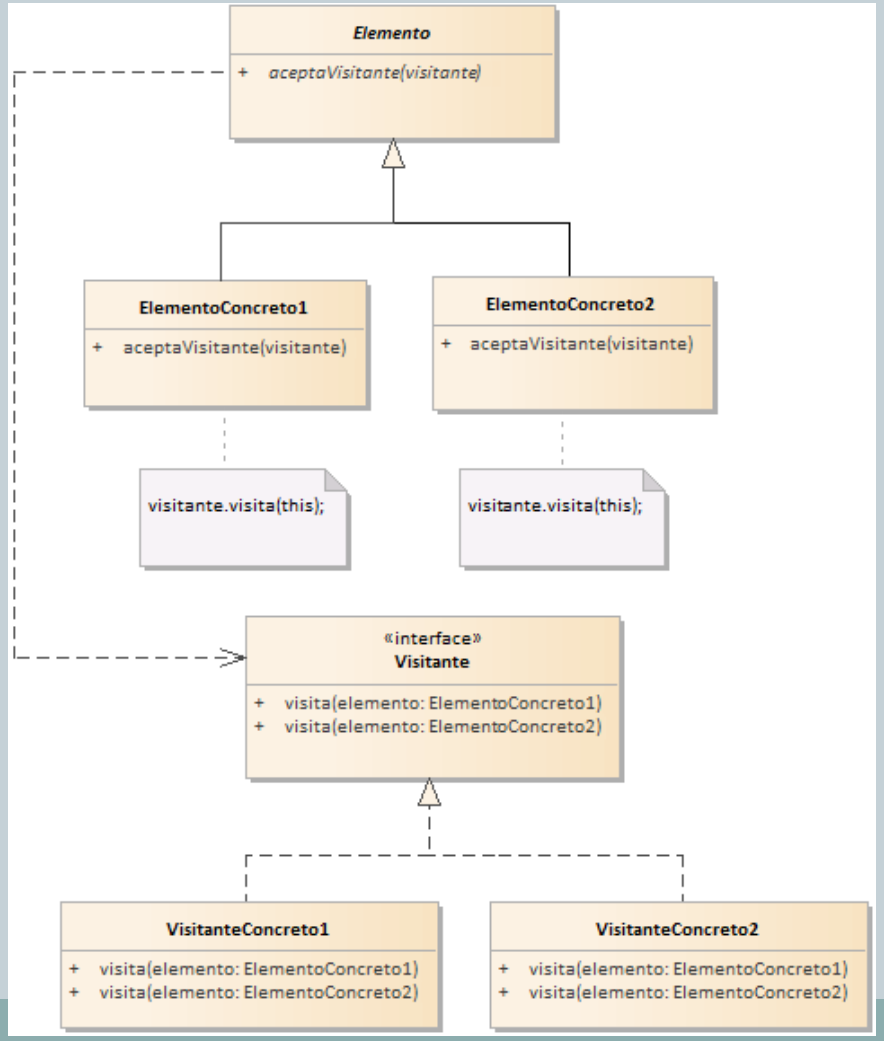
1. Diagrama de clases

- La figura del acetato 45 detalla la estructura genérica del patrón.

2. Participantes

- Los participantes del patrón son los siguientes:
  - Visitante es la interfaz que incluye la firma de los métodos que realizan una funcionalidad en un conjunto de clases. Existe un método para cada clase que recibe como argumento una instancia de esta clase.
  - VisitanteConcreto1 y VisitanteConcreto2 (VisitanteCalculoCostoEmpresa y VisitanteMailingComercila) implementan los métodos que realizan la funcionalidad correspondiente a la clase.
  - Elemento (Empresa) es una clase abstracta superclase de las clases de elementos. Incluye el método abstracto aceptaVisitante que acepta un visitante como argumento.
  - ElementoConcreto1 y ElementoConcreto2 (EmpresaSinFilial y EmpresaMadre) implementan el método aceptaVisitante que consiste en volver a llamar al visitante a través del método correspondiente de la clase.

El patrón Visitor, VII



El patrón Visitor, VIII

3. Colaboraciones

- Un cliente que utiliza un visitante debe en primer lugar crearlo como instancia de la clase de su elección y, a continuación, pasarlo como argumento al método aceptaVisitante de un conjunto de elementos.
- El elemento vuelve a llamar al método del visitante que corresponde con su clase.
- Le pasa una referencia hacia sí mismo como argumento para que el visitante pueda acceder a su estructura interna.

Dominios de aplicación

- El patrón se utiliza en los siguientes casos:
  - Es necesario agregar numerosas funcionalidades a un conjunto de clases sin volverlas pesadas.
  - Un conjunto de clases poseen una estructura fija y es necesario agregarles funcionalidades sin modificar su interfaz.
- Si la estructura del conjunto de clases a las que es necesario agregar funcionalidades cambia a menudo, el patrón Visitor no se adapta bien.
- En efecto, cualquier modificación de la estructura implica una modificación de cada visitante, lo cual puede tener un costo elevado.

El patrón Visitor, IX

Ejemplo en Java

- Retomamos el ejemplo de la figura del acetato 43.
- A continuación se muestra el código de la clase Empresa escrito en Java.
- El método aceptaVisitante es abstracto pues su código depende de la subclase.
- Ver el código del patrón.