Composición y variación de Patrones en Java

ABRAHAM SÁNCHEZ LÓPEZ
GRUPO MOVIS
FCC-BUAP

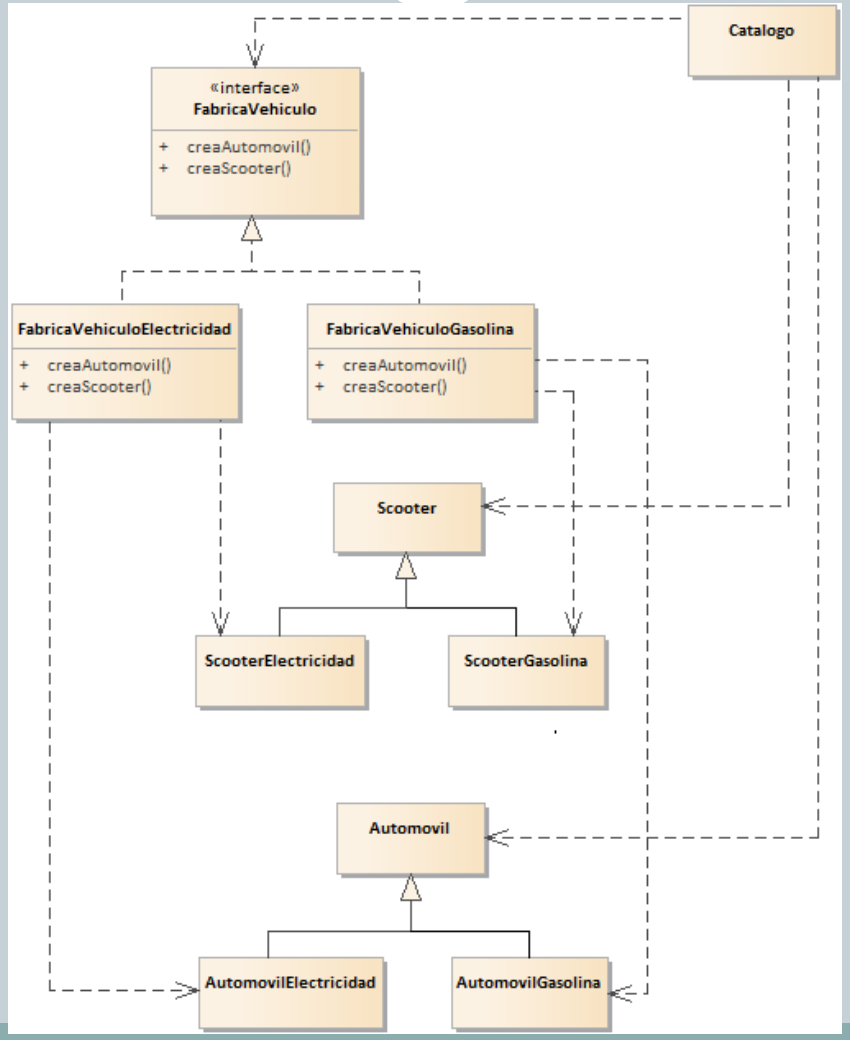
Introducción

- Los veintitrés patrones de diseño presentados anteriormente no constituyen una lista exhaustiva.
- Es posible crear nuevos patrones bien «ex nihilo» (de la nada!), o bien componiendo o adaptando patrones existentes.
- Estos nuevos patrones pueden tener un carácter general, a semejanza de los que hemos presentado en los temas anteriores, o ser específicos a un entorno de desarrollo particular.
- De este modo podemos citar el patrón de diseño de los JavaBeans o los patrones de diseño de los EJB.
- En esta parte del curso, vamos a mostrar tres nuevos patrones obtenidos mediante la composición y variación de patrones existentes.

El patrón Pluggable Factory, I

- Hemos presentado en un tema anterior el patrón Abstract Factory que permite abstraer la creación (instanciación) de productos de sus distintas familias. En este caso se crea una fabrica asociada a cada familia de productos.
- En el diagrama de la figura del acetato 4, se exponen dos productos: automóviles y scooters, descritos cada uno mediante una clase abstracta.
- Estos productos se organizan en dos familias: gasolina o electricidad. Cada una de las dos familias engendra una subclase concreta de cada clase de producto.
- Existen por lo tanto dos fábricas para las familias FabricaVehiculoGasolina y FabricaVehiculoElectricidad. Cada fabrica permite crear uno de los dos productos mediante los métodos apropiados.
- Este patrón organiza de forma muy estructurada la creación de objetos. Cada nueva familia de productos obliga a agregar una nueva fabrica y, por lo tanto, una nueva clase.
- De forma opuesta, el patrón Prototype presentando anteriormente proporciona la posibilidad de crear nuevos objetos de manera muy flexible.

