

LA BIBLIA DE MS

Dicen que las segundas partes nunca fueron buenas, pero ¿y las tercera?

PATRONES MULTICAPA

Capa de presentación:

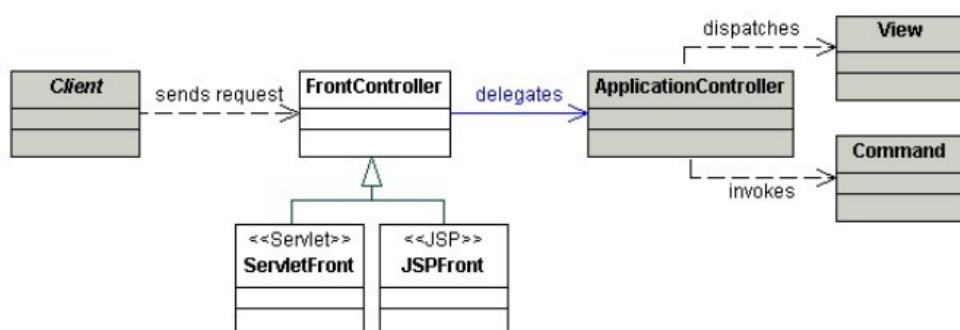
1. Front Controller

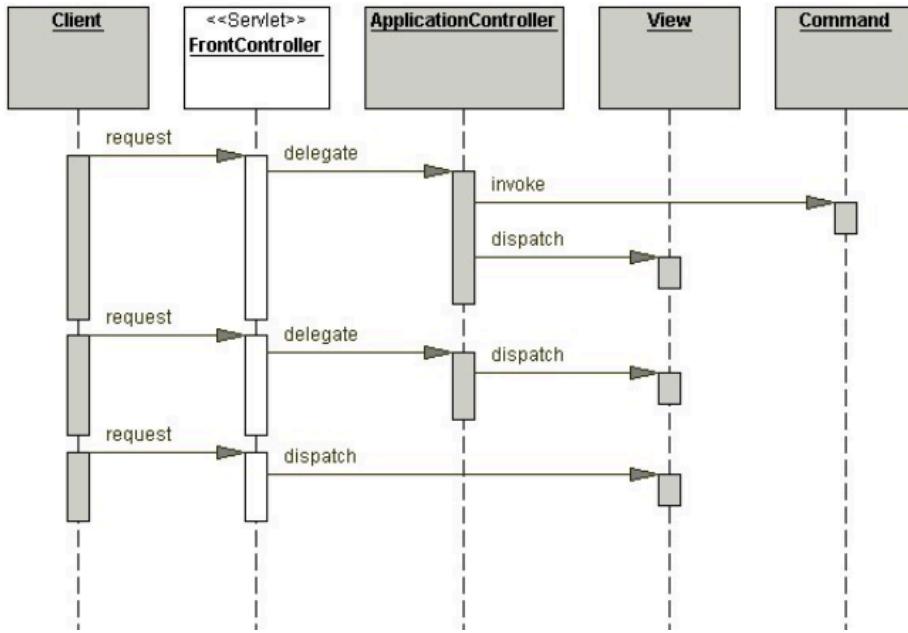
Es el punto de acceso del servidor que se "come" las peticiones HTTP (es un servlet) y se las pasa a un controlador invocando una vista o la lógica de negocio correspondiente.

En resumen, es un adaptador de entrada que transforma HTTP en algo orientado a objetos.

- Utiliza un `requestContext` para traducir el string HTTP
- Se crea un Application Controller
- Usando el `requestContext` se lo pasa al controlador para que gestione la petición y devuelva la respuesta.

En nuestra aplicación no lo usamos, ya que es Swing (usamos `ActionListener`).





2. Application Controller

Mapea eventos de entrada al servicio de aplicación.

Para volver más sistemático su proceso utilizamos el patrón Command, que encapsula las operaciones de negocio y la redirección a las vistas las colocamos al final.

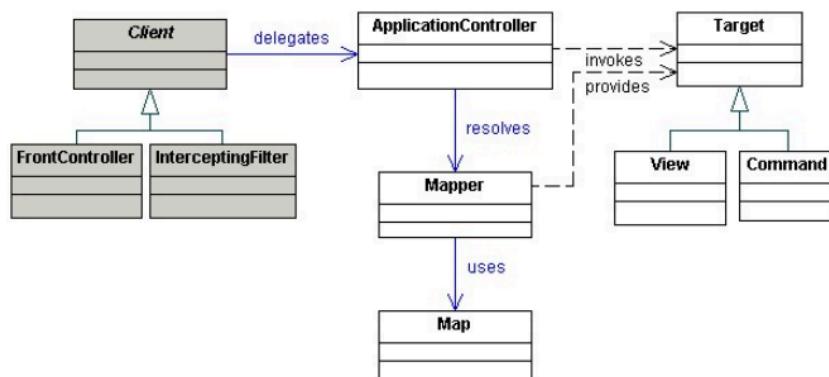
Los mecanismos para mapear eventos a comandos y de comandos a vistas se denominan factorías, y así tenemos factoría de comandos y factoría de vistas.

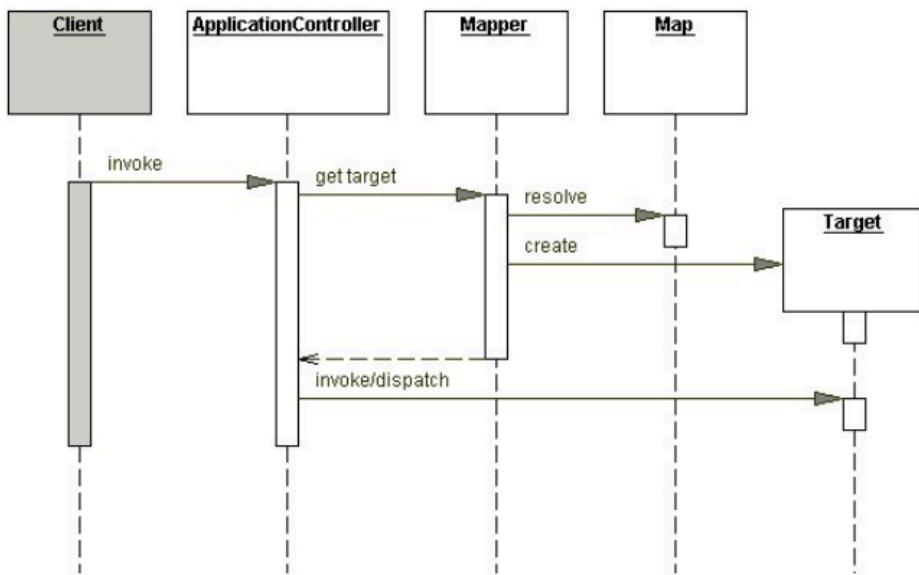
Ventajas:

- Centraliza la invocación y recuperación de componentes de procesado de peticiones
- Mejora la reutilización
- Mejora la modularidad.

