Patrones de estructuración

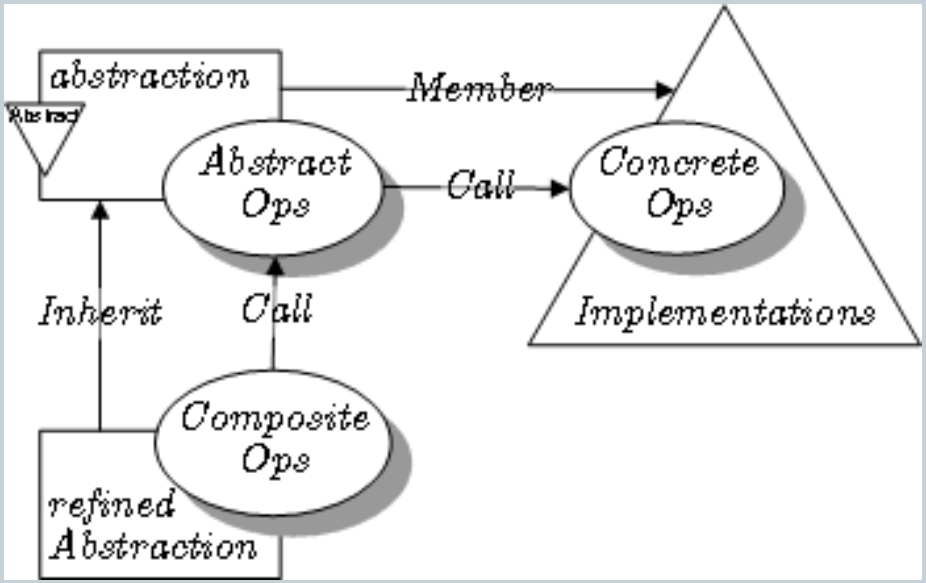
ABRAHAM SÁNCHEZ LÓPEZ
GRUPO MOVIS
FCC-BUAP

Introducción, I

- El objetivo de los patrones de estructuración es facilitar la independencia de la interfaz de un objeto o de un conjunto de objetos respecto de su implementación.
- En el caso de un conjunto de objetos, se trata también de hacer que esta interfaz sea independiente de la jerarquía de clases y de la composición de los objetos.
- Proporcionando interfaces, los patrones de estructuración encapsulan la composición de objetos, aumentan el nivel de abstracción del sistema de forma similar a como los patrones de creación encapsulan la creación de objetos. Los patrones de estructuración ponen de relieve las interfaces.
- La encapsulación de la composición no se realiza estructurando al objeto en sí mismo sino transfiriendo esta estructuración a un segundo objeto. Este queda íntimamente ligado al primero.
- Esta transferencia de estructuración significa que el primer objeto posee la interfaz de cara a los clientes y administra la relación con el segundo objeto que gestiona la composición y no tiene ninguna interfaz con los clientes externos.

Introducción, II

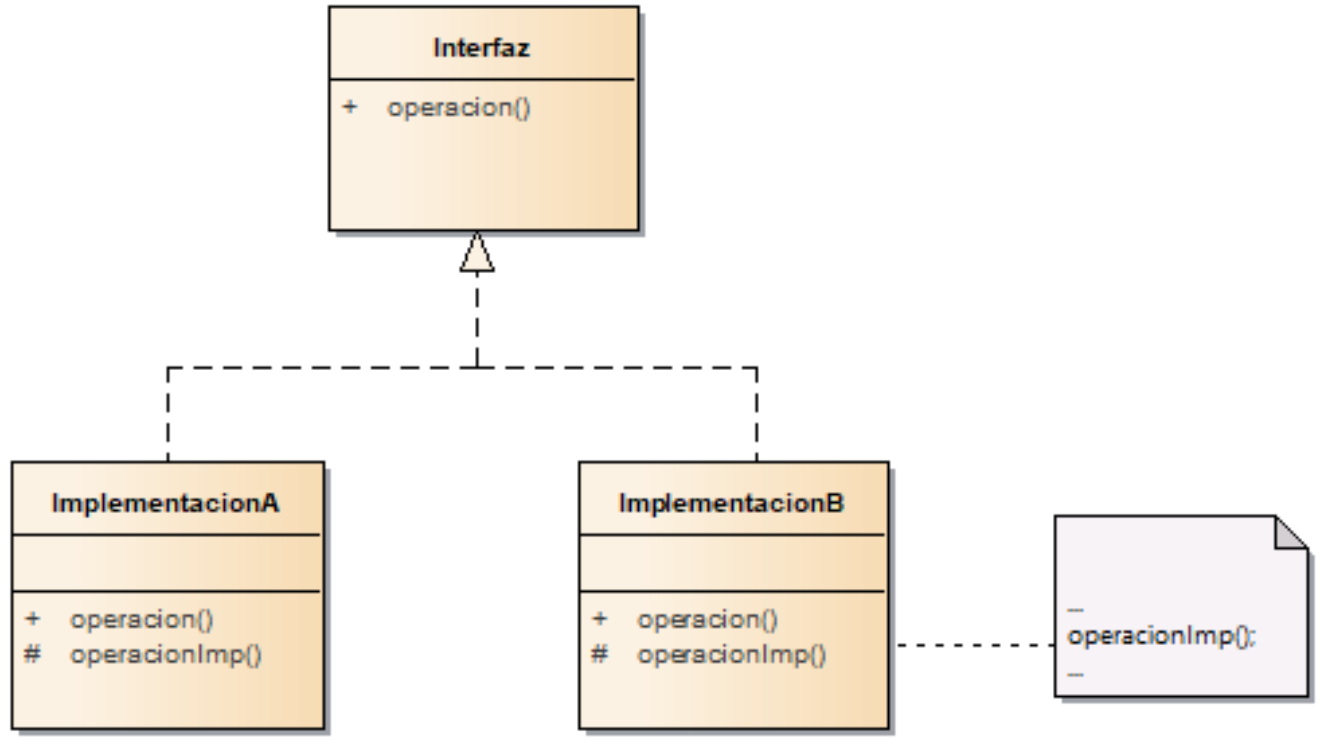
- Esta realización ofrece otra mejora que es la flexibilidad en la composición, la cual puede modificarse de manera dinámica.
- En efecto, es sencillo sustituir un objeto por otro siempre que sea la misma clase o que respete la misma interfaz.
- Los patrones Composite, Decorator y Bridge son un buen ejemplo de este mecanismo.



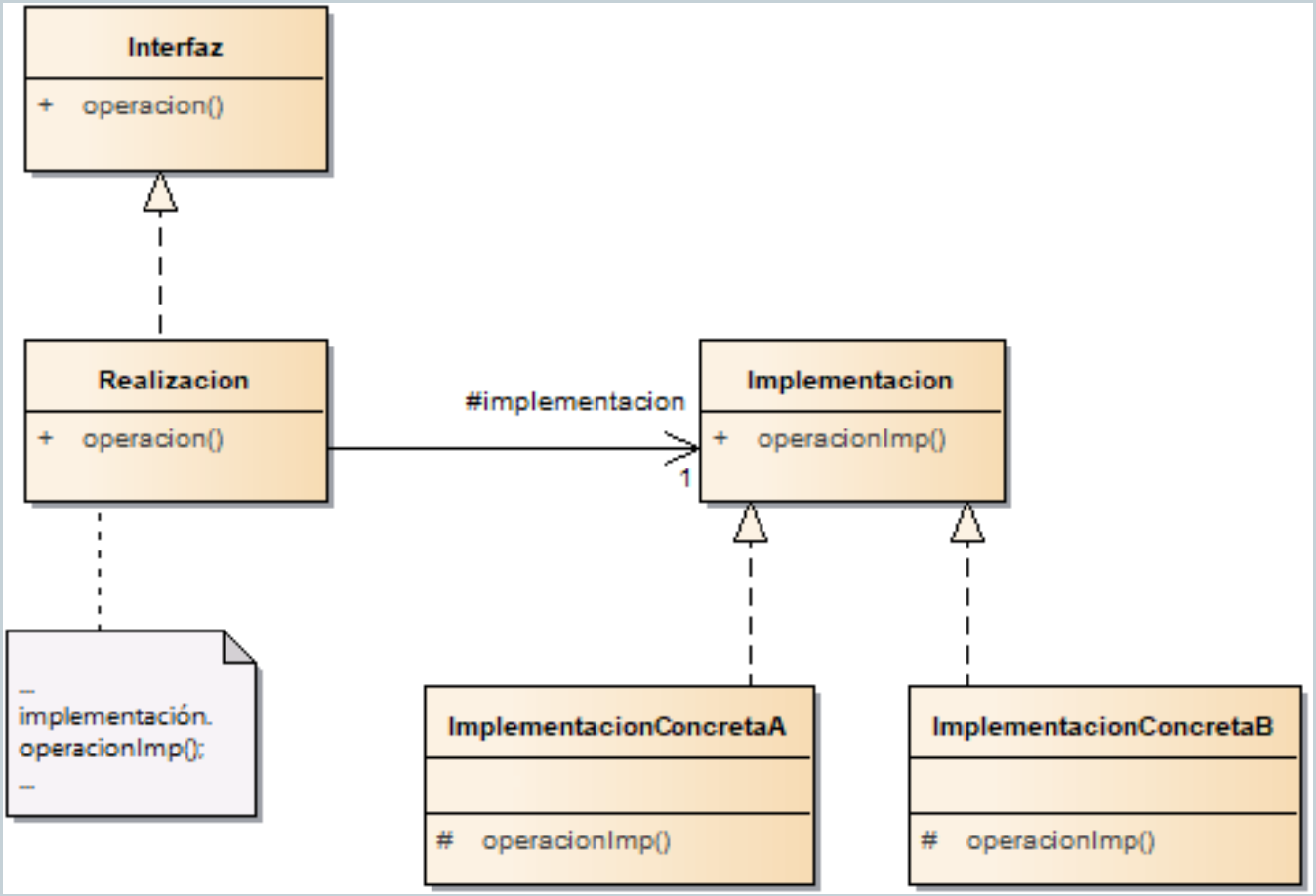
Composición estática y dinámica, I

- Tomemos el ejemplo de los aspectos de implementación de una clase. Situémonos en un marco en el que podamos tener varias implementaciones posibles.
- La solución clásica consiste en diferenciarlas a nivel de las subclases. Es el caso del uso de la herencia de una interfaz en varias clases de implementación, como ilustra el diagrama de clases de la figura del acetato 5.
- Esta solución consiste en realizar una composición estática. En efecto, una vez que se ha escogido la clase de implementación de un objeto, no es posible cambiarla.
- Como se ha explicado en la sección anterior, otra solución consiste en separar el aspecto de implementación en otro objeto tal y como se muestra en la figura del acetato 6.
- Las secciones correspondientes a la implementación se gestionan mediante una instancia de la clase ImplementaciónConcretaA o mediante una instancia de la clase ImplementaciónConcretaB.
- Esta instancia está referenciada por el atributo implementación. Puede sustituirse fácilmente por otra instancia durante la ejecución. Por ello, se dice que la composición es dinámica.

Implementación de un objeto por herencia



Implementación de un objeto por asociación



Composición estática y dinámica, II

- **Nota**: Esta solución presenta también la ventaja de encapsular la sección de implementación y la vuelve totalmente transparente a los clientes.
- Todos los patrones de estructuración están basados en el uso de uno o varios objetos que determinan la estructuración.
- La siguiente lista describe la función que cumple este objeto en cada patrón.
  - **Adapter**: adapta un objeto existente.
  - **Bridge**: implementa un objeto.
  - **Composite**: organiza la composición jerárquica de un objeto.
  - **Decorator**: se sustituye el objeto existente agregándole nuevas funcionalidades.
  - **Facade**: se sustituye un conjunto de objetos existentes confiriéndoles una interfaz unificada.
  - **Flyweight**: está destinado a la compartición y guarda un estado independiente de los objetos que lo referencian.
  - **Proxy**: se sustituye el objeto otorgando un comportamiento adaptado a necesidades de optimización o de protección.