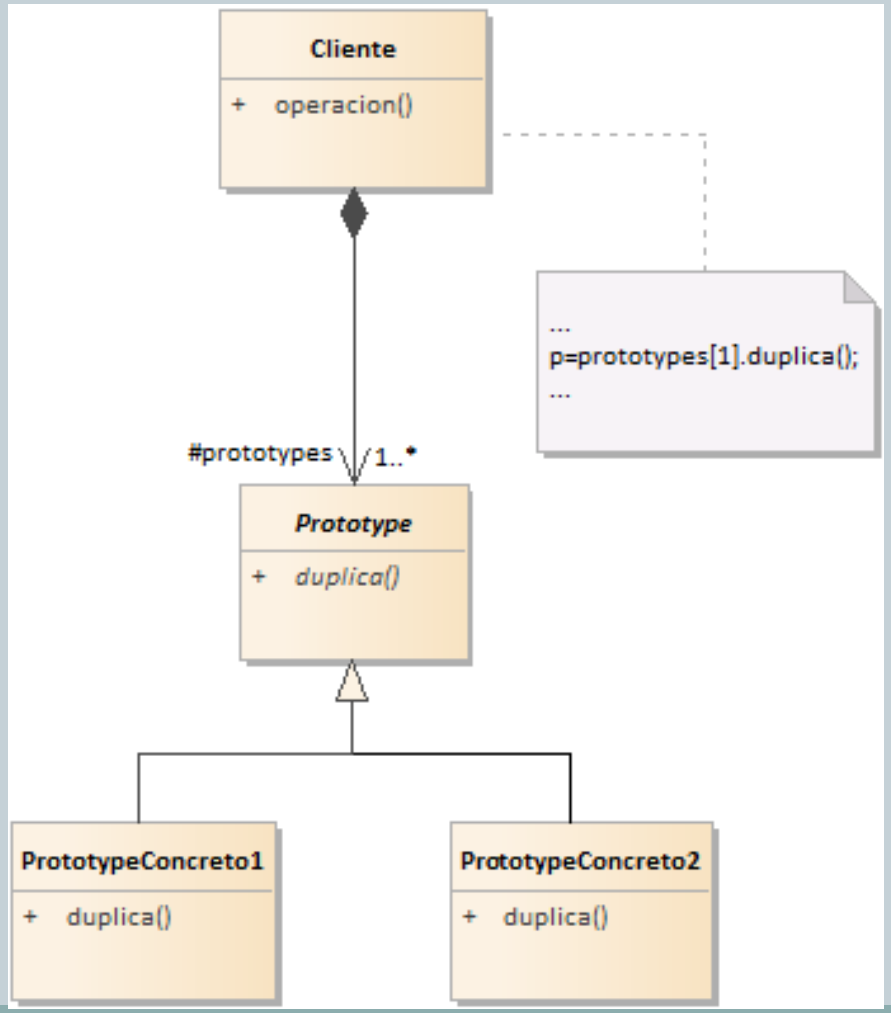
El patrón Pluggable Factory, II



El patrón Pluggable Factory, III

- La estructura del patrón Prototype se describe en la figura del acetato 6. Un objeto inicializado y listo para ser utilizado con capacidad de duplicarse se llama un prototipo.
- El cliente dispone de una lista de prototipos que puede duplicar cuando así lo desea. Esta lista se construye dinámicamente y puede modificarse en cualquier momento a lo largo de la ejecución. El cliente puede construir nuevos objetos sin conocer la jerarquía de clases de la que provienen.
- La idea del patrón Pluggable Factory consiste en componer estos dos patrones para conservar por un lado la idea de la creación de un producto invocando un método de la fabrica y por otro lado la posibilidad de cambiar dinámicamente la familia que se quiere crear.
- De este modo, la fabrica no necesita conocer las familias de objetos, el número de familias puede ser diferentes para cada producto y, por último, es posible variar el producto que se quiere crear no sólo únicamente por su subclase (su familia) sino también por valores diferentes de ciertos atributos.

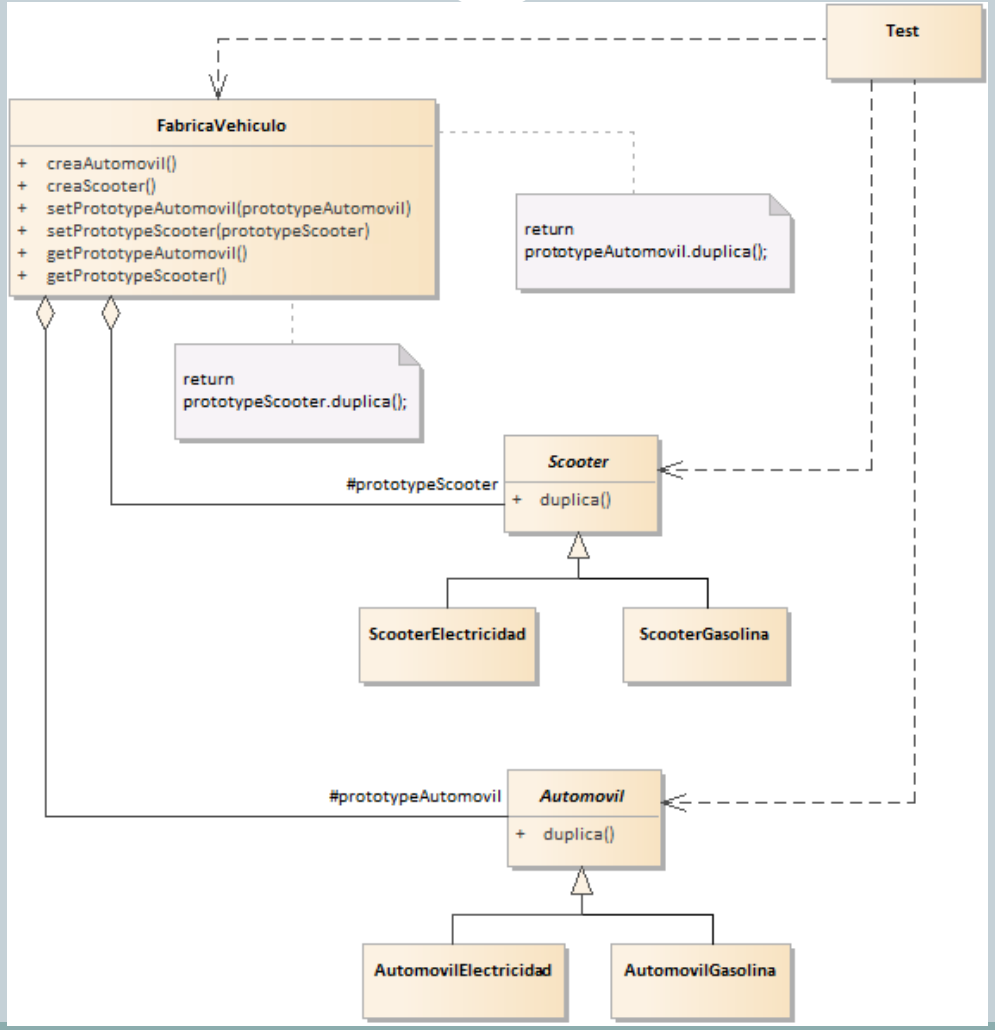
El patrón Pluggable Factory, IV



El patrón Pluggable Factory, V

- Profundizaremos este último punto en el ejemplo en Java.
- La figura del patrón 8 retoma el ejemplo del patrón Abstract Factory estructurado esta vez con ayuda del patrón Pluggable Factory.
- La clase de fabrica de objetos FabricaVehiculo ya no es una interafz como la figura del acetato 4 sino una clase concreta que permite la creación de los objetos y que no necesita subclases.
- Cada fabrica posee un enlace hacia un prototipo de cada producto. De forma más precisa, se trata de un vínculo hacia una instancia de una de las subclases de la familia Automovil y de un vínculo hacia una instancia de una de las subclases de la clase Scooter.
- Es aquí donde interviene el patrón Prototype.
- Cada producto se convierte en un prototipo. La clase abstracta que presenta y describe cada familia de productos le confiere la capacidad de clonado. Juega así el rol de la clase abstracta Prototype de la figura del acetato 6.