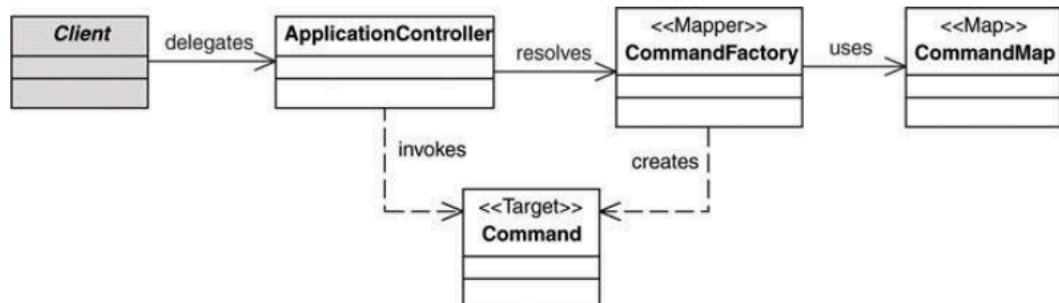
Desventajas:

- En aplicaciones grandes puede llegar a crecer mucho.



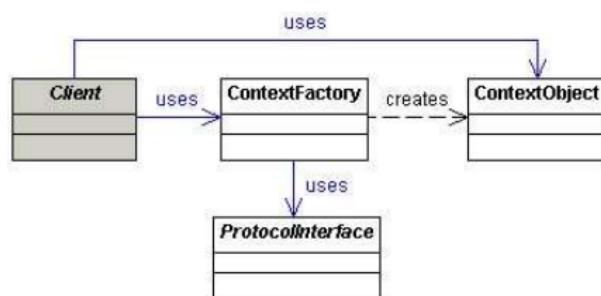


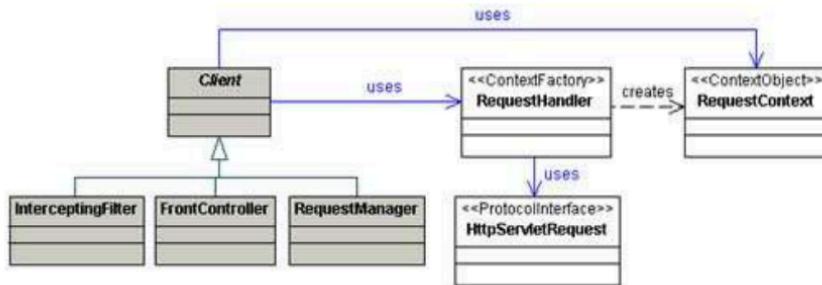
Con Command:



3. Contexto

Encapsula el estado de una forma independiente del protocolo para ser compartida por toda la aplicación. Es decir, se desea desacoplar componentes de aplicación y servicios propios del protocolo y se desea exponer las APIs necesarias. (Separar el protocolo HTTP de los datos).





Estructura del request context

No tenemos requestContext porque el actionListener separa la vista de la información que necesitamos (transfers).

No tenemos responseContext de comandos ya que solo tenemos una pareja (int, object)

En nuestra aplicación solo hay Application Controller, por lo que la secuencia handleRequest/handleResponse debe hacerse en la operación *handle* en el Application.

El contexto en nuestras aplicaciones se ve relegado a un simple par(int, object), que nos sirve de entrada para el Application Controller y de salida para el comando.

Ventajas:

- Mejora la reusabilidad
- Mejora las pruebas

Desventajas:

- Reduce el rendimiento

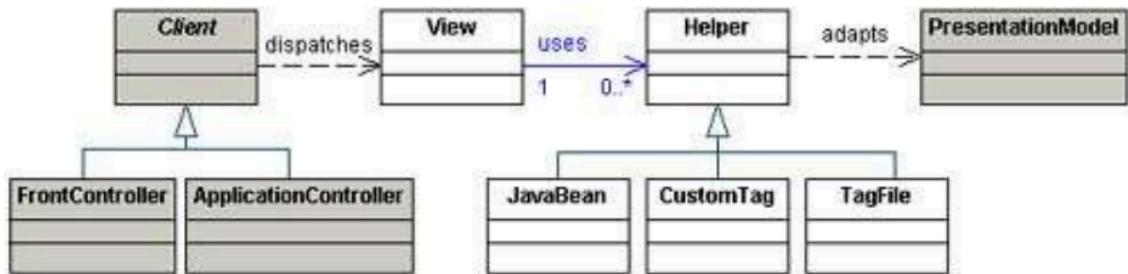
4. View Helper

Normalmente, las vistas a partir de un conjunto de información(transfer) se tienen que construir. El ayudante de vista facilita este proceso ayudando a la vista a construir parte de esta y puede ser usado por otras vistas que tengan un proceso de construcción común.