El patrón Adapter, I

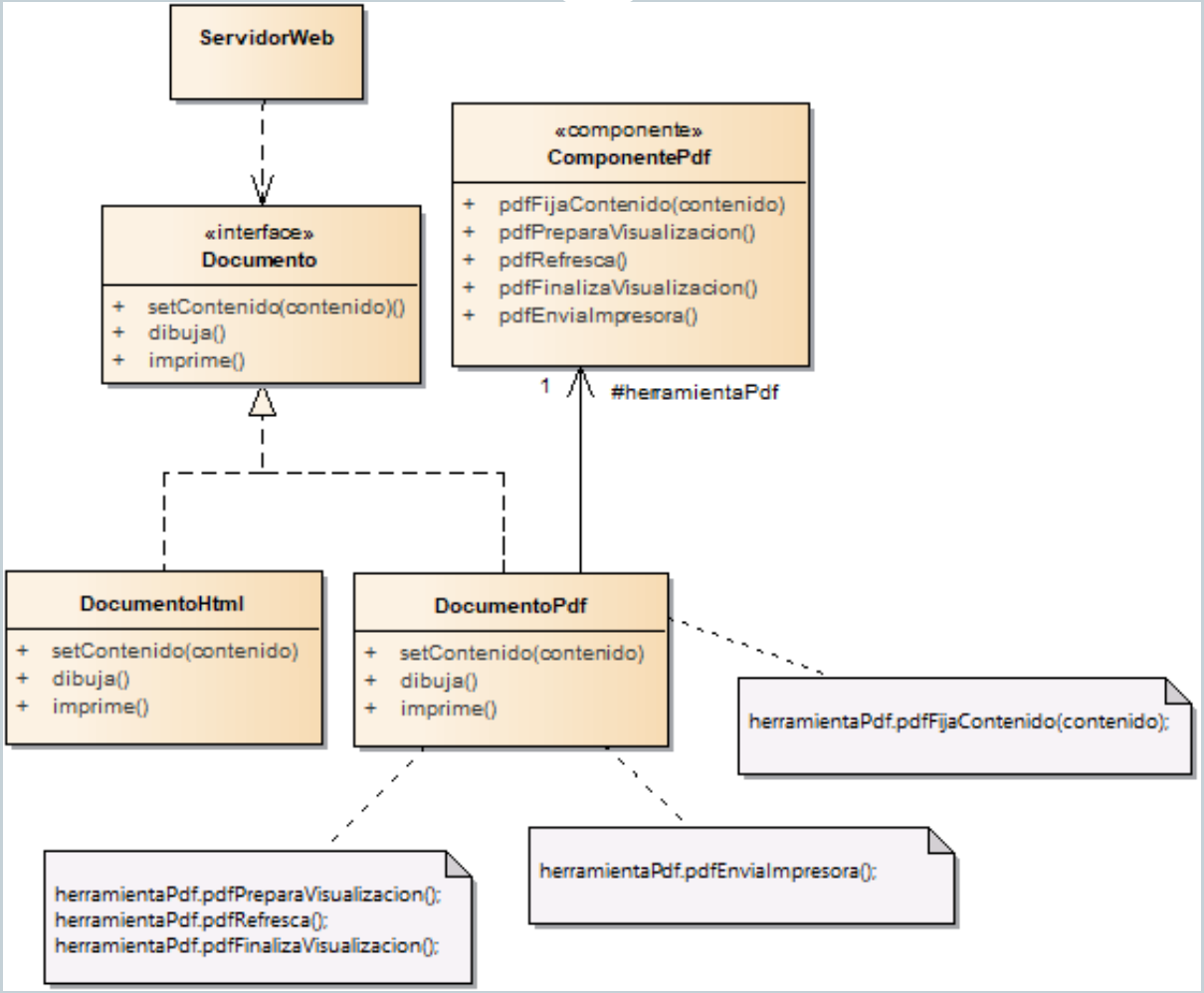
Descripción

- El objetivo del patrón Adapter es convertir la interfaz de una clase existente en la interfaz esperada por los clientes también existentes de modo que puedan trabajar de manera conjunta.
- Se trata de conferir a una clase existente una nueva interfaz para responder a las necesidades de los clientes.

Ejemplo

- El servidor web del sistema de venta de vehículos crea y administra los documentos destinados a los clientes. La interfaz Documento se ha definido para realizar esta gestión.
- La siguiente figura muestra su representación UML así como los tres métodos setContenido, dibuja e imprime. Se ha realizado una primera clase de implementación de esta interfaz: la clase DocumentoHtml que implementa estos tres métodos. Los objetos clientes de esta interfaz y esta clase cliente ya se han diseñado.

El patrón Adapter, II



El patrón Adapter, III

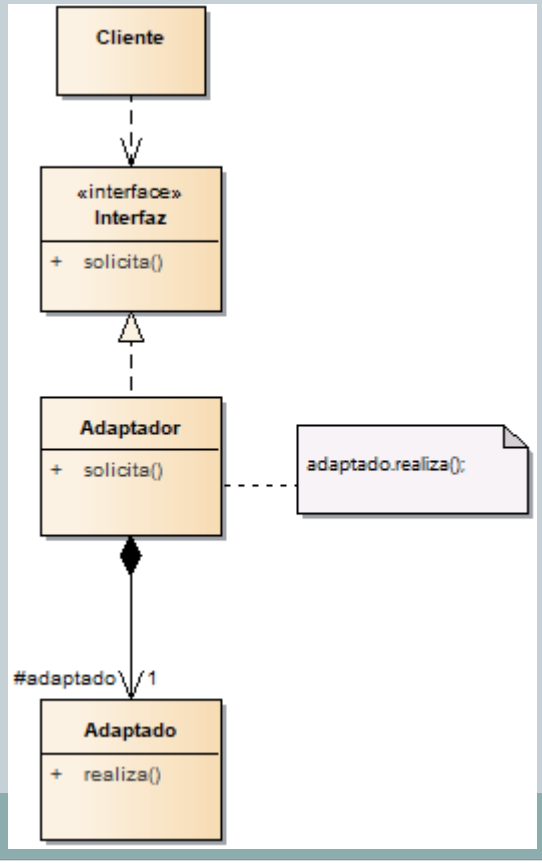
- Por otro lado, la agregación de documentos PDF supone un problema, pues se trata de documentos más complejos de construir y de administrar que los documentos HTML.
- Para ello se ha escogido un producto del mercado, aunque su interfaz no se corresponde con la interfaz Documento. La figura del acetato 9 muestra el componentePdf cuya interfaz incluye más métodos y la nomenclatura es por supuesto muy diferente (con el prefijo pdf).
- El patrón Adapter proporciona una solución que consiste en crear la clase DocumentoPdf que implemente la interfaz Documento y posea una asociación con ComponentePdf.
- La implementación de los tres métodos de la interfaz Documento consiste en delegar correctamente las llamadas al componente PDF.
- Esta solución se muestra igualmente en la figura del acetato 9, el código de los métodos se detalla con ayuda de notas.

El patrón Adapter, IV

Estructura

1. Diagrama de clases

- La siguiente figura detalla la estructura genérica del patrón.



El patrón Adapter, V

2. Participantes

- Los participantes del patrón son los siguientes
  - Interfaz (Documento) incluye la firma de los métodos del objeto.
  - Cliente (ServidorWeb) interactúa con los objetos respondiendo a la interfaz Interfaz.
  - Adaptador (DocumentoPdf) implementa los métodos de la interfaz Interfaz invocando a los métodos del objeto adaptado.
  - Adaptado (ComponentePdf) incluye el objeto cuya interfaz ha sido adaptada para corresponder a la interfaz Interfaz.

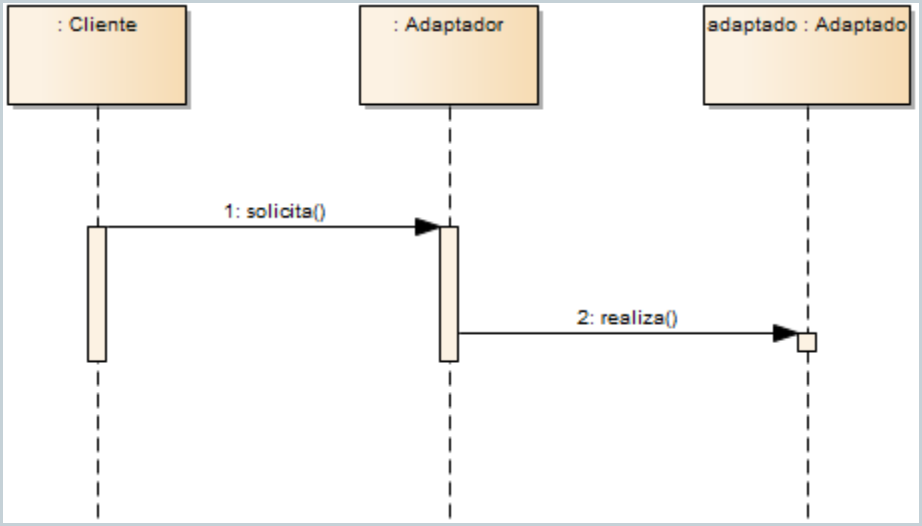
3. Colaboraciones

- El cliente invoca el método solicitud del adaptador que, en consecuencia, interactúa con el objeto adaptado invocando el método realiza. La figura del acetato 13 ilustra estas colaboraciones.

Dominios de aplicación

- El patrón se utiliza en los siguientes casos:

El patrón Adapter, VI



- Para integrar en el sistema un objeto cuya interfaz no se corresponde con la interfaz requerida en el interior de este sistema.
- Para proveer interfaces múltiples a un objeto en su etapa de diseño.

Ejemplo en PHP

- A continuación se presenta el código del ejemplo en Java.

El patrón Bridge, I

Descripción

- El objetivo del patrón Bridge es separar el aspecto de implementación de un objeto de su aspecto de representación y de interfaz.
- De este modo, por un lado la implementación puede encapsularse por completo y por otro lado la implementación y la representación pueden evolucionar de manera independiente y sin que ninguna suponga restricción alguna sobre la otra.

Ejemplo

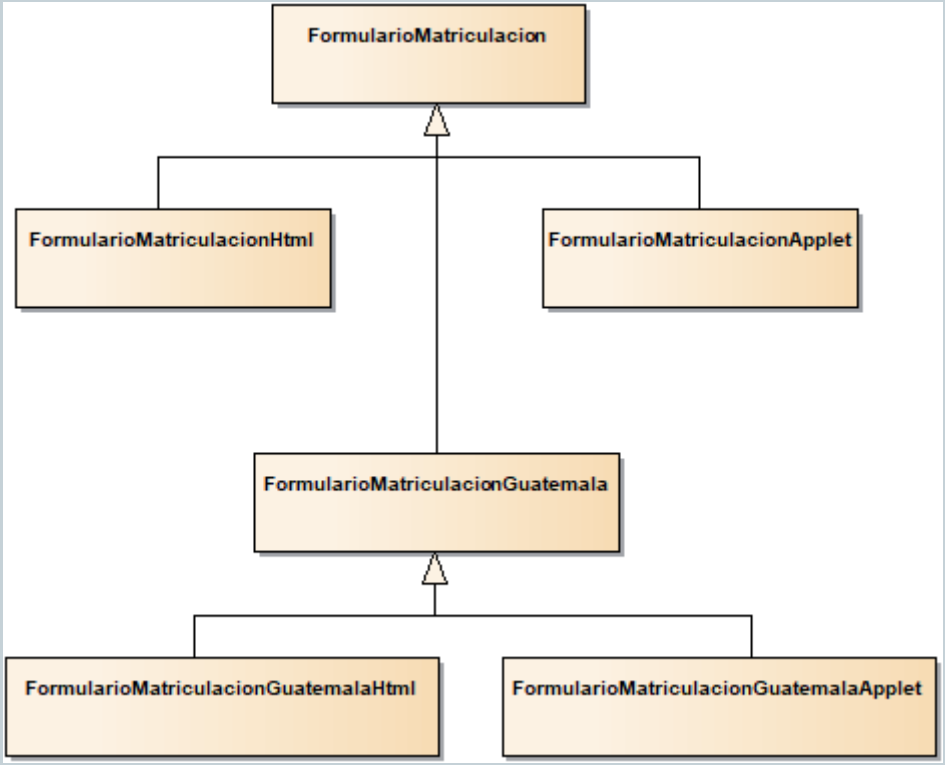
- Para realizar la solicitud de emplacamiento de un vehículo, conviene precisar sobre esta solicitud cierta información importante como el número de placa existente. El sistema muestra un formulario para solicitar esta información.
- Existen dos implementaciones de los formularios:
  - Formularios HTML;
  - Formularios basados en un applet Java
- Por lo tanto es posible introducir una clase abstracta FormularioMatriculacion y dos subclases concretas FormularioMatriculacionHtml y FormularioMatriculacionApplet.

El patrón Bridge, II

- En una primera etapa, las solicitudes de emplacamiento sólo afectan a México.
- A continuación, se hace necesario introducir una nueva subclase de FormularioMatriculacion correspondiente a las solicitudes de emplacamiento o matriculación de Guatemala, subclase llamada FormularioMatriculacionGuatemala.
- Esta subclase debe a su vez ser abstracta y tener dos subclases concretas por cada implementación.
- La figura del acetato 16 muestra el diagrama de clases correspondiente.
- Este diagrama pone de manifiesto dos problemas:
  - La jerarquía mezcla al mismo nivel subclases de implementación y una subclase de representación: FormularioEmplacamientoGuatemala.

    Además para cada representación es preciso introducir dos subclases de implementación, lo cual conduce rápidamente a una jerarquía muy compleja.
  - Los clientes son dependientes de la implementación. En efecto, deben interactuar con las clases concretas de implementación.