El patrón Pluggable Factory, VI



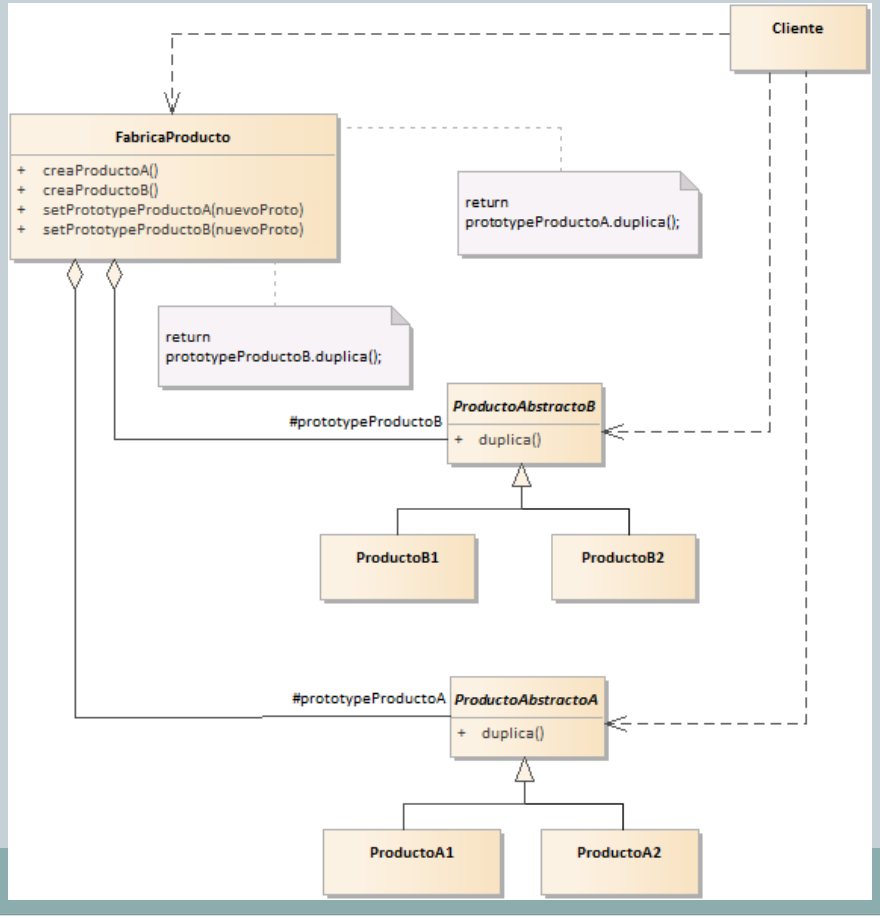
El patrón Pluggable Factory, VII

- Los dos enlaces presentes en FabricaVehiculo hacia cada prototipo pueden modificarse dinámicamente mediante los métodos setPrototypeAutomovil y setPrototypeScooter.
- La fabrica necesita, por otro lado, que sus clientes estén inicializados mediante estos dos métodos o bien por algún otro medio (como el constructor de la clase) para poder funcionar.
- El funcionamiento de la fabrica lo realizan los dos métodos creaAutomovil y creaScooter que se apoyan en la capacidad de clonado de ambos prototipos.
- La clase Test representa el cliente de la fabrica y las clases de los productos. Veremos su rol en el ejemplo Java.
- El nombre del patrón proviene por un lado del funcionamiento de la fabrica de productos que es dependiente de los prototipos proporcionados y por otro lado de la posibilidad de cambiar (enchufar, “plug” en inglés) dinámicamente estos prototipos.
- La figura del acetato 8 muestra el ejemplo del patrón Pluggable Factory.

El patrón Pluggable Factory, VIII

Estructura

- La siguiente figura ilustra la estructura genérica del patrón.



El patrón Pluggable Factory, IX

- Los participantes del patrón son los siguientes:
  - FabricaProducto (FabricaVehiculo) es la clase concreta que mantiene los vínculos hacia los prototipos de producto, ofrece los métodos que crean los distintos productos así como los métodos que permiten fijar los prototipos.
  - ProductoAbstractoA y ProductoAbstractoB (Scooter y Automovil) son las clases abstractas de los productos independientemente de su familia. Proporcionan a los productos la capacidad de clonado para conferirles el status de prototipo. Las familias se incluyen en sus subclases concretas.
  - Cliente es la clase que utiliza la clase FabricaProducto.
- La colaboración entre los objetos se describe a continuación:
  - El cliente crea u obtiene los prototipos conforme los necesita.
  - El cliente crea una instancia de la clase FabricaProducto y le proporciona los prototipos necesarios para su funcionamiento.
  - A continuación utiliza esta instancia para crear sus productos a través de los métodos de creación. Puede proporcionar nuevos productos a la fabrica.

El patrón Pluggable Factory, X

- A diferencia del patrón Abstract Factory, donde se aconseja crear una única instancia de las fábricas concretas (las cuales no poseen estado), aquí es concebible crear varias instancias de la fabrica de productos, estando cada instancia ligada a prototipos distintos de los productos.