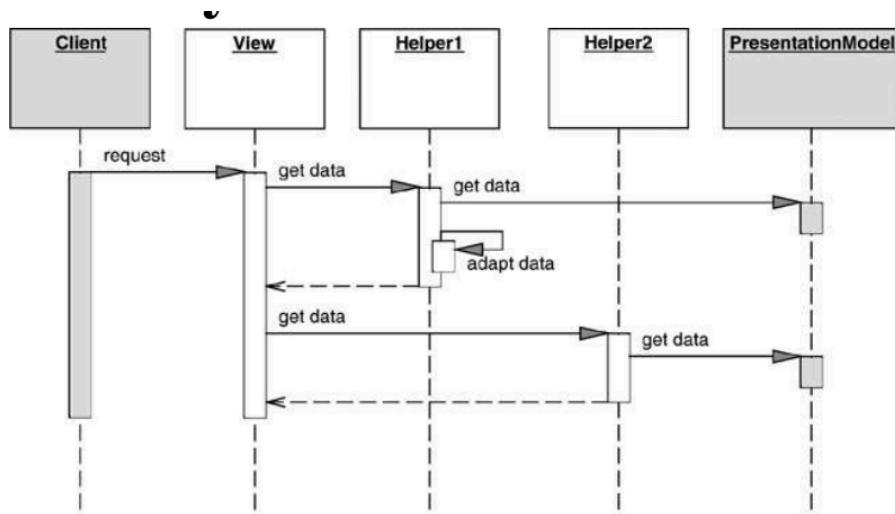
Es decir, una vista delega sus responsabilidades de procesamiento a sus ayudantes, y éstas llevan a cabo procesamiento relacionado con el formateo.

Por ejemplo, si queremos listar productos, clientes... Un ayudante de vista puede ser aquel que se encargue de construir una tabla para listar.

En nuestro proyecto era opcional su uso.



El controlador frontal o de aplicación dirige una vista y va a utilizar un helper para adaptar la información en un modelo de presentación.



Interacción en el patrón ayudante de vista

5. Service to Worker

Se utiliza cuando se desea llevar a cabo el manejo de peticiones y la invocación de la lógica de negocio antes de pasar el control a la vista.

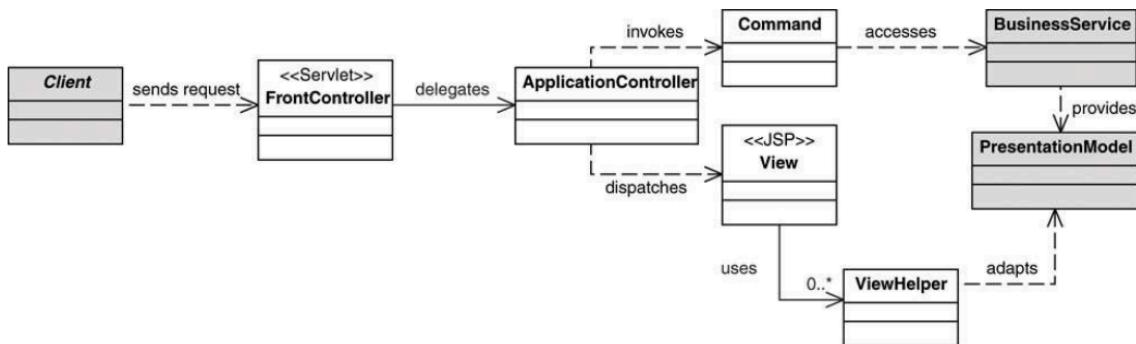
Articula en un solo patrón el Front Controller, el Application Controller y el Ayudante de Vista.

Ventajas:

- Centraliza el control y mejora la reusabilidad, modularidad y mantenibilidad.
 - El application controller es reutilizable entre aplicaciones y únicamente hay que cambiar los mappers
 - Si los mappers usan carga dinámica, solo hay que cambiar los ficheros de configuración de texto

Desventajas:

- Complejidad



6. Intercepting Filter

Se usa cuando es necesario preprocesar la petición antes de enviarla a negocio, y que sea débilmente acoplado con el mismo. Los componentes de pre y posprocesado se quiere que sean independientes.

Se puede añadir, eliminar y combinar filtros sin modificar el código existente.

Ventajas:

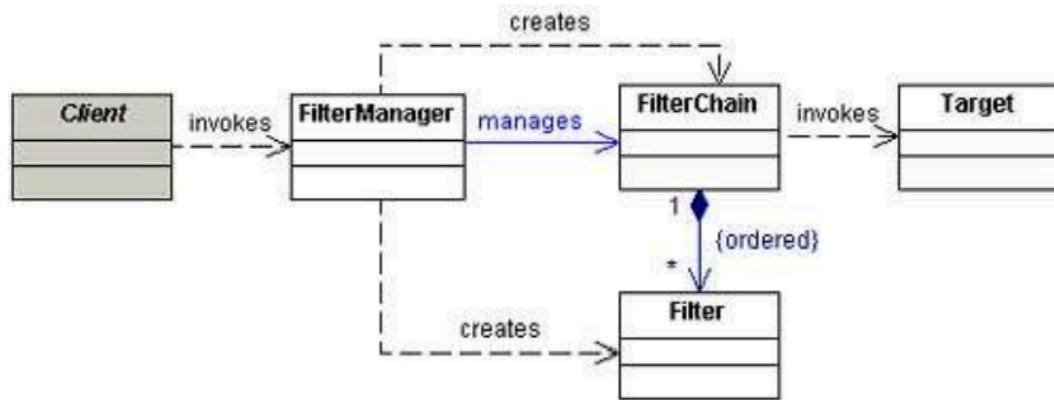
- Centraliza el control
- Mejora la reusabilidad
- Lugar centralizado y común de las peticiones

Desventajas:

- No está pensado para que un filtro llame a otro

Diferencia entre un Intercepting Filter y una Cadena de Responsabilidad:

- La cadena tiene una lógica de "o te trato yo, o te trata el siguiente"
- Un intercepting tiene lógica "te trato yo y te trata el siguiente...", por ejemplo compruebo que tu sesión es válida, que tu navegador es compatible....



7. Composite View

Cuando navegamos por el navegador tenemos:

- Un header que se mantiene constante.
- Un footer
- Un contenido que va cambiando

Yo quiero que mis páginas tengan siempre esta estructura. Todos los lenguajes de programación web tienen una cosa que se llaman templates, ¿por qué se llaman así?

Porque es una forma de hacer páginas web compuestas, entonces es un mecanismo para decir que todas las páginas web van a tener una parte común. Lo podemos simular con JFrames con plantillas.