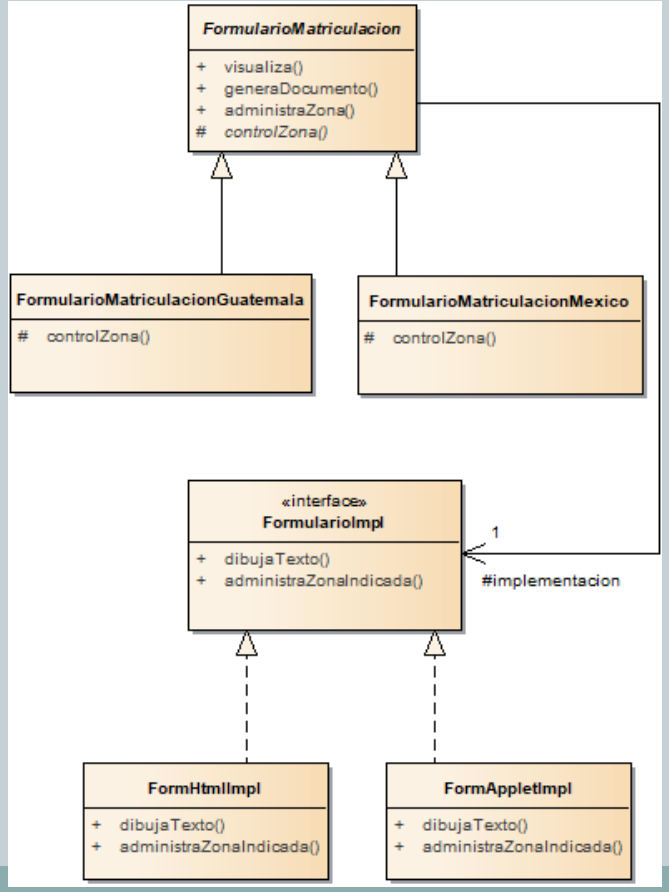
El patrón Bridge, III



- La solución del patrón Bridge consiste en separar aquellos aspectos de representación de los de implementación y en crear dos jerarquías de clases tal y como se ilustra en la figura del acetato 17.

El patrón Bridge, IV

- Las instancias de la clase FormularioMatriculacion mantiene el enlace implementación hacia una instancia que responde a la interfaz FormularioImpl.



El patrón Bridge, V

- La implementación de los métodos de FormularioMatriculacion está basada en el uso de los métodos descritos en FormularioImpl.
- En cuanto a la clase FormularioMatriculacion, ahora es abstracta y existe una sublase concreta para cada país (FormularioMatriculacionMexico y FormularioMatriculacionGuatemala).

Estructura

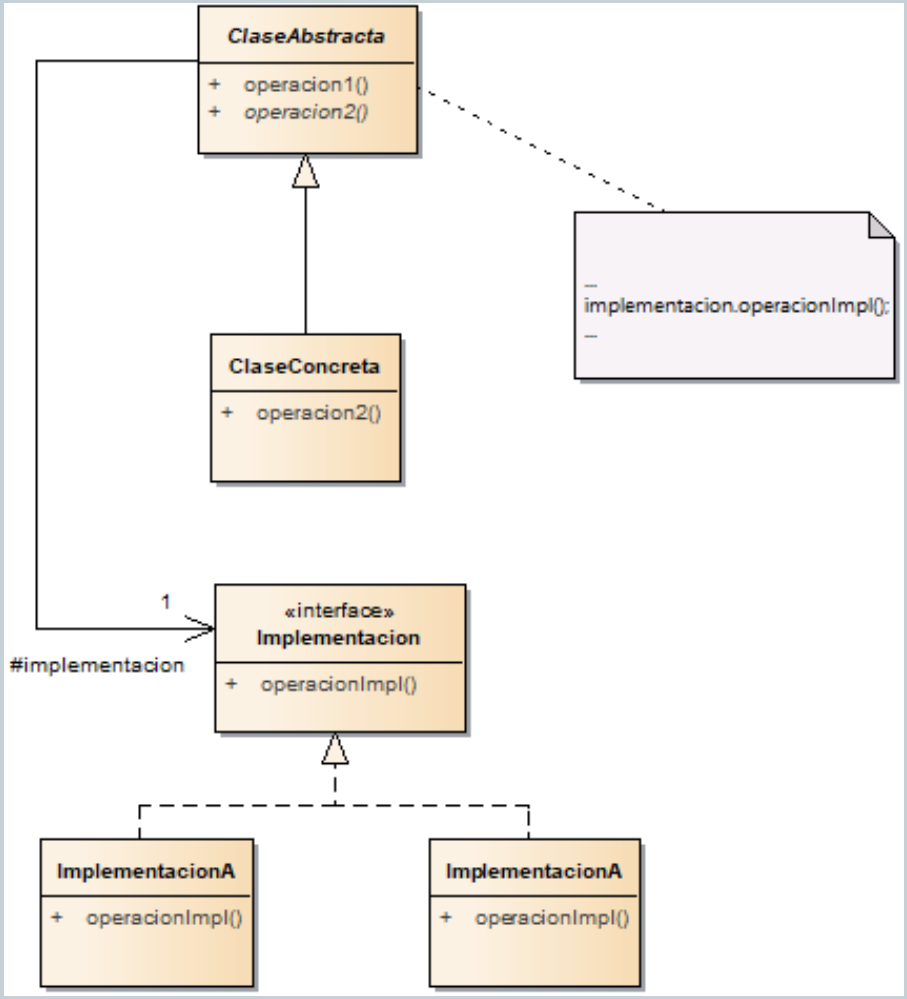
1. diagrama de clases

- La figura del acetato 19 detalla la estructura genérica del patrón.

2. Participantes

- Los participantes del patrón son los siguientes:
  - ClaseAbstracta (FormularioMatriculacion) es la clase abstracta que representa los objetos de dominio. Mantiene la interfaz para los clientes y contiene una referencia hacia un objeto que responde a la interfaz Implementación.
  - ClaseConcreta (FormularioMatriculacionMexico y FormularioMatriculacionGuatemala) es la clase concreta que implementa los métodos de ClaseAbstracta.

El patrón Bridge, VI



El patrón Bridge, VII

- Implementación (FormularioImpl) define la interfaz de las clases de implementación. Los métodos de esta interfaz no deben corresponder con los métodos de ClaseAbstracta. Ambos conjuntos de métodos son diferentes. La implementación incluye por lo general métodos de bajo nivel y los métodos de ClaseAbstracta son de alto nivel.
- ImplementaciónA, ImplementaciónB (FormHtmlImpl, FormAppletImpl) son clases concretas que realizan los métodos incluidos en la interfaz Implementación.

3. Colaboraciones

- Las operaciones de ClaseAbstracta y de sus subclases invocan a los métodos incluidos en la interfaz Implementación.

Dominios de aplicación

El patrón se utiliza en los siguientes casos:

- Para evitar que exista un vínculo demasiado fuerte entre la representación de los objetos y su implementación, en especial cuando la implementación se selecciona en el curso de ejecución de la aplicación.
- Para que los cambios en la implementación de los objetos no tengan impacto en las interacciones entre los objetos y sus clientes.

El patrón Bridge, VIII

- Para permitir a la representación de los objetos y a su implementación conservar su capacidad de extensión mediante la creación de nuevas subclases.
- Para evitar obtener jerarquías de clases demasiado complejas como se ilustra en la figura del acetato 16

Ejemplo en Java

- A continuación presentamos un ejemplo escrito en Java basado en el diagrama de clases de la figura del acetato 17.
- Comenzamos por la interfaz que describe la implementación de los formularios que contienen dos métodos, uno para visualizar un texto y otro para administrar una zona concreta.

El patrón Composite, I

Descripción

- El objetivo del patrón Composite es ofrecer un marco de diseño de una composición de objetos de profundidad variable, diseño que estará basado en un árbol.
- Por otro lado, esta composición está encapsulada respecto a los clientes de los objetos que pueden interactuar sin tener que conocer la profundidad de la composición.

Ejemplo

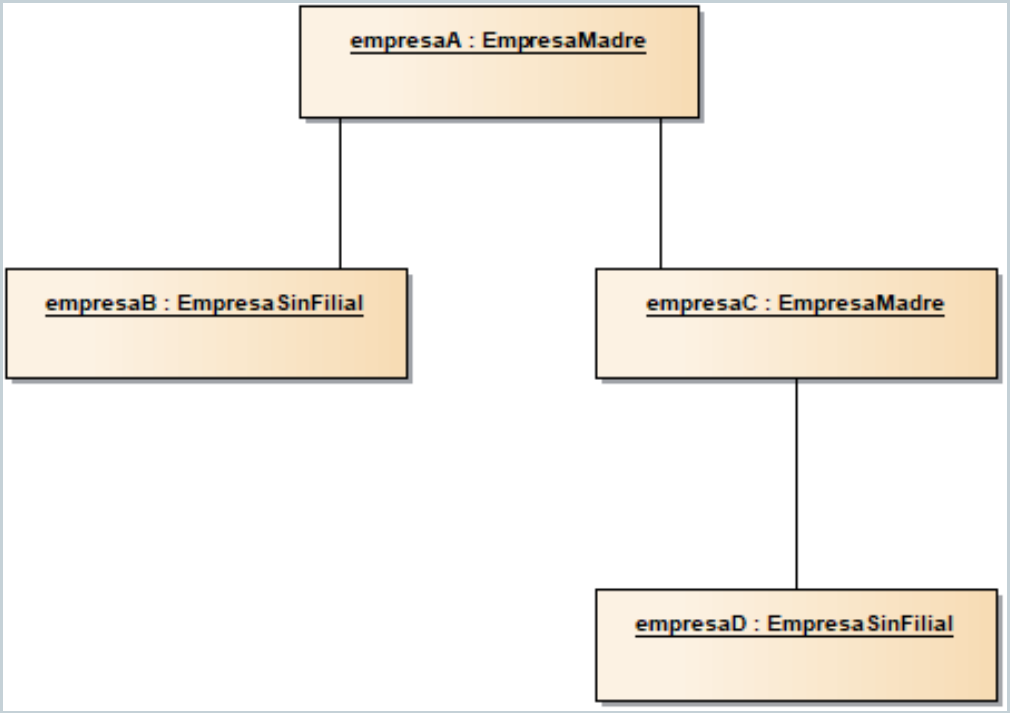
- En nuestro sistema de venta de vehículos, queremos representar las empresas cliente, en especial para conocer el número de vehículos de los que disponen y proporcionarles ofertas de mantenimiento para su parque de vehículos.
- Las empresas que posean filiales solicitan ofertas de mantenimiento que tengan en cuenta el parque de vehículos de sus filiales.
- Una solución inmediata consiste en procesar de forma diferente las empresas sin filiales y las que posean filiales. No obstante esta diferencia en el proceso entre ambos tipos de empresa vuelve a la aplicación más compleja y dependiente de la composición interna de las empresas cliente.

El patrón Composite, II

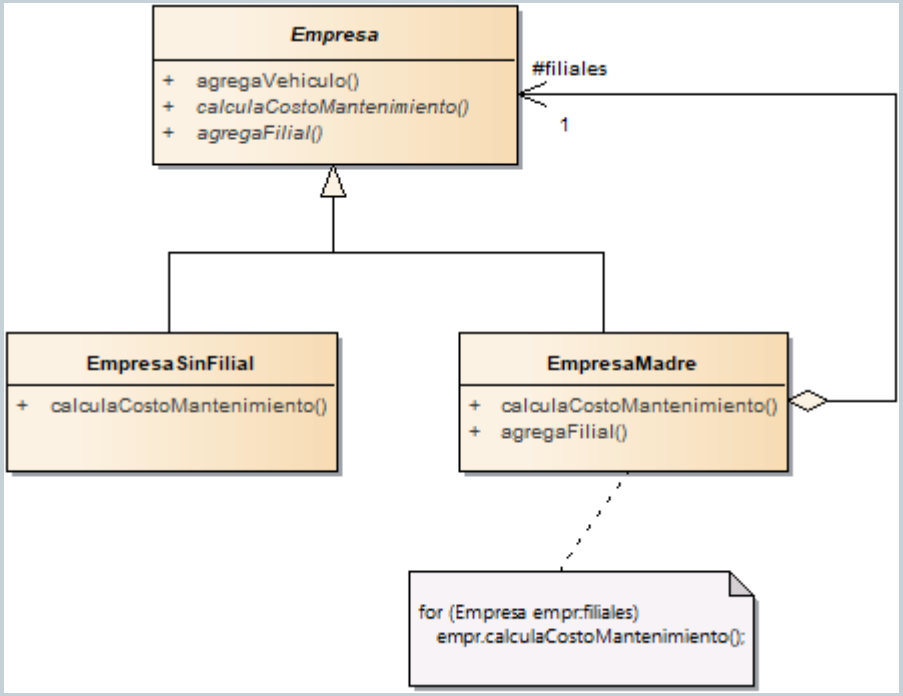
- El patrón Composite resuelve este problema unificando ambos tipos de empresa y utilizando la composición recursiva. Esta composición recursiva es necesaria puesto que una empresa puede tener filiales que posean, ellas mismas, otras filiales.
- Se trata de una composición en árbol (tomamos la hipótesis de la ausencia de una filial común entre dos empresas) tal y como se muestra en la figura del acetato 24 donde las empresas madre se sitúan sobre sus filiales.
- La figura del acetato 25 presenta el diagrama de clases correspondiente. La clase abstracta Empresa contiene la interfaz destinada a los clientes. Posee dos subclases concretas, a saber EmpresaSinFilial y EmpresaMadre, esta última guarda una relación de agregación con la clase Empresa representando los enlaces con sus filiales.
- La clase Empresa posee tres métodos públicos de los cuales sólo uno es concreto y los otros dos son abstractos. El método concreto es el método agregaVehículo que no depende de la composición en filiales de la empresa.