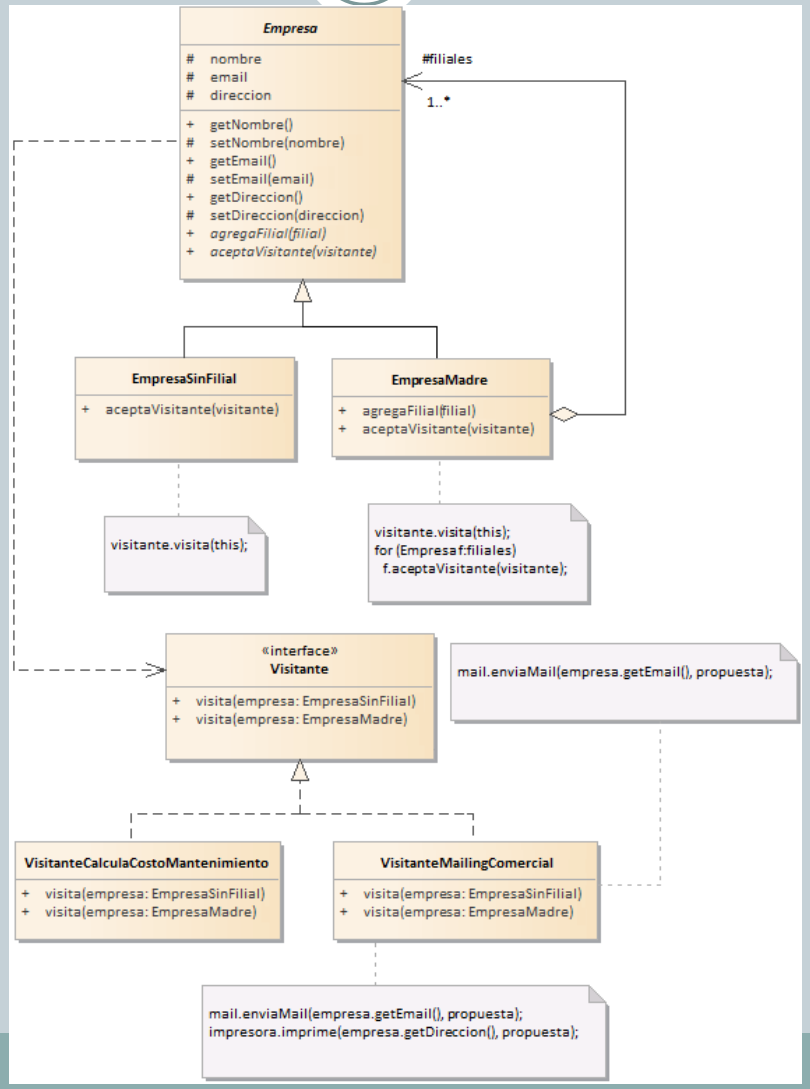
Ejemplo en Java

- A continuación mostramos el código Java del ejemplo correspondiente al diagrama de clases de la figura del acetato 8.
- Presentamos primero las clases correspondientes a los productos (Automovil y Scooter) así como sus subclases.
- Cada destacar que estas clases definen ahora prototipos. Su capacidad de clonado la provee el método duplica. Los métodos de acceso permiten fijar y obtener el valor de los atributos pertinentes de estos objetos.
- Ver la carpeta correspondiente.

El patrón Reflective Visitor, I

- Hemos estudiado anteriormente el patrón Visitor para poder agregar nuevas funcionalidades a un conjunto de clases sin tener que modificar estas clases tras cada agregación.
- Cada nueva funcionalidad da pie a una clase de visitante que implementa esta funcionalidad incluyendo un conjunto de métodos, uno por cada clase.
- Todos estos métodos tienen el mismo nombre, por ejemplo visita, y tienen un único parámetro cuyo tipo es el de la clase para la que se implementa la funcionalidad.
- No obstante para implementar el patrón Visitor, las clases que deben ser visibles requieren una ligera modificación, a saber la inclusión de un método para aceptar al visitante, método cuyo único fin es invocar al método visita con un parámetro correctamente tipado.
- El nombre de este método es a menudo acepta o aceptaVisitante.
- La figura del acetato 14 muestra una implementación del patrón Visitor con el objetivo de visitar una jerarquía de objetos descrita mediante el patrón Composite.