Aquí es donde aparecen las vistas compuestas, de forma que sea más fácil tener partes comunes a todas sin duplicarse.

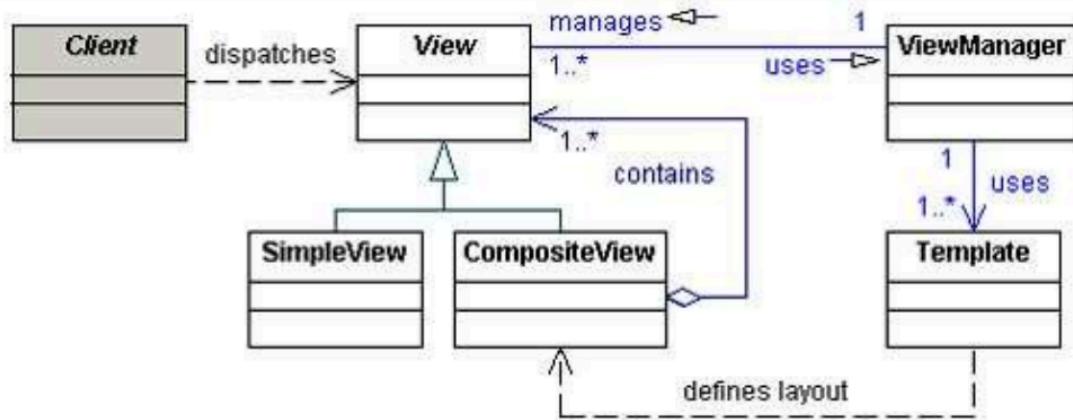
En resumen, una vista compuesta está constituida por varias vistas atómicas y el contenido es independiente.

Ventajas:

- Mejora la modularidad y reutilizabilidad

Desventajas:

- Reduce el rendimiento



8. Dispatcher View

En ocasiones donde la lógica de negocio es bastante limitada, y tenemos vistas estáticas, podemos invocar lógica de negocio desde las vistas, que son conocidas como despachadoras y que serán el punto de inicio para peticiones con respuestas estáticas (HTML) y dinámicas.

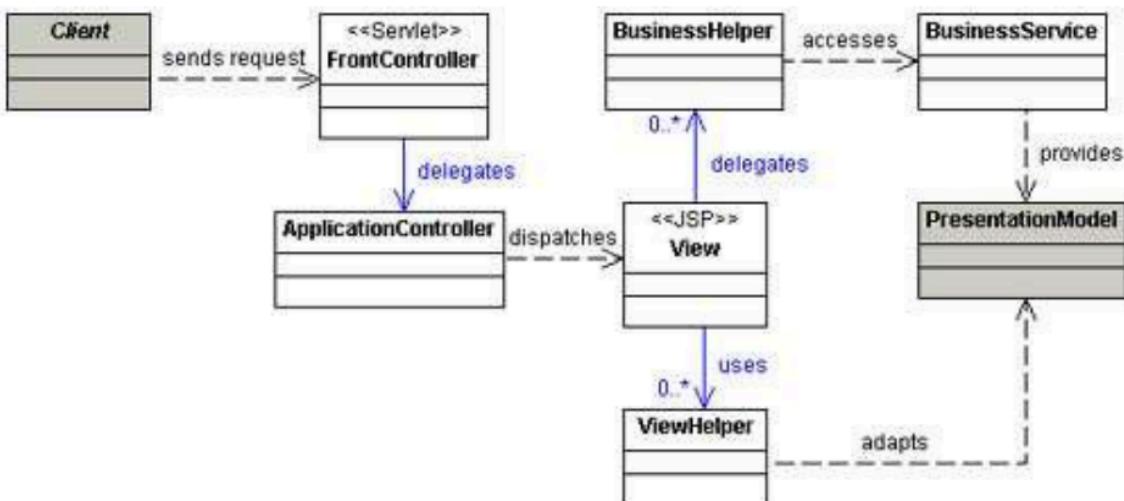
Usualmente estas vistas hacen reads de la base de datos (solo reciben los datos volcados por la BD).

Ventajas:

- Se aprovechan librerías

Desventajas:

- Potencial falta de separación entre la vista del modelo y la lógica de control



Capa de negocio:

9. Transfer

Sirve para mover datos de una capa a otra de forma que tengamos un bajo acoplamiento.

Son creados por:

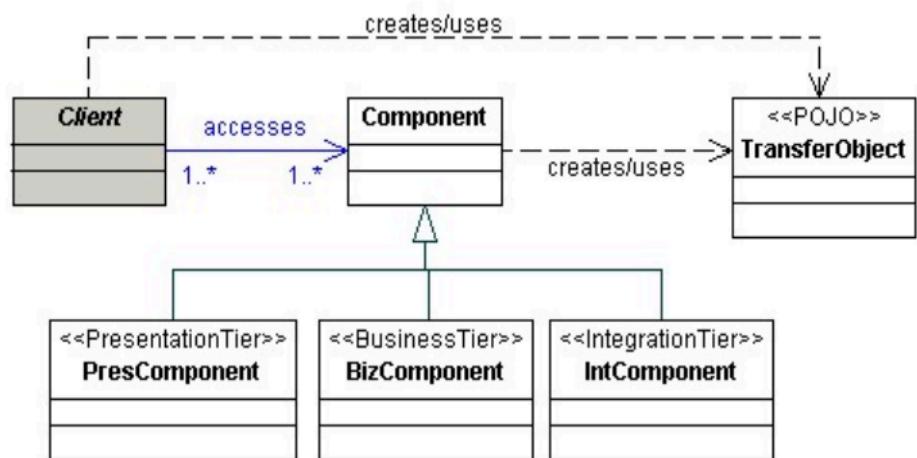
- Vista: van del usuario a la capa de recursos
- DAO: van de la capa de recursos hacia el usuario
- TOA: minoritariamente

Ventajas:

- Ayuda a independizar capas

Desventajas:

- Introduce datos desactualizados



10. Servicio de aplicación

Es aquel que implementa la lógica de negocio que proporciona todos los casos de uso. Estos indican una serie de pasos para cumplir una funcionalidad requerida por el usuario.