El patrón Composite, III

- En cuanto a los otros dos métodos, se implementan en las subclases concretas (agregaFilial sólo tiene una implementación vacía en EmpresaSinFilial y por lo tanto no se representa en el diagrama de clases).



El patrón Composite, IV



Estructura

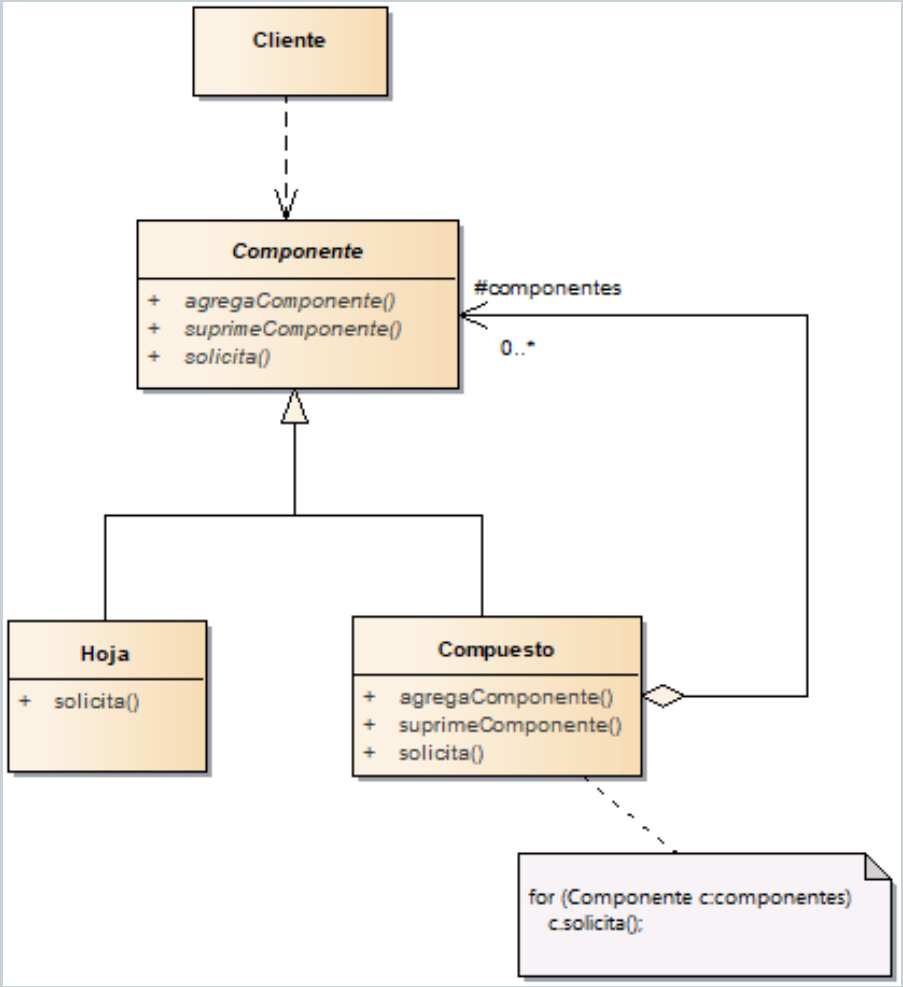
1. Diagrama de clases

- La figura del acetato 26 detalla la estructura genérica del patrón.

2. Participantes

- Los participantes del patrón son los siguientes:

El patrón Composite, V



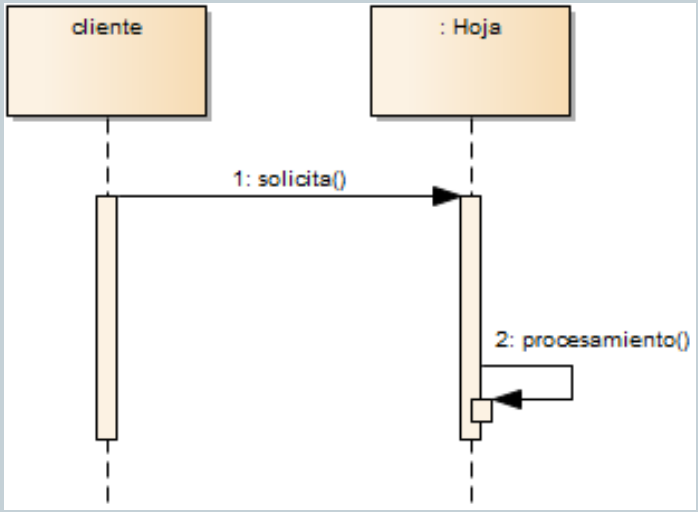
El patrón Composite, VI

- Componente (Empresa) es la clase abstracta que contiene la interfaz de los objetos de la composición, implementa los métodos comunes e introduce la firma de los métodos que gestionan la composición agregando o suprimiendo componentes.
- Hoja (EmpresaSinFilial) es la clase concreta que describe las hojas de la composición (una hoja no posee componentes).
- Compuesto (EmpresaMadre) es la clase concreta que describe los objetos compuestos de la jerarquía. Esta clase posee una asociación de agregación con la clase Componente.
- Cliente es la clase de los objetos que acceden a los objetos de la composición y que los manipulan.

3. Colaboraciones

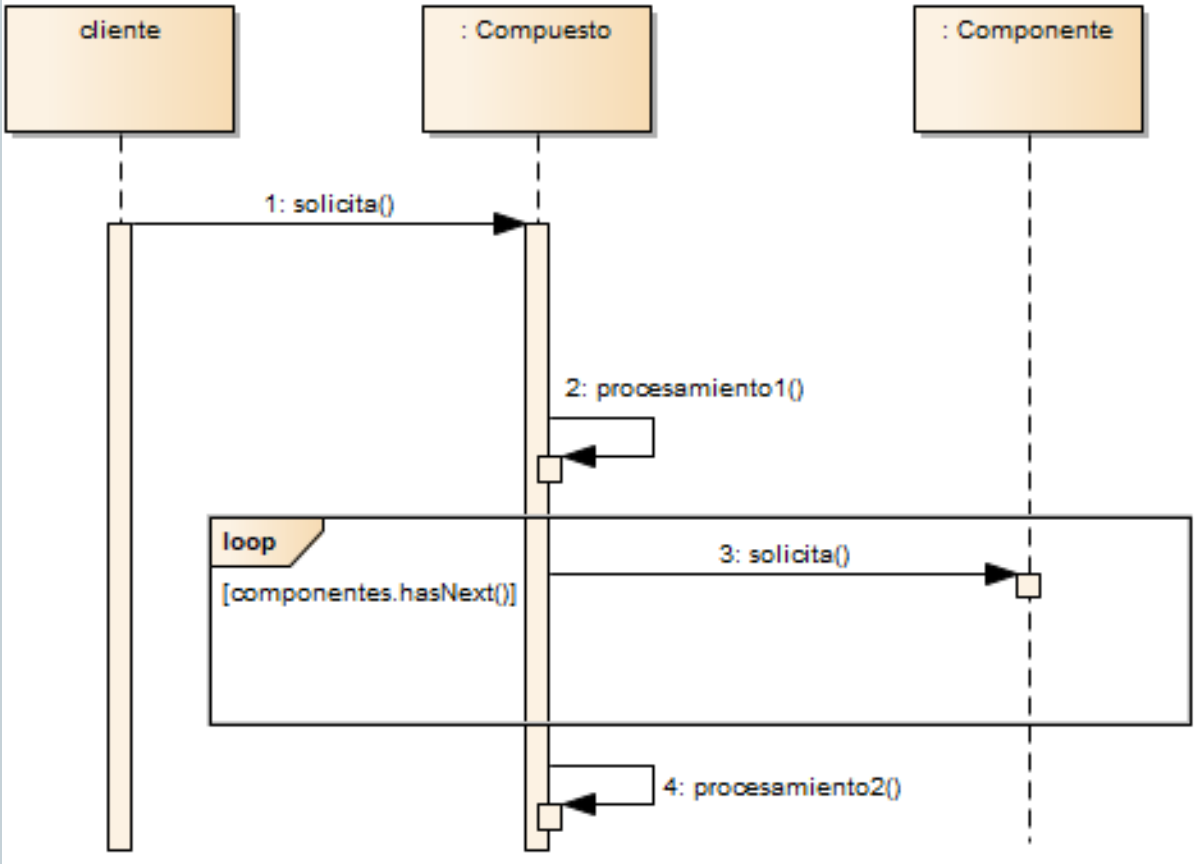
- Los clientes envían sus peticiones a los componentes a través de la interfaz de la clase Componente.
- Cuando un componente recibe una petición, reacciona en función de su clase. Si el componente es una hoja, procesa la petición tal y como se muestra en la figura del acetato 28.

El patrón Composite, VII



- Si el componente es una instancia de la clase Compuesto, realiza un procesado previo, generalmente envía un mensaje a cada uno de sus componentes y realiza un procesamiento posterior.
- La figura del acetato 29 muestra este comportamiento de llamada recursiva a otros componentes que van a procesar, en su turno, esta petición ya sea como hoja o ya sea como compuesto.

El patrón Composite, VIII



El patrón Composite, IX

Dominios de aplicación

- El patrón se usa en los siguientes casos:
  - Es necesario representar jerarquías de composición en un sistema.
  - Los clientes de una composición deben ignorar si se comunican con objetos compuestos o no.

Ejemplo en Java

- Retomamos el ejemplo de las empresas y la gestión de su parque de vehículos.
- El código fuente en Java de la clase abstracta Empresa aparece a continuación.
- Conviene observar que el método agregaFilial reenvía un resultado booleano que indica si ha sido posible realizar o no la agregación.
- Revisar la carpeta del patrón Composite.

El patrón Decorator, I

Descripción

- El objetivo del patrón Decorator es agregar dinámicamente funcionalidades suplementarias a un objeto. Esta agregación de funcionalidades no modifica la interfaz del objeto y es transparente de cara a los clientes.
- El patrón Decorator constituye una alternativa respecto a la creación de una subclase para enriquecer el objeto.

Ejemplo

- El sistema de venta de vehículos dispone de una clase VistaCatalogo que muestra, bajo el formato de un catalogo electrónico, los vehículos disponibles en una página web.
- Queremos, a continuación, visualizar datos suplementarios para los vehículos de "alta gama", a saber la información técnica ligada al modelo.
- Para agregar esta funcionalidad, podemos crear una subclase de visualización específica para los vehículos de "alta gama". Ahora, queremos mostrar el logotipo de la marca de los vehículos de "gamas media y alta".