El patrón Reflective Visitor, II



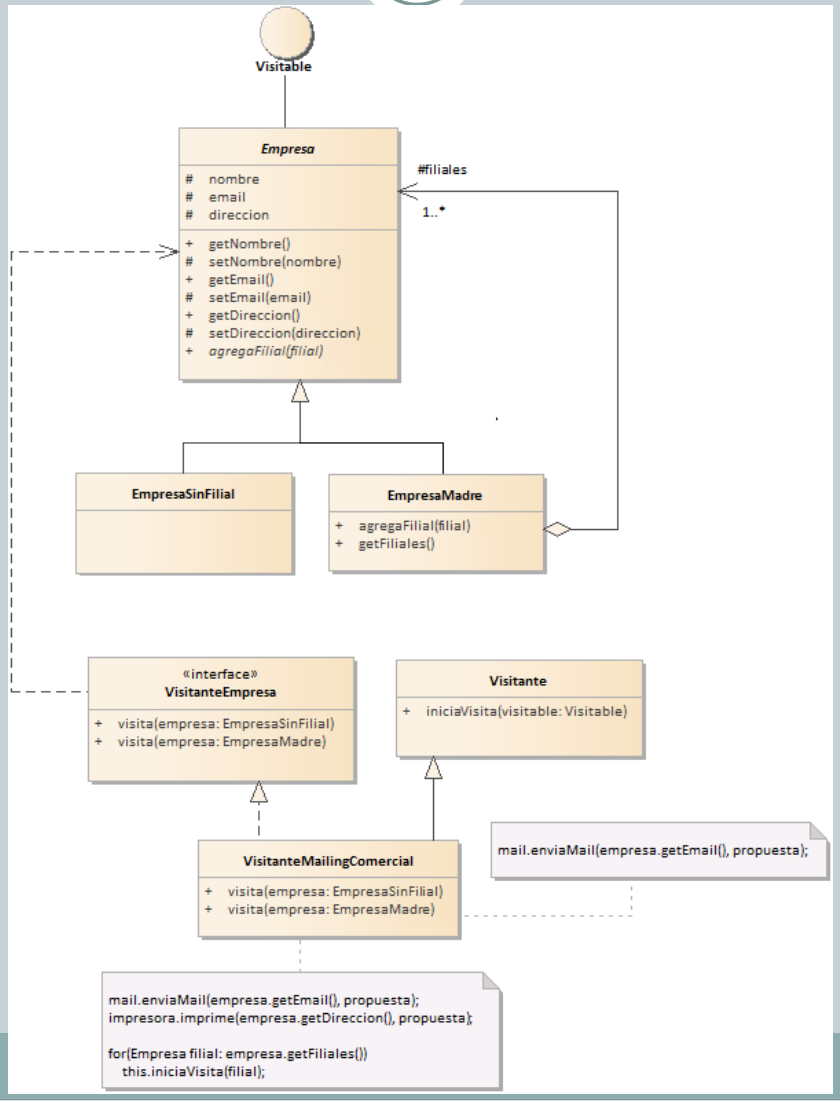
El patrón Reflective Visitor, III

- Estos objetos son empresas que en ocasiones, cuando se trata de las empresas madres, poseen filiales.
- Las dos funcionalidades agregadas son el cálculo de los costos de mantenimiento y la posibilidad de enviar un mailing comercial a una empresa y a todas sus filiales, incluyendo a las filiales de las filiales.
- El método aceptaVisitante de la clase EmpresaMadre incluye un ciclo foreach que solicita a cada una de las filiales que acepte a su visitante.
- El patrón Reflective Visitor es una variante del patrón Visitor que utiliza la capacidad de reflexión estructural del lenguaje de programación de modo que la implementación no requiera la inclusión del método de aceptación del visitante en las clases que deban ser visitadas.
- La reflexión estructural de un lenguaje de programación es su capacidad para proveer un medio de examinar el conjunto de clases que forman el programa y su contenido (atributos y métodos).

El patrón Reflective Visitor, IV

- La reflexión estructural existe al día de hoy en la mayor parte de los lenguajes de programación orientados a objetos (Java y C# integran esta capacidad).
- De este modos la implementación anterior se simplifica con el patrón Reflective Visitor tal y como se muestra en la figura del acetato 17. Esta implementación difiere de la implementación anterior en los siguientes puntos:
  - Las clases que representan a las empresas ya no incluyen un método para aceptar al visitante. Implementan la interfaz Visitable que sirve de tipo para el argumento del método iniciaVisitante del visitante.
  - La clase Visitante es una clase abstracta que incluye el método iniciaVisita, que desencadena la visita de un objeto. Todo visitante concreto hereda de este método. Su código consiste en encontrar el método visita del visitante mejor adaptado al objeto a visitar.
  - La interfaz VisitanteEmpresa especifica los dos métodos que todo visitante de las empresas debe implementar. Se trata de dos métodos destinados a visitar las dos subclases de la clase Empresa.

El patrón Reflective Visitor, V



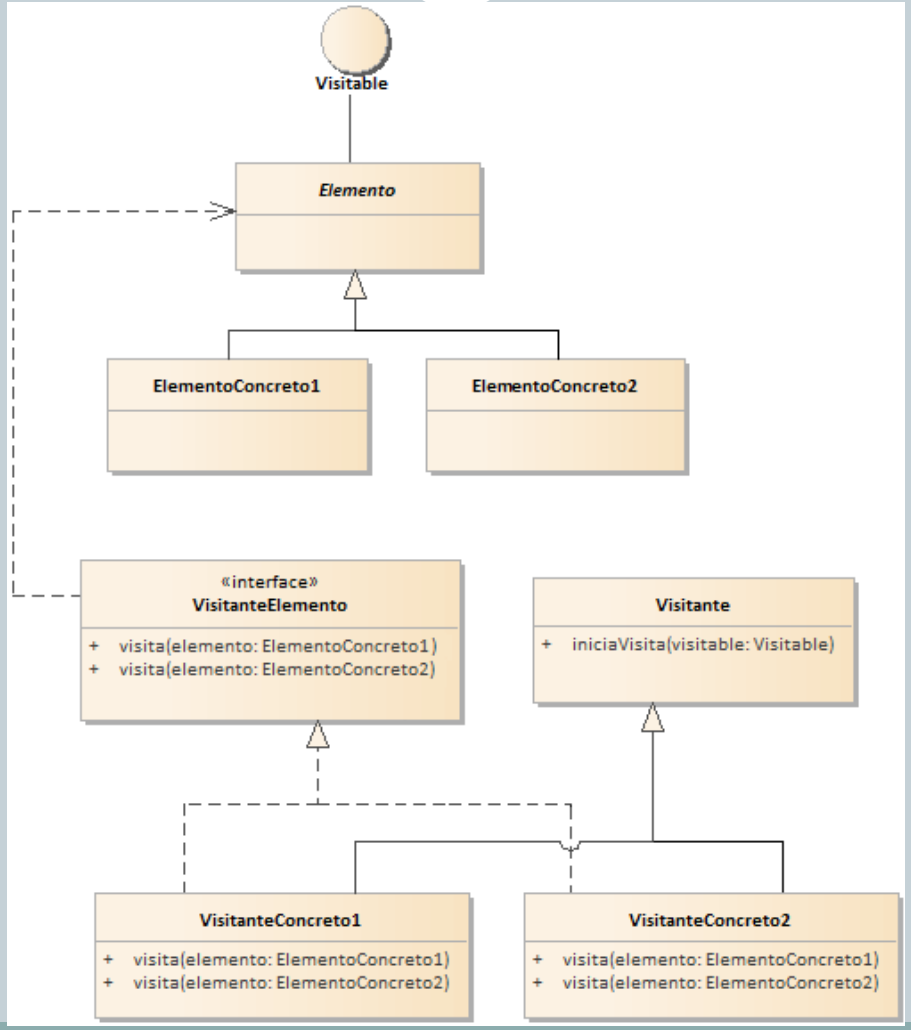
El patrón Reflective Visitor, VI

- La clase VisitanteMailingComercial describe el visitante, cuya funcionalidad consiste en enviar un mailing a una empresa y a todas sus filiales. El método destinado a visitar una empresa madre incluye en su código un ciclo foreach que inicia la visita de sus filiales. Para ello, utiliza un acceso de lectura de la asociación filiales que se ha agregado en la clase EmpresaMadre. En la implementación del patrón Visitor, la visita de las filiales se desencadenaba en el método aceptaVisitante de la clase EmpresaMadre. Habiendo desaparecido este último método, le corresponde al visitante iniciar esta visita.