Es decir, proporciona una capa de servicio uniforme.

¿Quién debe iniciar y acabar la transacción?

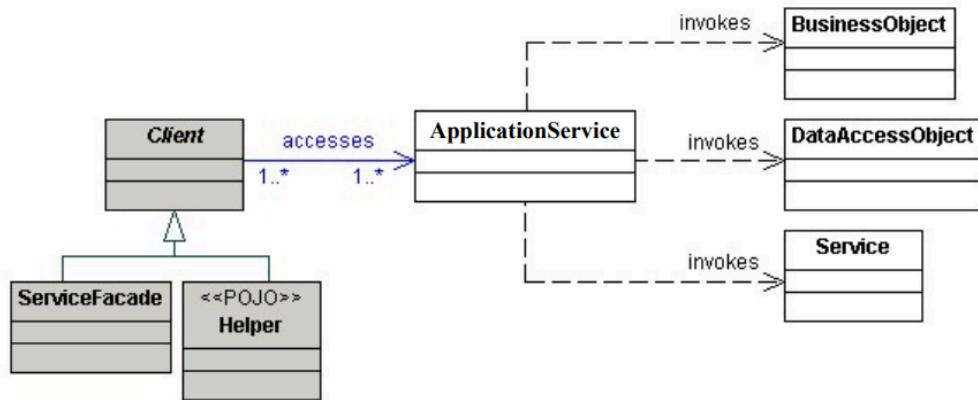
El SA, pues es el que lleva todo el curso de la operación. Se debe iniciar cuando se inicia la operación, y se cierra cuando acaba la operación con commit (éxito) o rollback (fracaso).

¿Por qué los SA y los DAOs no tienen atributos?

Si tengo la lógica de negocio distribuida, y hay varios procesos queriendo acceder a la misma clase, el servidor solo crea una copia del objeto, siendo compartido por varias hebras. Si tuviera atributos daría lugar a resultados indeseados.

Ventajas:

- Evita duplicidad de código
- Centraliza lógica de negocio



Usando DAOs, SAs y Transfers hemos partido la programación orientada a objetos, porque tenemos las operaciones por un lado y los datos por otro.

Por ello, si queremos una representación más fiel usamos [objetos de negocio](#).

11. Objeto de negocio

Son objetos que van a traducir directamente el modelo de dominio a una representación orientada a objetos y que van a poder implementar lógica de negocio.

Los objetos de negocio encapsulan y manejan datos del negocio, comportamiento y persistencia.

¿Los transfers pueden tener punteros entre ellos?

Si los transfer tuvieran punteros entre sí entonces se traería una gran parte de la base de datos a memoria, por lo que no es lógico. Por eso usamos TOA.

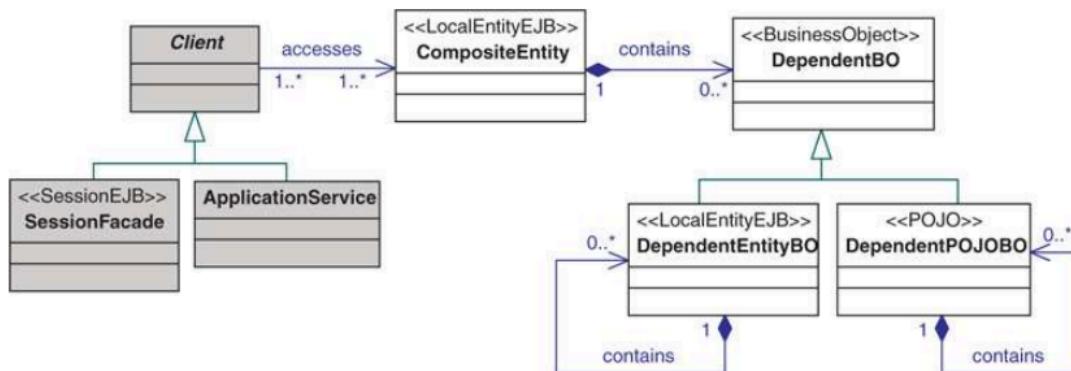
Ahora, con los objetos de negocio sí que usamos punteros, y esta es una gran diferencia con los transfers; además de que implementan reglas de negocio.

12. Entidad Compuesta

Sirve para implementar objetos del negocio persistentes como entidades JPA. Agrega un conjunto de objetos del negocio relacionados con entidades JPA de grano grueso.

Ventajas:

- Reduce la dependencia del esquema de la base de datos
- Incrementa la granularidad de los objetos



¿Cuál es la diferencia entre un objeto de negocio y una entidad compuesta?

Los objetos de negocio es la forma abstracta y la entidad compuesta es la implementación de los objetos de negocio en un marco.

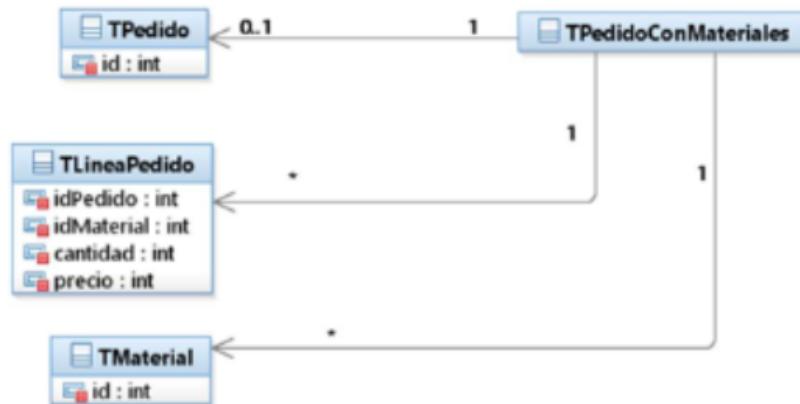
13. TOA

Se utiliza el TOA para construir un modelo de aplicación como un objeto transferencia compuesto.

Agrega diferentes objetos transferencia provenientes de distintos componentes y los devuelve al cliente.