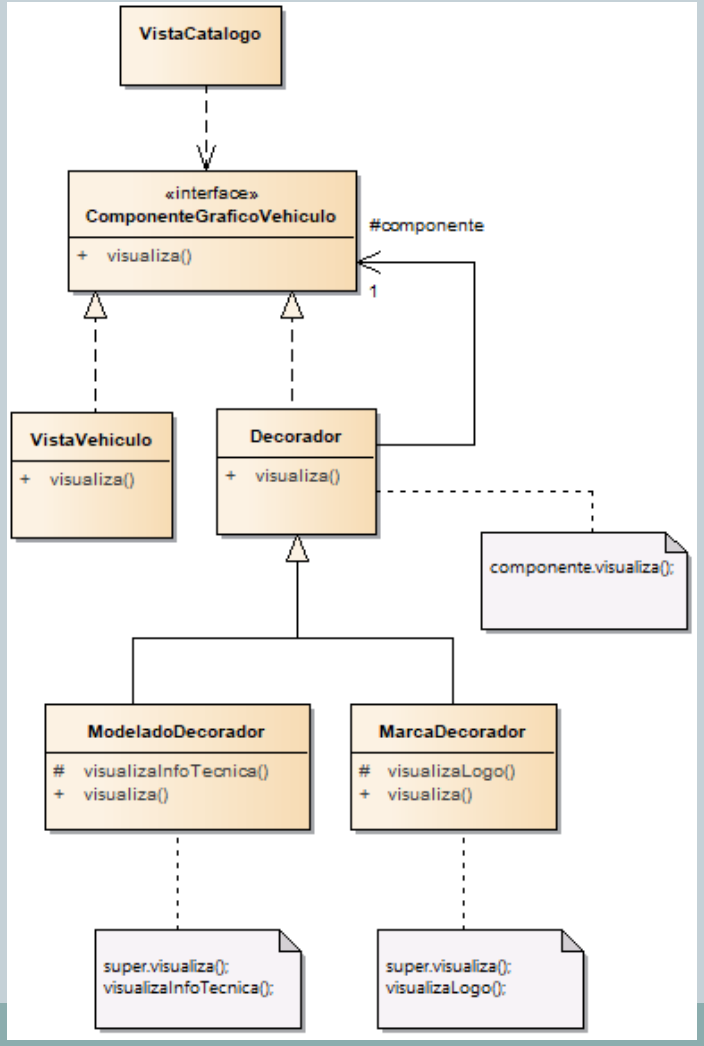
El patrón Decorator, II

- Conviene crear una nueva subclase para estos vehículos, superclase de la clase vehículos de “alta gama”, lo cual se vuelve rápidamente complejo.
- Es fácil darse cuenta de que la herencia no está adaptada a lo que se demanda por dos razones:
  - La herencia es una herramienta demasiado potente para agregar esta funcionalidad.
  - La herencia es un mecanismo estático.
- El patrón Decorator proporciona otro enfoque que consiste en agregar un nuevo objeto llamado decorador que se sustituye por el objeto inicial y que lo referencia.
- Este decorador posee la misma interfaz, lo cual vuelve a la sustitución transparente de cara a los clientes.
- En nuestro caso, el método visualiza lo intercepta el decorador que solicita al objeto inicial su visualización y a continuación la enriquece con información complementaria.
- La figura del acetato 34 ilustra el uso del patrón Decorator para enriquecer la visualización de vehículos.

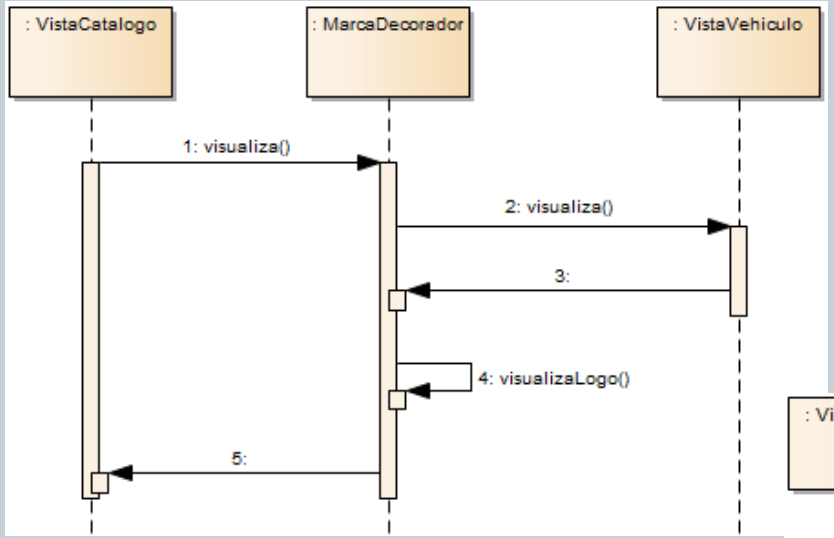
El patrón Decorator, III

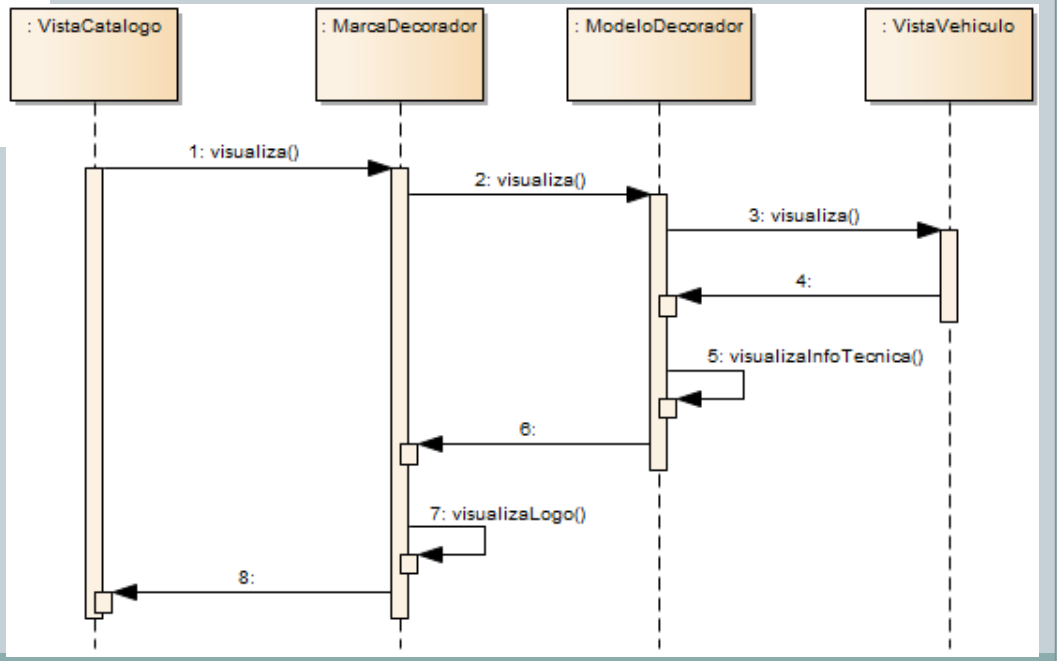
- La interfaz ComponenteGraficoVehiculo constituye la interfaz común a la clase VistaVehiculo, que queremos enriquecer, y a la clase abstracta Decorador, interfaz constituida únicamente por el método visualiza.
- La clase Decorador posee una referencia hacia un componente gráfico. Esta referencia la utiliza el método visualiza que delega la visualización en este componente.
- Existen dos clases concretas de decorador, subclases de Decorador. Su método visualiza empieza llamando al método visualiza de Decorador y a continuación muestra los datos complementarios tales como la información técnica del vehículo o el logotipo de la marca.
- La figura del acetato 35 muestra la secuencia de llamadas de mensaje destinadas a la visualización de un vehículo para el cual se tiene que mostrar el logo de la marca.
- La figura de abajo del acetato 35 muestra la secuencia de llamadas de mensaje destinadas a la visualización de un vehículo para el cual se tiene que mostrar la información técnica del modelo y el logotipo de su marca.

El patrón Decorator, IV



El patrón Decorator, V





El patrón Decorator, VI

- Esta figura ilustra bien el hecho de que los decoradores son componentes puesto que pueden transformar el componente de un nuevo decorador, lo cual da lugar a una cadena de decoradores.
- Esta posibilidad de encadenar componentes en la que es posible agregar o eliminar dinámicamente un decorador proporciona una gran flexibilidad

Estructura

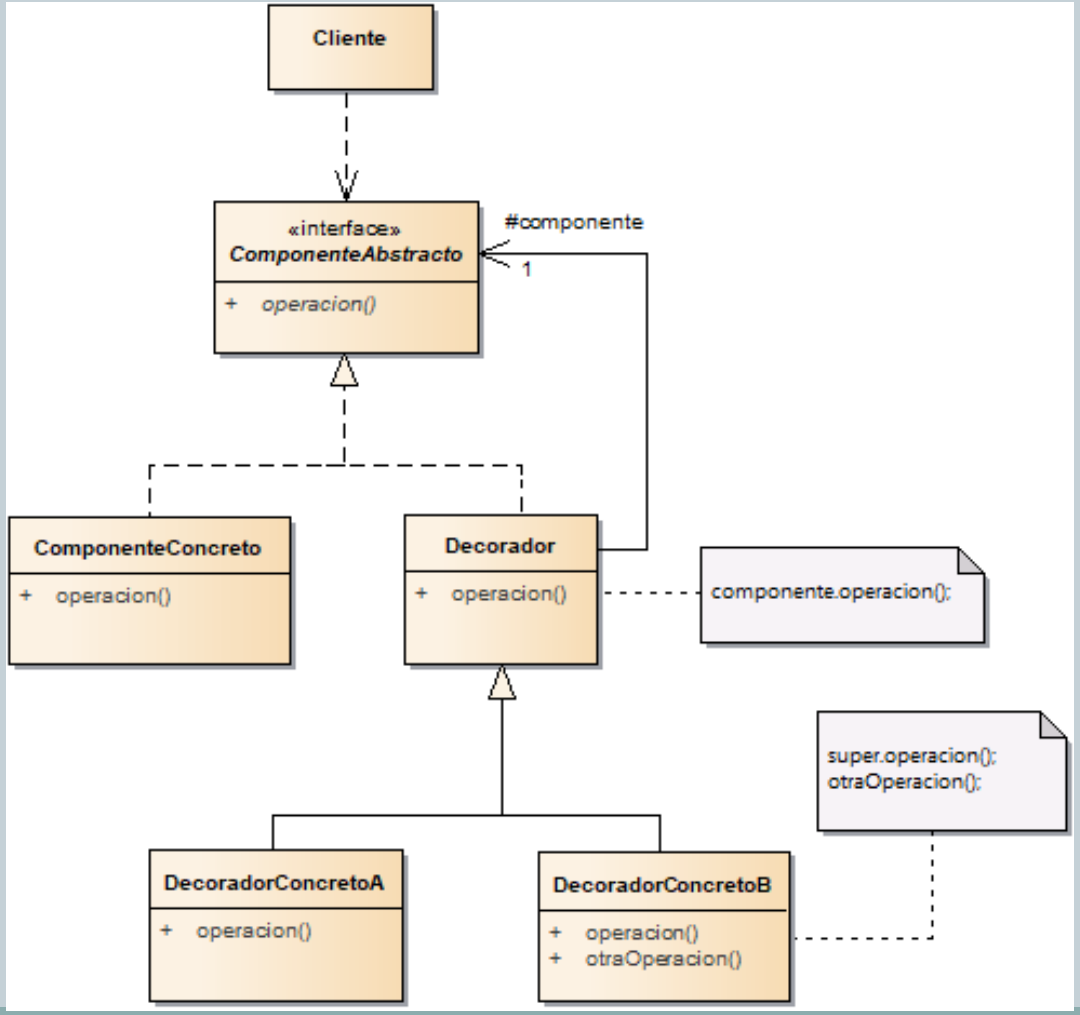
1. Diagrama de clases

- La figura del acetato 37 detalla la estructura genérica del patrón.

2. Participantes

- Los participantes del patrón son los siguientes:
  - ComponenteAbstracto (ComponenteGraficoVehiculo) es la interfaz común al componente y a los decoradores.
  - ComponenteConcreto (VistaVehiculo) es el objeto inicial al que se deben agregar las nuevas funcionalidades.

El patrón Decorator, VII



El patrón Decorator, VIII

- Decorador es una clase abstracta que guarda una referencia hacia el componente.
- DecoradorConcretoA y DecoradorConcretoB (ModeloDecorador y MarcaDecorador) son subclases concretas de Decorador que tienen como objetivo implementar las funcionalidades agregadas al componente.

3. Colaboraciones

- El decorador se sustituye por el componente. Cuando recibe un mensaje destinado a este último, lo redirige al componente realizando operaciones previas o posteriores a esta redirección.

Dominios de aplicación

- El patrón Decorator puede utilizarse en los siguientes dominios:
  - Un sistema agrega dinámicamente funcionalidades a un objeto, sin modificar su interfaz, es decir sin que los clientes de este objeto tengan que verse modificados.
  - Un sistema gestiona funcionalidades que pueden eliminarse dinámicamente.
  - El uso de la herencia para extender los objetos no es práctico, lo cual puede ocurrir cuando su jerarquía ya es de por si compleja.

El patrón Decorator, IX

Ejemplo en Java

- Presentamos a continuación el código fuente Java del ejemplo, comenzando por la interfaz ComponenteGraficoVehiculo.
- Ver la carpeta del patrón Decorator.

El patrón Facade, I

Descripción

- El objetivo del patrón Facade es agrupar las interfaces de un conjunto de objetos en una interfaz unificada volviendo a este conjunto más fácil de usar por parte de un cliente.
- El patrón Facade encapsula la interfaz de cada objeto considerada como interfaz de bajo nivel en una interfaz única de nivel más elevado. La construcción de la interfaz unificada puede necesitar implementar métodos destinados a componer las interfaces de bajo nivel.