Estructura

- La figura del acetato 19 detalla la estructura del patrón Reflective Visitor.
- Los participantes del patrón son los siguientes:
  - Visitante es la clase abstracta que incluye el método iniciaVisita que desencadena la visita de un objeto. Todo visitante concreto hereda de este método. Su código consiste en encontrar el método visita del visitante mejor adaptado al objeto a visitar, es decir aquel cuyo tipo del argumento corresponda con la clase de instanciación del objeto a visitar o, en su defecto, aquel cuyo tipo del argumento sea la superclase o la interfaz más próxima a la clase de instanciación del objeto a visitar.

El patrón Reflective Visitor, VII



El patrón Reflective Visitor, VIII

- VisitanteElemento es la interfaz que incluye la firma de los métodos que realizan una funcionalidad en un conjunto de clases. Existe un método por cada clase que recibe como argumento una instancia de esta clase.
- VisitanteConcreto1 y VisitanteConcreto2 (VisitanteMailingComercial) implementan los métodos que realizan la funcionalidad correspondiente a la clase.
- Visitable es la interfaz vacía que sirve para tipar las clases visitables. Es el tipo del argumento del método iniciaVisita de la clase Visitante.
- Elemento (Empresa) es una clase abstracta superclase de las clases de elementos. Implementa la interfaz Visitable.
- ElementoConcreto1 y ElementoConcreto2 (EmpresaSinFilial y EmpresaMadre) son las dos subclases concretas de la clase Elemento. No requieren ninguna modificación para recibir un visitante.

- La colaboración entre los objetos se describe a continuación:
  - Un cliente que utiliza un visitante debe en primer lugar crearlo como instancia de la clase de visitante de su elección e invocar al método iniciaVisita de este visitante pasando como parámetro el objeto a visitar.

El patrón Reflective Visitor, IX

- El método iniciaVisita del visitante encuentra el método visita del visitante mejor adaptado al objeto a visitar y, a continuación, lo invoca.

Ejemplo en Java

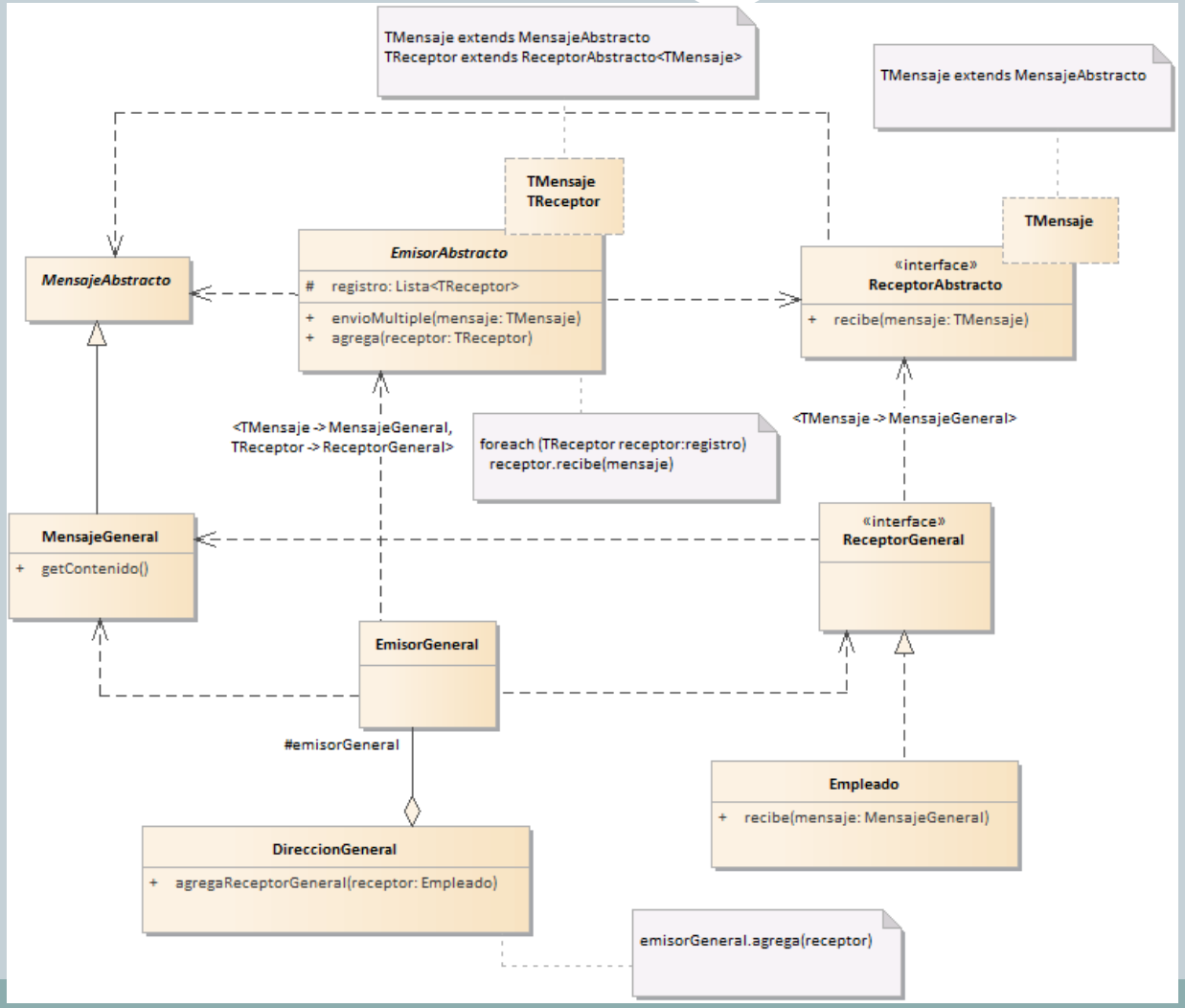
- A continuación se muestra el código escrito en Java del ejemplo correspondiente al diagrama de clases de la figura del acetato 17.
- Presentamos en primer lugar las clases que describen a las empresas así como la interfaz Visitable que implementa la clase abstracta Empresa.
- Ver el código del patrón.