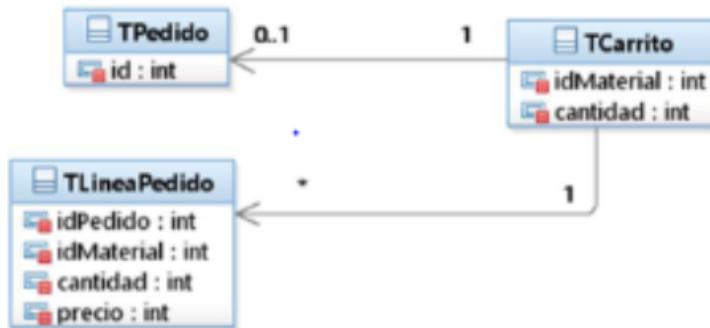
Se usa en los carritos normalmente.

Ejemplo:



Para leer un pedido con sus materiales usaremos un TOA, pues nuestra estructura de transfers no lo permite.

Ejemplo con TOA para un carrito:



14. Manejador de lista de valores

Permite a un cliente remoto iterar sobre una lista de resultados grande, proporcionándole un mecanismo de búsqueda e interacción eficientes, y manteniendo los resultados en el servidor.

Ventajas:

- Mejora el rendimiento de la red
- Cachea resultados de la búsqueda
- Proporciona partición en capa
- Evita sobrecarga

Desventajas:

- Puede ser costoso
- Puede incluir datos desactualizados

15. Fachada de sesión

Se utiliza cuando se desea exponer componentes de negocio y servicios a clientes remotos.

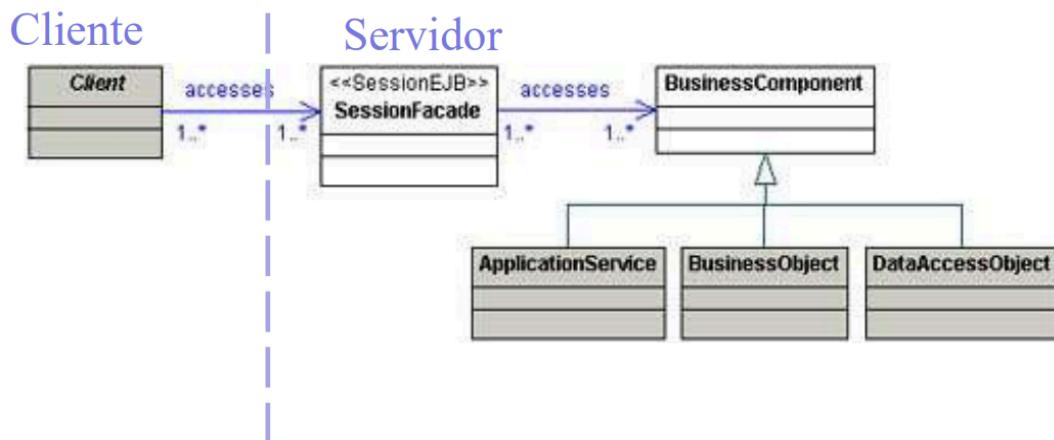
De esta forma, el cliente accede a ella en vez de acceder directamente a los componentes de negocio.

Centraliza peticiones y puede filtrar y hacer controles de seguridad.

Es el patrón que se usa para acceder remotamente a un SA mediante RPC.

Ventajas:

- Introduce una capa que da servicios a clientes remotos
- Promueve la división en capa
- Reduce el acoplamiento
- Centraliza la gestión de seguridad



16. Localizador del servicio

Implementa toda la lógica de petición para que el cliente pueda localizar los servicios de aplicación. Los clientes lo pueden cachear para que no tengan que buscarlo.

Explicación Formal:

Se usa para localizar de manera transparente componentes de negocio y servicios de una manera uniforme.

17. Delegado del negocio

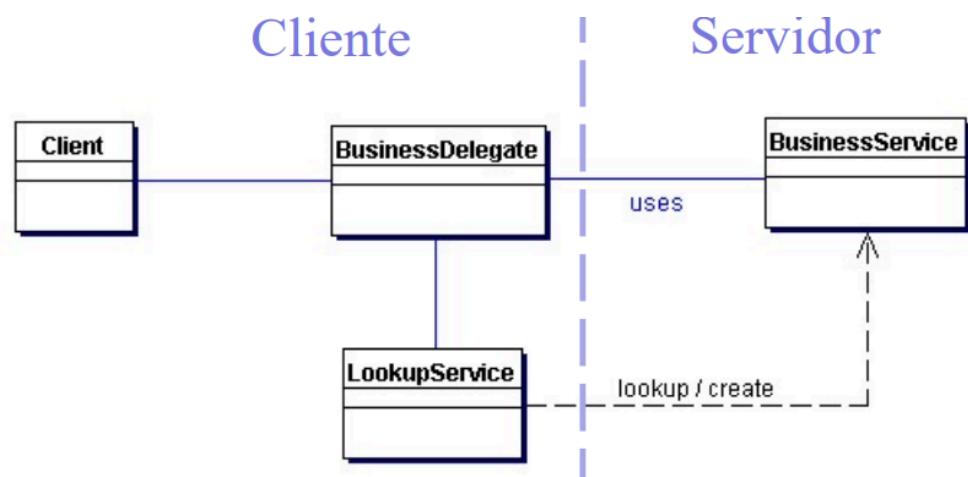
Evita que los clientes tengan que tratar con detalles de acceso a componentes distribuidos en una aplicación multicapa.

Oculta los detalles de implementación del servicio de negocio, como la búsqueda y acceso.

Es decir, es el representante de la lógica remota en el cliente y le oculta si estoy interactuando con un objeto de mi máquina o en remoto.

¿Qué relación tiene con un localizador del servicio?

Los delegados de negocio usan localizadores de servicio para acceder al objeto que va a responder la petición.