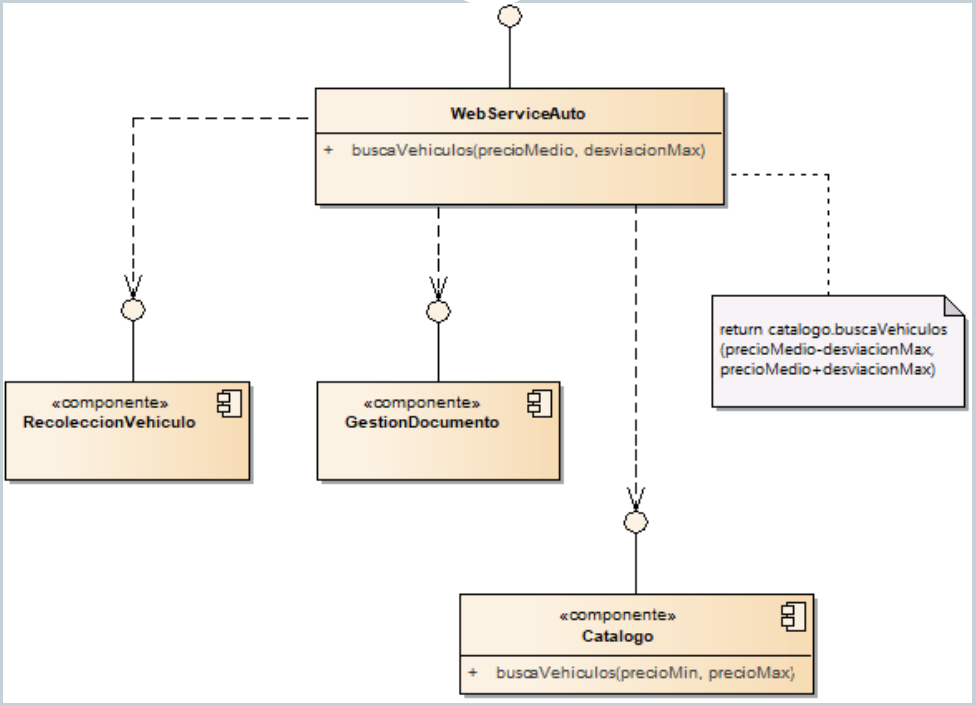
Ejemplo

- Queremos ofrecer la posibilidad de acceder al sistema de venta de vehículos como servicio web. El sistema está modelado como una arquitectura bajo la forma de un conjunto de componentes que poseen su propia interfaz como:
  - El componente Catalogo.
  - El componente GestionDocumento.
  - El componente RecuperarVehiculo.

El patrón Facade, II

- Es posible dar acceso al conjunto de la interfaz de estos componentes a los clientes del servicio web, aunque esta posibilidad presenta dos inconvenientes principales:
  - Algunas funcionalidades no las utilizan los clientes del servicio web, como por ejemplo las funcionalidades de visualización del catalogo.
  - La arquitectura interna del sistema responde a las exigencias de modularidad y evolución que no forman parte de las necesidades de los clientes del servicio web, para los que estas exigencias suponen una complejidad inútil.
- El patrón Facade resuelve este problema proporcionando una interfaz unificada más sencilla y con un nivel de abstracción más elevado.
- Una clase se encarga de implementar esta interfaz unificada utilizando los componentes del sistema.
- Esta solución se ilustra en la figura del acetato 42.
- La clase WebServiceAuto ofrece una interfaz a los clientes del servicio web. Esta clase y su interfaz constituyen una fachada de cara a los clientes.

El patrón Facade, III



- La interfaz de la clase WebServiceAuto está constituida por el método buscaVehiculos (precioMedio, desviacionMax) cuyo código consiste en invocar al método buscaVehiculos(precioMin, precioMax) del catálogo adaptando el valor de los argumentos de este método en función del precio medio y de la desviación máxima.

El patrón Facade, IV

- Conviene observar que si bien la idea del patrón es construir una interfaz de más alto nivel de abstracción, nada nos impide proporcionar en la fachada accesos directos a ciertos métodos de los componentes del sistema.

Estructura

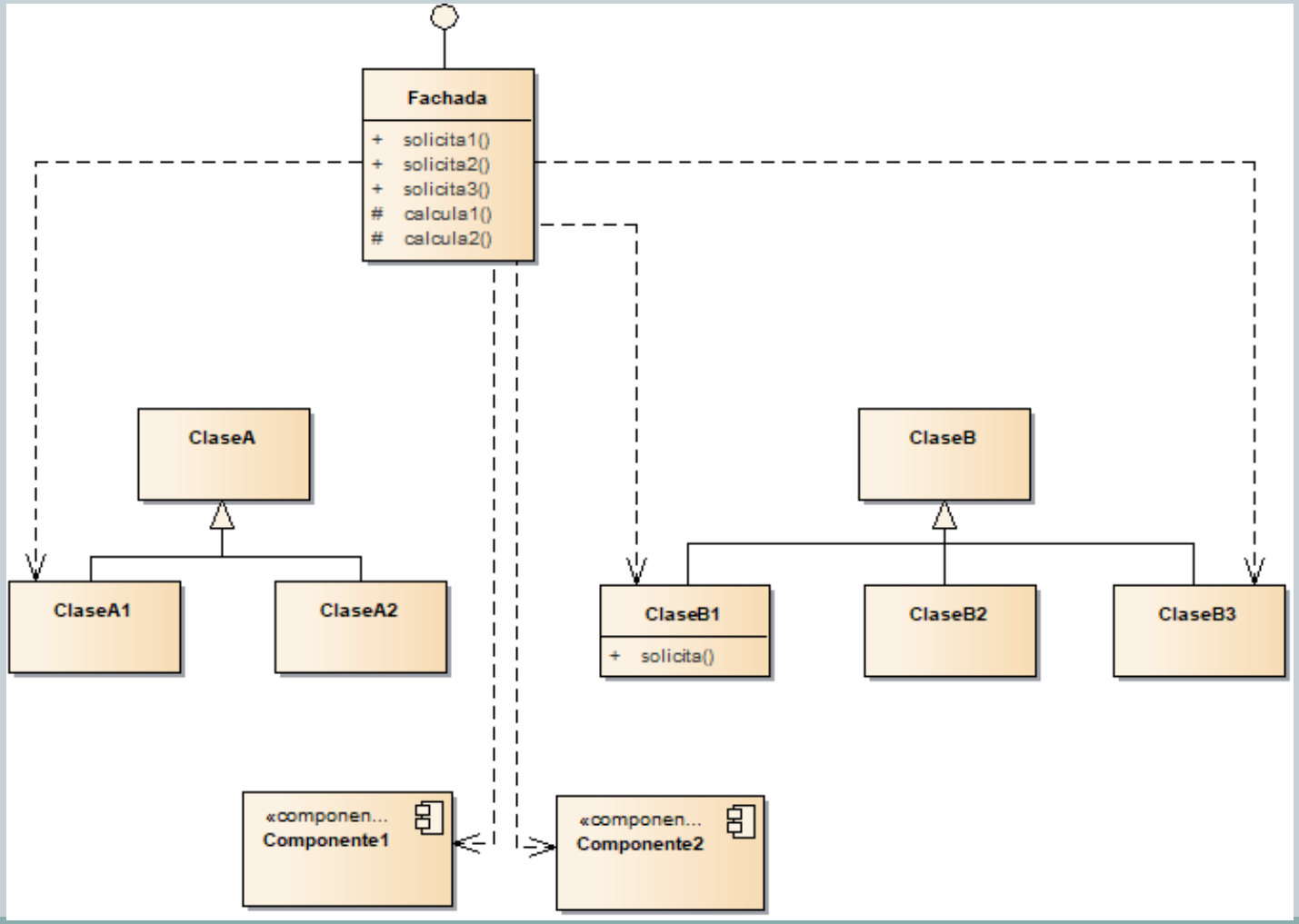
1. Diagrama de clases

- La figura del acetato 44 detalla la estructura genética del patrón.

2. Participantes

- Los participantes del patrón son los siguientes:
  - Facade (WebServiceAuto) y su interfaz constituyen la parte abstracta expuesta a los clientes del sistema. Esta clase posee referencias hacia las clases y componentes que forman el sistema y cuyos métodos se utilizan en la fachada para implementar la interfaz unificada.
  - Las clases y componentes del sistema (RecuperarVehiculo. GestionDocumento y Catalogo) implementan las funcionalidades del sistema y responden a las consultas de la fachada. No necesitan a la fachada para trabajar.

El patrón Facade, V



El patrón Facade, VI

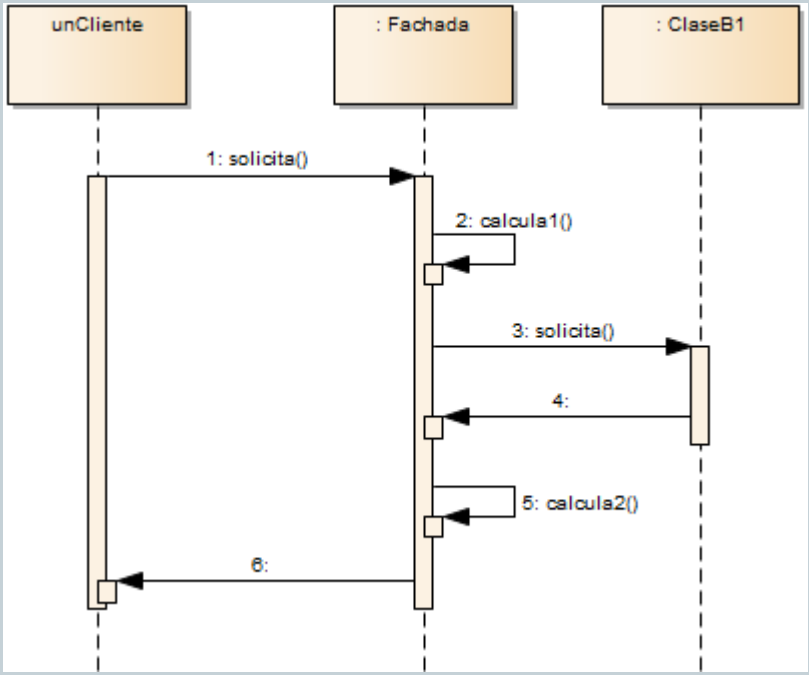
3. Colaboraciones

- Los clientes se comunican con el sistema a través de la fachada que se encarga, de forma interna, de invocar a las clases y los componentes del sistema.
- La fachada no puede limitarse a transmitir las invocaciones.
- También debe realizar la adaptación entre su interfaz y la interfaz de los objetos del sistema mediante código específico.
- El diagrama de secuencia de la figura del acetato 46 ilustra esta adaptación para un ejemplo cuyo código especifico a la fachada debe invocarse (métodos calcula1 y calcula2).
- Nota: los clientes que utilizan la fachada no deben acceder directamente a los objetos del sistema.

Dominios de aplicación

- El patrón se utiliza en los siguientes casos:

El patrón Facade, VII



- Para proveer una interfaz simple de un sistema complejo. La arquitectura de un sistema puede estar basada en numerosas clases pequeñas, que ofrecen una buena modularidad y capacidad de evolución. No obstante estas propiedades tan estupendas no interesan en absoluto a los clientes, que sólo necesitan un acceso simple que responda a sus exigencias.

El patrón Facade, VIII

- Para dividir un sistema en subsistemas, la comunicación entre subsistemas se define de forma abstracta a su implementación gracias a las fachadas.
- Para sistematizar la encapsulación de la implementación de un sistema de cara al exterior.

Ejemplo en Java

- Retomamos el ejemplo del servicio web que vamos a simular con ayuda de un pequeño programa escrito en Java. Se muestra en primer lugar el código fuente de los componentes del sistema, comenzando por la clase ComponenteCatalogo y su interfaz Catalogo, así como la clase DescripcionVehiculo.
- La base de datos que constituye el catálogo se reemplaza por una sencilla tabla de objetos de la clase DescripcionVehiculo.
- El método buscaVehiculo realiza la búsqueda de uno o varios vehículos en función de su precio gracias a un simple ciclo.
- Ver la carpeta del respectivo patrón.

El patrón Flyweight, I

Descripción

- El objetivo del patrón Flyweight es compartir de forma eficaz un conjunto de objetos de granularidad fina.