El patrón Multicast, I

Descripción y ejemplo

- El objetivo del patrón Multicast es gestionar los eventos producidos en un programa para transmitirlos a un conjunto de receptores afectados. El patrón está basado en un mecanismo de suscripción de receptores en los emisores.
- Queremos implementar un programa de envío de mensajes entre las direcciones (general, comercial, financiera, etc.) de un concesionario y sus empleados.
- Cada empleado puede suscribirse a la dirección a la que pertenece y recibir todos los mensajes emitidos por ella. Un empleado no puede suscribirse a una dirección a la que no pertenece. Todos los empleados pueden suscribirse a la dirección general para recibir sus mensajes.
- La estructura de los mensajes pueden variar de una dirección a otra: desde una simple línea de texto para los mensajes comerciales, hasta un conjunto de líneas para los mensajes generales provenientes de la dirección general.
- El diagrama de clases de la figura del acetato 23 exponen la solución proporcionada por el patrón Multicast.

El patrón Multicast, II



El patrón Multicast, III

- La genericidad de tipos se utiliza para crear un mensaje, un emisor y un receptor abstracto y genérico, a saber las clases MensajeAbstracto y EmisorAbstracto así como la interfaz ReceptorAbstracto.
- La clase EmisorAbstracto presenta dos funcionalidades:
  - Gestiona un registro (una lista) de receptores con la posibilidad de suscribirse gracias al método agrega.
  - Permite enviar un mensaje al conjunto de receptores presentes en el registro gracias al método envioMultiple.
- Está basada en dos tipos genéricos, a saber Tmensaje que está acotado por MensajeAbstracto y Treceptor acotado por ReceptorAbstracto<TMensaje>.
- De este modo, cualquier mensaje es obligatoriamente una subclase de MensajeAbstracto y todo receptor una subclase de ReceptorAbstracto<TMensaje>.
- La clase MensajeAbstracto es una clase abstracta totalmente vacía. Sólo existe con fines de tipado.

El patrón Multicast, IV

- La interfaz ReceptorAbstracto es una interfaz que incluye la forma del método recibe. Esta interfaz está basada en el tipo genérico TMensaje acotado por MensajeAbstracto.
- Nos interesamos ahora en el caso particular de los mensajes que provienen de la dirección general. Para estos mensajes, creamos una subclase para cada clase abstracta:
  - La subclase concreta MensajeGeneral que describe la estructura de un mensaje de la dirección general. El método getContenido permite obtener el contenido de dicho mensaje.
  - La subclase concreta EmisorGeneral obtenida vinculando los dos parámetros genéricos con MensajeGeneral para TMensaje y a ReceptorGeneral para TReceptor.
  - La interfaz ReceptorGeneral que hereda de la interfaz ReceptorAbstracto vinculando el parámetro genérico TMensaje con MensajeGeneral.
- La clase Empleado incluye los objetos que representan a los empleados del concesionario. Implementa la interfaz ReceptorGeneral.