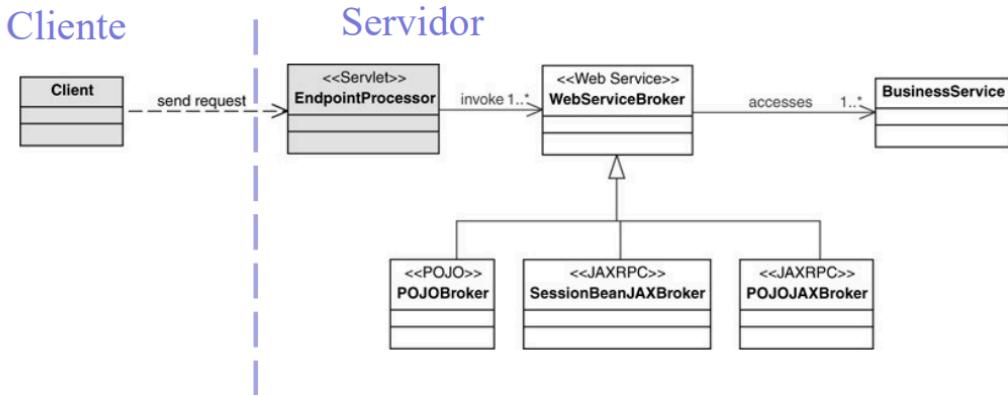
Capa de integración:

18. Agente de servicio web (broker)

Se usa para proporcionar acceso a uno o más servicios usando XML y protocolos web.

(Básicamente es un objeto que ponemos por el lado del servidor para hacer visible un servicio de aplicación como servicio web.)

Es el patrón que se usa para acceder remotamente a un SA mediante servicios web.



19. Activador de servicio

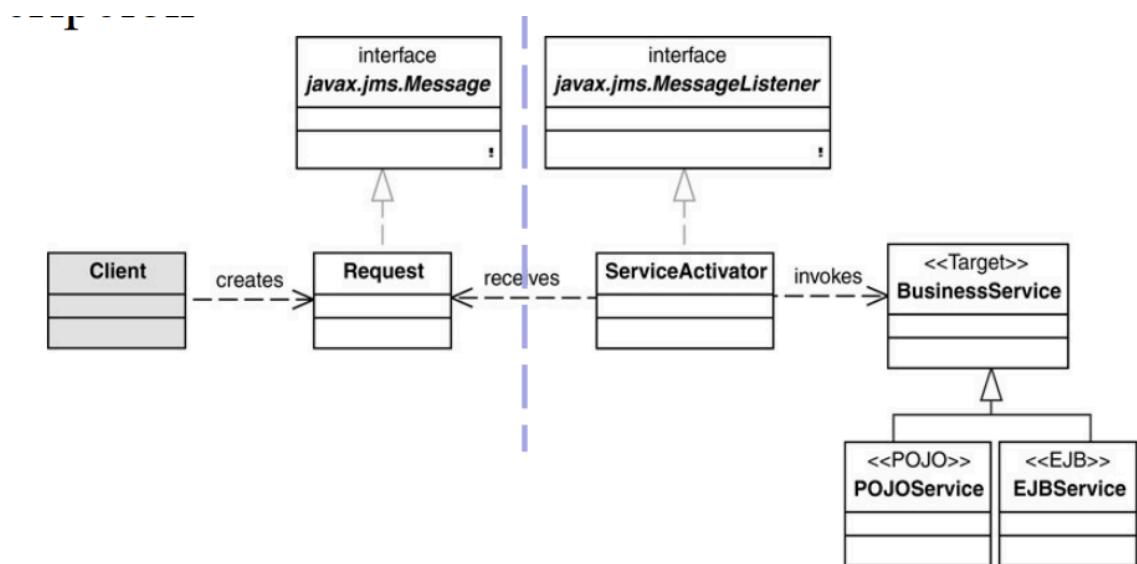
Sirve para recibir peticiones asíncronas e invocar uno o más servicios de negocio.

Es decir, actúa como un intermediario que recibe los mensajes de los clientes, los procesa y luego invoca a los servicios de negocio, todo de manera asíncrona.

También se emplea cuando se desea integrar mensajería publicar/suscribir y punto a punto.

Integra JMS (el servicio de mensajería de Java).

Es el patrón que se usa para acceder remotamente a un SA de forma asíncrona.



¿Qué relación tiene un delegado del negocio con fachada de sesión/agente de servicio web/activador de servicio?

Que el delegado de negocio es el mecanismo utilizado para invocarlos de forma remota

20. DAO

Se usa cuando se desea encapsular la manipulación y acceso a datos en una capa separada.

Te da los objetos que pidas de la capa de recursos en forma de transfer.

```
import java.sql.*;

public class UpdateCar {

    public static void UpdateCarNum(int carNo, int empNo) throws
SQLException {

    Connection con = null;
    PreparedStatement pstmt = null;
    try { con = DriverManager.getConnection("jdbc:default:connection");
    pstmt = con.prepareStatement(
                    "UPDATE EMPLOYEES " +
                    "SET CAR_NUMBER = ? " +
                    "WHERE EMPLOYEE_NUMBER = ?");

    pstmt.setInt(1, carNo);
    pstmt.setInt(2, empNo);
    pstmt.executeUpdate();

    }
    finally {
        if (pstmt != null) pstmt.close();
        if (con != null) con.close();
    }
}
}
```

Usamos prepareStatement para que sea más sencillo escribir la query SQL (se usa en el proyecto).

Por otra parte, hay que cerrar la conexión, y en este caso usamos executeUpdate() porque estamos realizando una actualización.