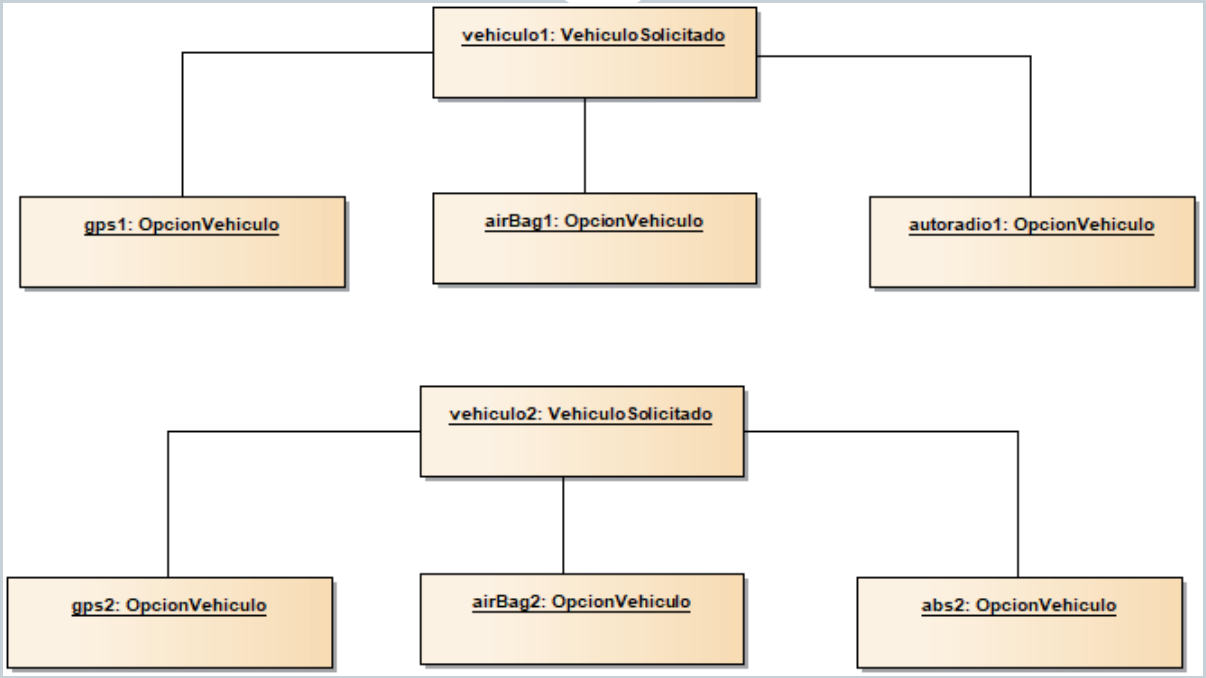
Ejemplo

- En el sistema de venta de vehículos, es necesario administrar las opciones que el comprador puede elegir cuando está comprando un nuevo vehículo.
- Estas opciones están descritas por la clase OpcionVehiculo que contiene varios atributos tales como el nombre, la explicación, un logotipo, el precio estándar, las incompatibilidades con otras opciones, con ciertos modelos, etc.
- Por cada vehículo solicitado, es posible asociar una nueva instancia de esta clase. No obstante a menudo existe un gran número de opciones para cada vehículo solicitado, lo cual obliga al sistema a gestionar un conjunto enorme de objetos de tamaño pequeño (de granularidad fina).

El patrón Flyweight, II

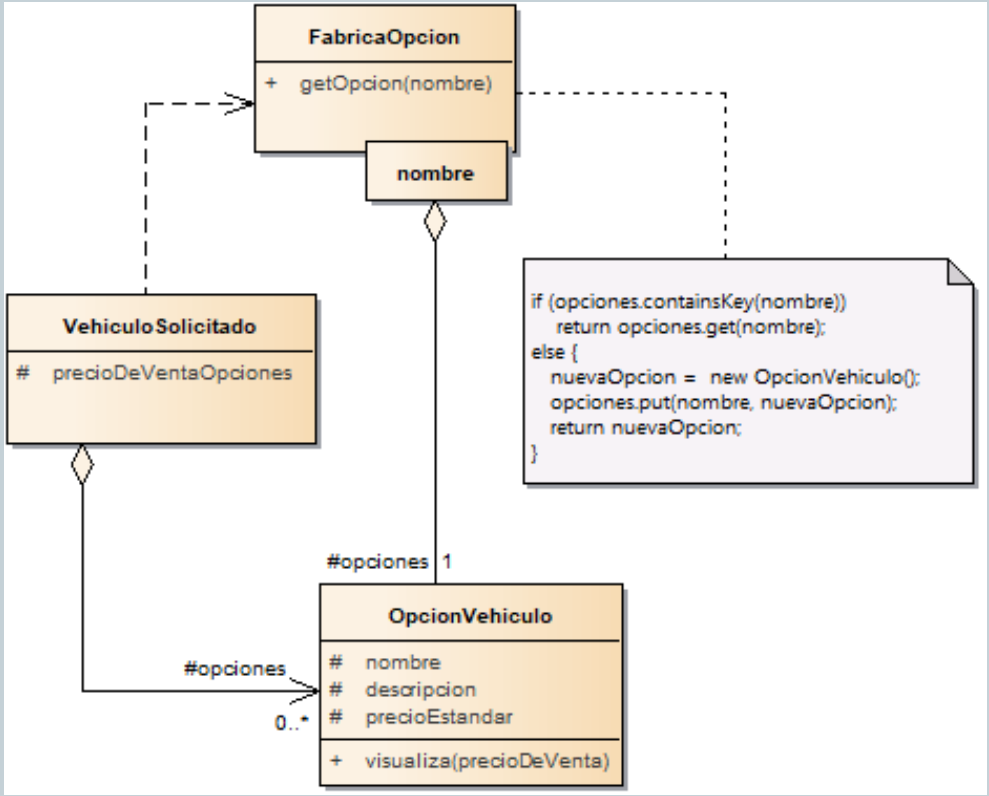
- Este enfoque presenta sin embargo la ventaja de poder almacenar la nivel de opción la información específica a sí misma y al vehículo, como por ejemplo el precio de venta de la opción que puede diferir de un vehículo a otro.
- Esta solución se presenta con un pequeño ejemplo de la figura del acetato 50 y es fácil darse cuenta de que es necesario gestionar un gran número de instancias de OpcionVehiculo mientras que entre ellas contienen datos idénticos.
- El patrón Flyweight proporciona una solución a este problema compartiendo las opciones:
  - La compartición se realiza mediante una fábrica a la que el sistema se dirige para obtener una referencia hacia una opción. Si esta opción no se ha creado hasta ahora, la fábrica procede a su creación antes de enviar la referencia.
  - Los atributos de una opción contienen solamente su información específica independientemente de los vehículos solicitados: esta información constituye el **estado intrínseco** de las opciones.
  - La información particular a una opción y a un vehículo se almacena a nivel de vehículo: esta información constituye el **estado extrínseco** de las opciones. Se pasan como parámetros en las llamadas a los métodos de las opciones.

El patrón Flyweight, III



- Nota: En el marco de este patrón, las opciones son los objetos llamados flyweights (peso mosca).
- La figura del acetato 51 ilustra el diagrama de clases de esta solución.
- Este diagrama de clases incluye las siguientes clases:

El patrón Flyweight, IV



- OpcionVehiculo cuyos atributos contienen el estado intrínseco de una opción. El método visualiza recibe como parámetro el precioDeVenta que representa el estado extrínseco de una opción.

El patrón Flyweight, V

- FabricaOpcion cuyo método getOpcion reenvía una opción a partir de su nombre. Su funcionamiento consiste en buscar la opción en la asociación cualificada y en crearla en caso contrario.
- vehiculoSolicitado que posee una lista de las opciones seleccionadas así como su precio de venta.

Estructura

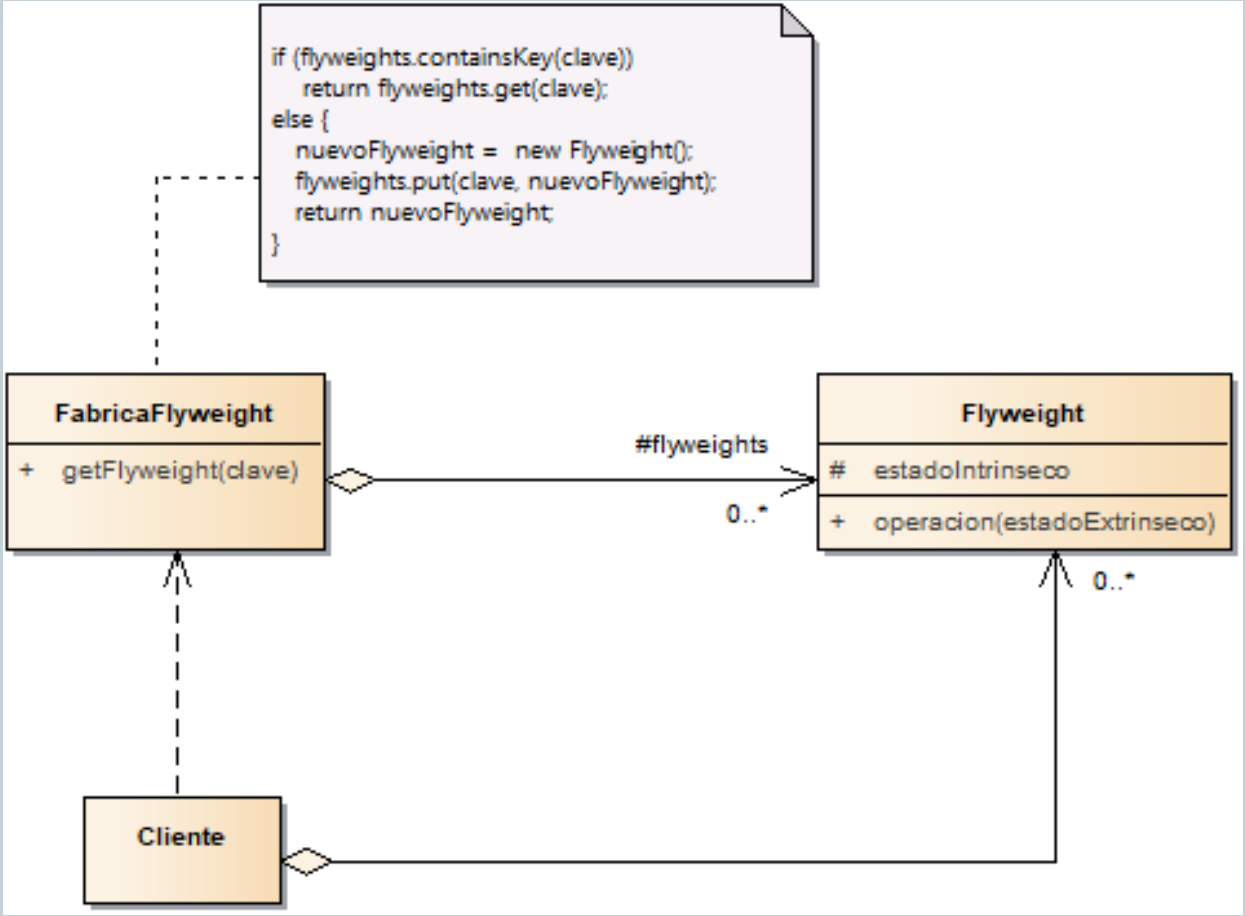
1. Diagrama de clases

- La figura del acetato 53 detalla la estructura genérica del patrón.

2. Participantes

- Los participantes del patrón son los siguientes:
  - FabricaFlyweight (FabricaOpcion) crea y administra los flyweights. La fábrica se asegura de que los flyweights se comparten gracias al método getFlyweight que devuelve la referencia hacia los flyweights.
  - Flyweight (OpcionVehiculo) mantiene el estado intrínseco e implementa los métodos. Estos métodos reciben y determinan a su vez el estado extrínseco de los flyweights.
  - Cliente (VehiculoSolicitado) contiene un conjunto de referencias hacia los flyweights que utiliza. El cliente debe a su vez guardar el estado extrínseco de estos flyweights.

El patrón Flyweight, VI



El patrón Flyweight, VII

3. Colaboraciones

- Los clientes no deben crear ellos mismos los flyweights sino utilizar el método getFlyweight de la clase FabricaFlyweight que garantiza que los flyweights se comparten.
- Cuando un cliente invoca un método de un flyweight, debe transmitirle su estado extrínseco.

Dominio de aplicación

- El dominio de aplicación del patrón Flyweight es la posibilidad de compartir pequeños objetos (pesos mosca).
- Los criterios de uso son los siguientes:
  - El sistema utiliza un gran número de objetos.
  - El almacenamiento de los objetos es costoso porque existe una gran cantidad de objetos.
  - Existen numerosos conjuntos de objetos que pueden reemplazarse por algunos objetos compartido una vez que parte de su estado se vuelve extrínseco.

El patrón Flyweight, VIII

Ejemplo en Java

- La clase OpcionVehiculo posee un constructor que permite definir el estado intrínseco de la opción.
- En este ejemplo, a parte del nombre, los demás atributos toman valores constantes o que están basados directamente en el nombre.
- Normalmente, estos valores deberían provenir de una base de datos.
- El método visualiza recibe el precio de venta como parámetro, que constituye el estado extrínseco.
- Ver la carpeta del patrón.

El patrón Proxy, I

Descripción

- El patrón Proxy tiene como objetivo el diseño de un objeto que sustituye a otro objeto (el sujeto) y que controla el acceso.
- El objeto que realiza la sustitución posee la misma interfaz que el sujeto, volviendo la sustitución transparente de cara a los clientes.