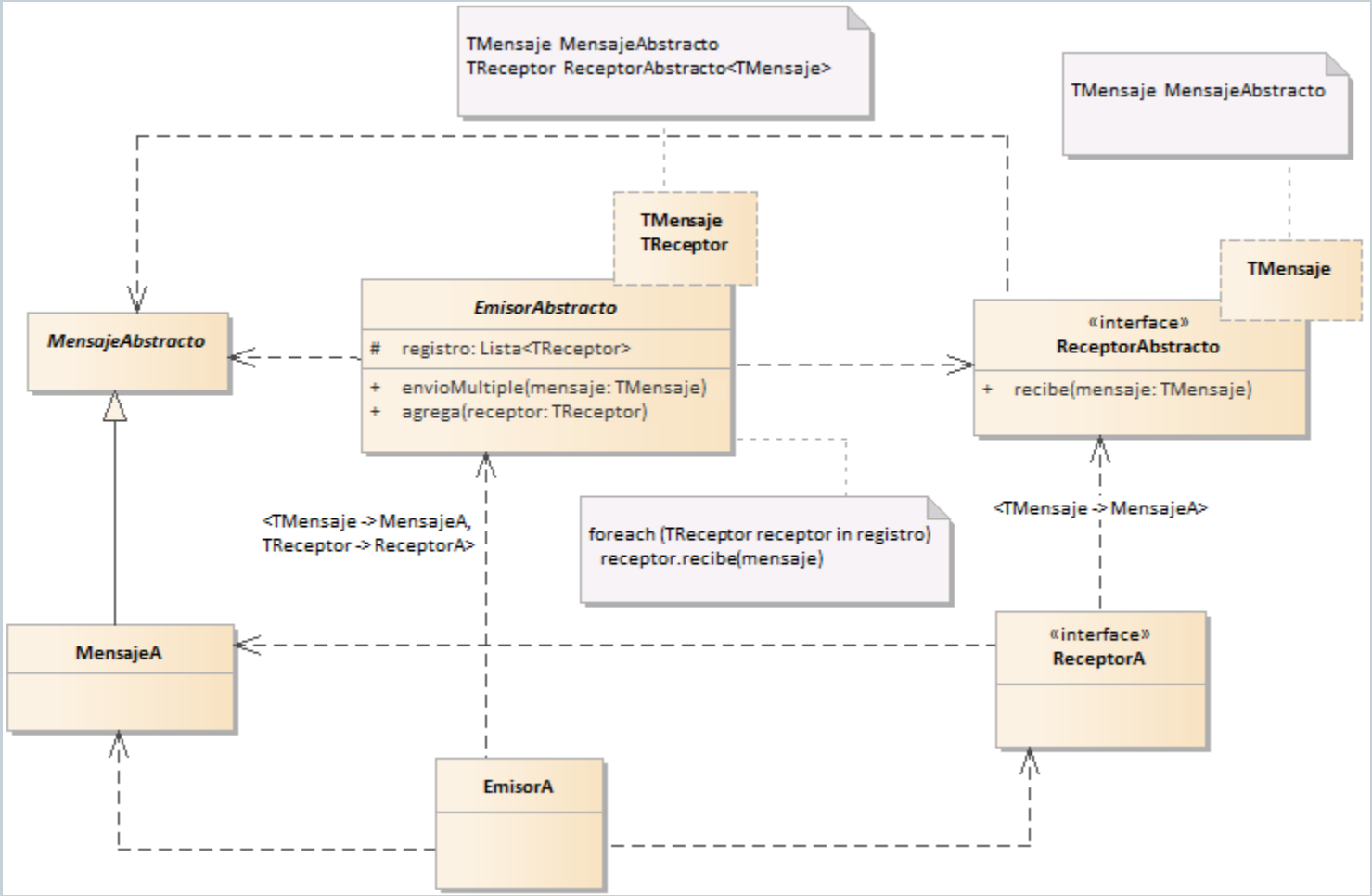
El patrón Multicast, V

- De este modo sus instancias estarán dotadas de la capacidad de recibir los mensajes provenientes de la dirección general y pueden suscribirse para recibirlos.
- La clase DireccionGeneral representa a la dirección general.
- Posee un enlace hacia emisorGeneral, instancia de la clase EmisorGeneral que le permite enviar mensajes.
- El método agregaReceptorGeneral permite reportar la suscripción de los empleados a nivel de la clase DireccionGeneral.

Estructura

- La estructura genérica del patrón Multicast se ilustra en el diagrama de clases de la figura del acetato 27.
- Este diagrama de clases es muy similar al diagrama del ejemplo de la figura del acetato 23, habiendo conservado las clases abstractas en el ejemplo.

El patrón Multicast, VI



El patrón Multicast, VII

- Los participantes del patrón son los siguientes:
  - MensajeAbstracto es la clase abstracta que incluye el tipo de los mensajes.
  - EmisorAbstracto es la clase abstracta que implementa el registro de los receptores y el método envioMultiple que envía un mensaje a todos los receptores del registro.
  - ReceptorAbstracto es la interfaz que defien la firma del método recibe.
  - MensajeA (MensajeGeneral) es una subclase concreta de MensajeAbstracto que describe la estructura de los mensajes.
  - EmisorA (EMisorGeneral) es una subclase concreta que representa a los emisores de mensajes. Vincula el parámetro TMensaje con MensajeA y el parámetro TReceptor con ReceptorA.
  - ReceptorA (ReceptorGeneral) es una interfaz que hereda de ReceptorAbstracto vinculando el parámetro TMensaje con MensajeA. Debe implementarse en todos los objetos que quieran tener la capacidad de recibir mensajes tipados por la clase MensajeA.
- La colaboración entre los objetos se describe a continuación:
  - Los receptores se suscriben al emisor del mensaje.

El patrón Multicast, VIII

- Los emisores envían mensajes a los receptores suscritos.

Ejemplo en Java

- A continuación se muestra el código escrito en Java del ejemplo correspondiente al diagrama de clases de la figura del acetato 23.
- Presentamos en primer lugar las clases abstractas e interfaces MensajeAbstracto, ReceptorAbstracto y EmisorAbstracto.
- A continuación se muestran las clases e interfaces relativos a los mensajes generales: MensajeGeneral, ReceptorGeneal y EmisorGeneral.
- El código de la clase DireccionGeneral se muestra a continuación. Esta clase posee un vínculo hacia una instancia de la clase EmisorGeneral. Está se utiliza para agregar empleados al registro y para enviar los mensajes.
- La clase Empleado aparece a continuación. Implementa la interfaz ReceptorGeneral para poder recibir mensajes generales. Esta clase es abstracta: la dotaremos de dos subclases concretas Administrativo y Comercial.

El patrón Multicast, IX

- La subclase concreta Administrativo aparece a continuación. Es muy simple.
- Presentamos también un segundo mensaje: los mensajes comerciales ligados a la dirección comercial. El código escrito en Java de las clases e interfaces correspondientes, es MensajeComercial, ReceptorComercial, EmisorComercial, DireccionComercial y Comercial.
- La clase Comercial no implementa ReceptorComercial sino que utiliza en su lugar una instancia de una clase anónima interna, ¡que implementa dicha interfaz! Es esta instancia y no la instancia de la clase Comercial la que recibe los mensajes.
- La razón es la que Java rechaza que una clase implemente dos veces la misma interfaz. Si la clase comercial implementara ReceptorComercial, entonces implementaría dos veces la interfaz Receptor-Abstracto: una primera vez con ReceptorComercial que hereda de ReceptorAbstracto<MensajeComercial> y una segunda vez heredando de la clase Empleado que implementa ReceptorGeneral que hereda de ReceptorAbstracto<MensajeGeneral>.

El patrón Multicast, X

- Por último, se muestra el código escrito en Java de un programa de prueba para este conjunto de clases.
- Ver la carpeta del patrón.

Comparación con el patrón Observer, I

- El patrón Observer presenta grandes similitudes con el patrón Multicast. Por un lado, permite inscribir observadores, el equivalente a los receptores. Por otro lado, puede enviar una notificación de actualización a los observadores, es decir un equivalente a los mensajes.
- En el caso del patrón Observer, la notificación de actualización no transmite información, a diferencia de los mensajes del patrón Multicast. No obstante, no es muy complicado extender el patrón Observer para agregar una transmisión de información durante la notificación de actualización.
- Es por lo tanto válido preguntarse si el patrón Multicast no es más que una simple extensión del patrón Observer. La respuesta es negativa.
- El objetivo del patrón Observer es construir una dependencia entre un sujeto y los observadores de modo que cada modificación del sujeto se notifique a sus observadores.
- Un conjunto formado por el sujeto y sus observadores constituye, de cierta manera, un único objeto compuesto.

Comparación con el patrón Observer, II

- Por otro lado, un uso casi inmediato del patrón Multicast es la posibilidad de crear varios emisores enviando mensajes a un único o varios receptores. Un receptor puede estar conectado a varios emisores, como es el caso de nuestro ejemplo donde un comercial puede recibir mensajes de la dirección general y de la dirección comercial.
- Este uso es opuesto al objetivo del patrón Observer, y demuestra que Observer y Multicast son en efecto dos patrones diferentes.