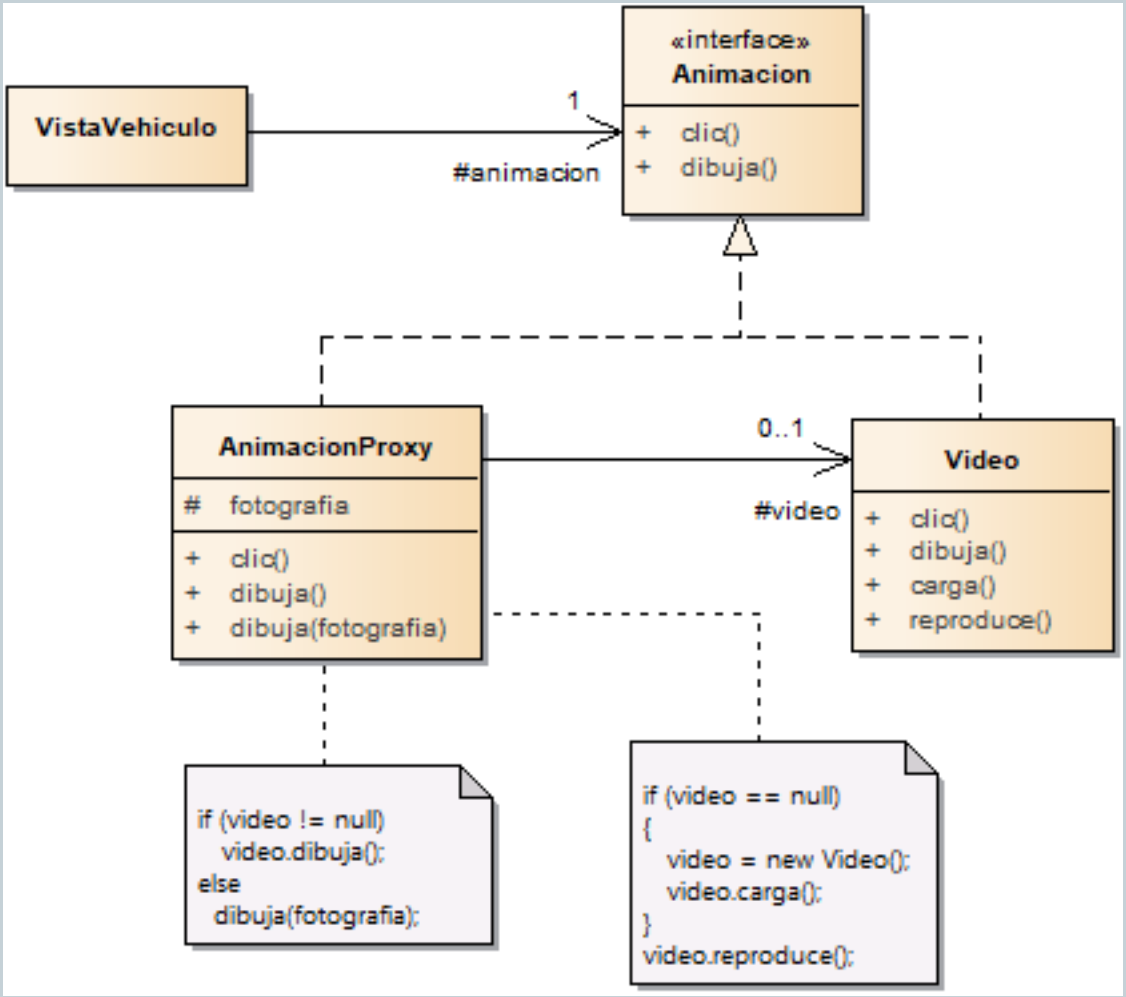
Ejemplo

- Queremos ofrecer para cada vehículo del catálogo la posibilidad de visualizar un pequeño video de presentación del vehículo. Un clic sobre la fotografía de la presentación del vehículo permitirá reproducir este vídeo.
- Una página del catálogo contiene numerosos vehículos y es muy pesado guardar en memoria todos los objetos de animación, pues los vídeos necesitan gran cantidad de memoria, y su transferencia a través de la red toma bastante tiempo.
- El patrón Proxy ofrece una solución a este problema difiriendo la creación de los sujetos hasta el momento en que el sistema tiene necesidad de ellos, en este caso después de un clic en la fotografía del vehículo.

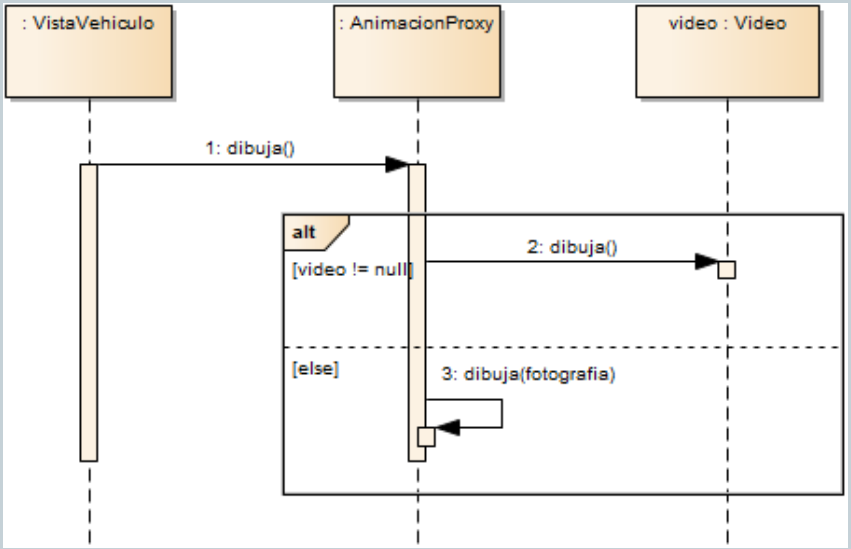
El patrón Proxy, II

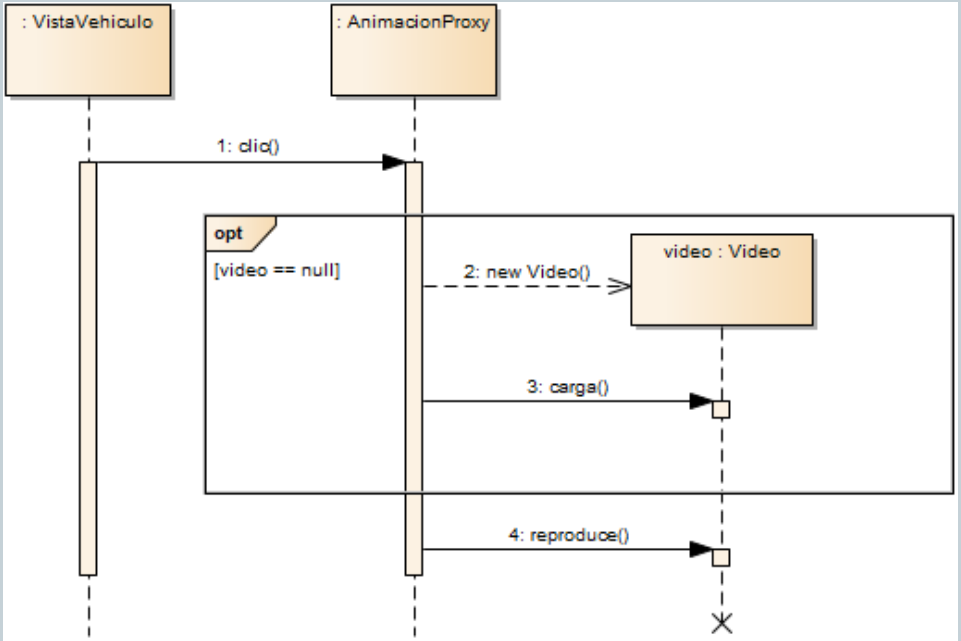
- La solución aporta dos ventajas:
  - La página del catálogo se carga mucho más rápidamente, sobre todo si tiene que cargarse a través de una red como Internet.
  - Sólo aquellos vídeos que van a visualizarse se crean, cargan y reproducen.
- El objeto fotografía se llama el proxy del video. Procede a la creación del sujeto únicamente tras haber hecho un clic en ella. Posee la misma interfaz que el objeto vídeo.
- La figura del acetato 58 muestra el diagrama de clases correspondiente. La clase del proxy, AnimacionProxy, y la clase del video, Video, implementan ambas la misma interfaz, es decir Animación.
- Cuando el proxy recibe el mensaje dibuja, muestra el vídeo si ha sido creado y cargado. Cuando el proxy recibe el mensaje clic, reproduce el vídeo después de haberlo creado y cargado previamente.
- El diagrama de secuencia para el mensaje dibuja se detalla en la figura del acetato 59 arriba y en la figura del acetato 59 abajo para el mensaje clic.

El patrón Proxy, III



El patrón Proxy, IV





El patrón Proxy, V

Estructura

1. Diagrama de clases

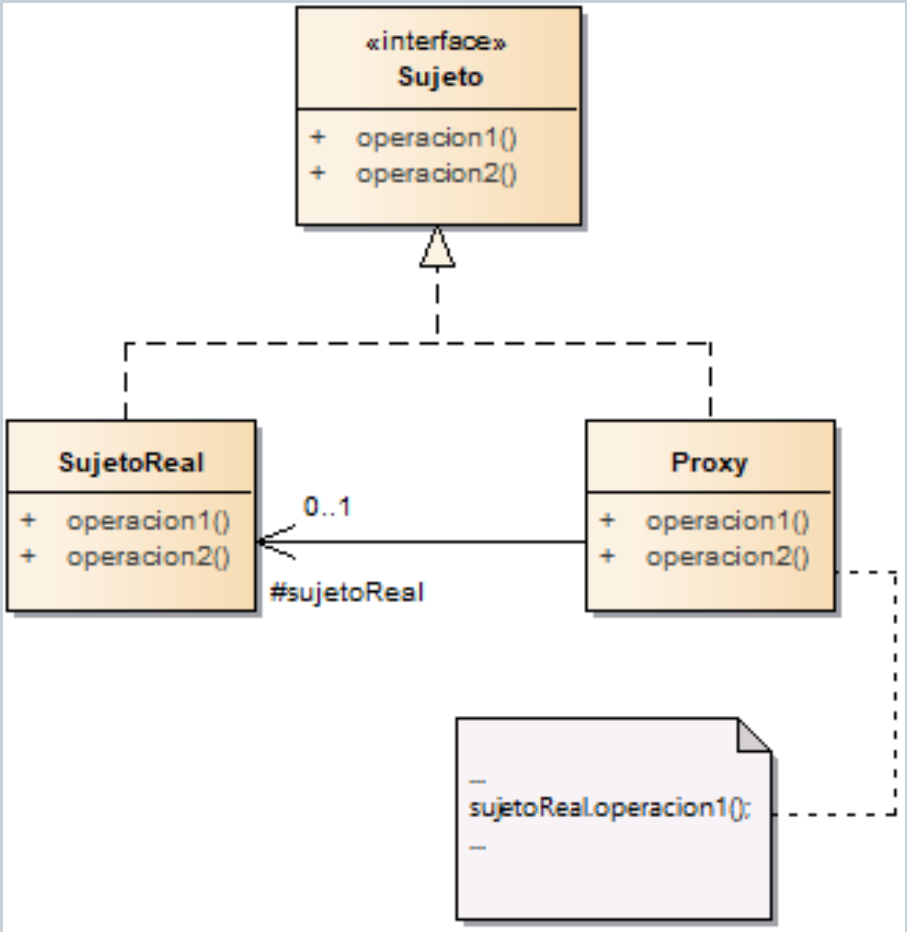
- La figura del acetato 61 ilustra la estructura genérica del patrón.
- Conviene observar que los métodos del proxy tienen dos comportamientos posibles cuando el sujeto real no ha sido creado: o bien crean el sujeto real y a continuación le delegan el mensaje (es el caso del método clic del ejemplo), o bien ejecutan un código de sustitución (es el caso del método dibuja del ejemplo).

2. Participantes

- Los participantes del patrón son los siguientes:
  - Sujeto (Animacion) es la interfaz común al proxy y al sujeto real.
  - SujetoReal (Video) es el objeto que el proxy controla y representa.
  - Proxy (AnimacionProxy) es el objeto que se sustituye por el sujeto real. Posee una interfaz idéntica a este último (interfaz Sujeto). Se encarga de crear y de destruir al sujeto real y de delegarle los mensajes.

3. Colaboraciones

El patrón Proxy, VI



El patrón Proxy, VII

- El proxy recibe las llamadas del cliente en lugar del sujeto real. Cuando lo juzga apropiado, delega estos mensajes en el sujeto real. Debe, en este caso, crear previamente el sujeto real si no está creado ya.

Dominios de aplicación

- Los proxys son muy útiles en programación orientada a objetos. Existen distintos tipos de proxy. Vamos a ilustrar tres:
  - Proxy virtual: permite crear un objeto de tamaño importante en el momento adecuado.
  - Proxy remoto: permite acceder a un objeto ejecutándose en otro entorno. Este tipo de proxy se implementa en sistemas de objetos remotos (CORBA, Java RMI).
  - Proxy de protección: permite securizar el acceso a un objeto, por ejemplo mediante técnicas de autenticación.

Ejemplo en Java

- Retomamos nuestro ejemplo en Java. El código fuente de la interfaz Animacion se puede revisar en la carpeta del patrón.