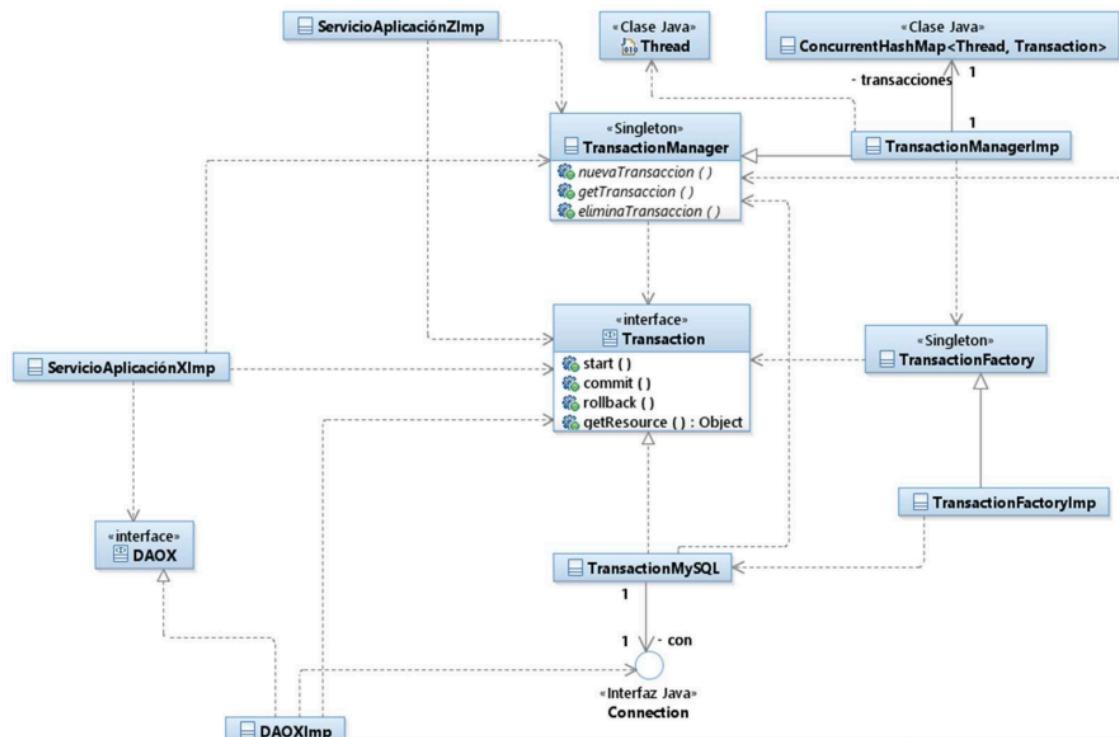
21. Almacén del dominio

Se usa para persistir de manera transparente un modelo de objetos.

Resuelve:

- Persistencia: DAOs (StoreManager) y transfers
- Carga dinámica: proxy (StateDelegate + StateManager)
- Transaccionalidad: transacción (Transaction)
- Conurrencia (PersistenceManager + TransactionManager)

Sin usar PersistenceManager ni StateDelegate ni eso:



El patrón proxy se usa en la carga dinámica de los objetos de negocio y también el stateDelegate es un proxy :)

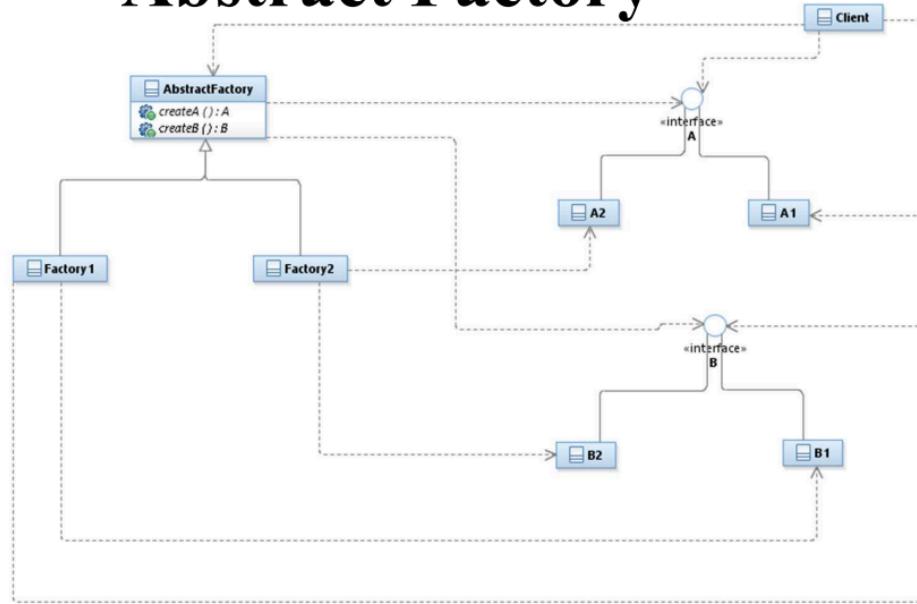
PATRONES GOF

De creación:

22. Factoría abstracta

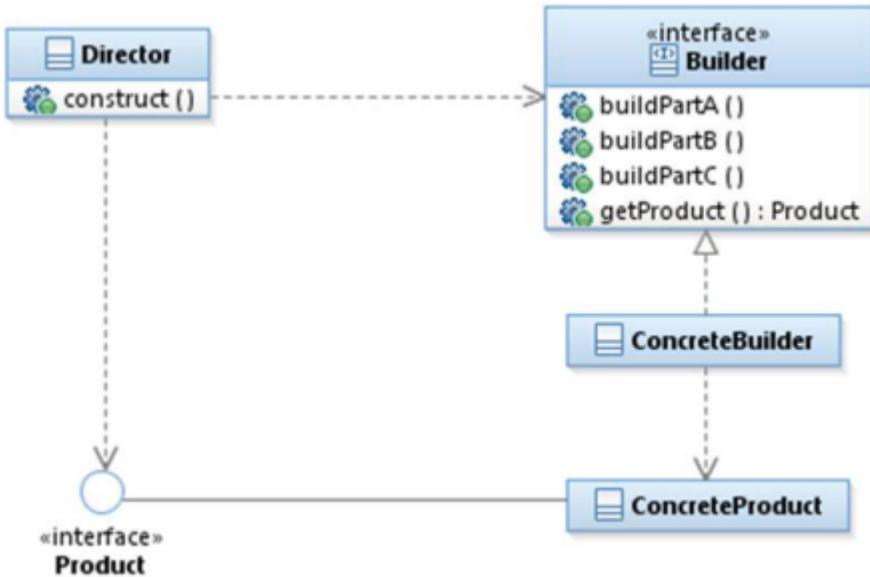
Su objetivo es crear una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas. Una factoría crea objetos devolviendo sus interfaces.

Abstract Factory



23. Builder

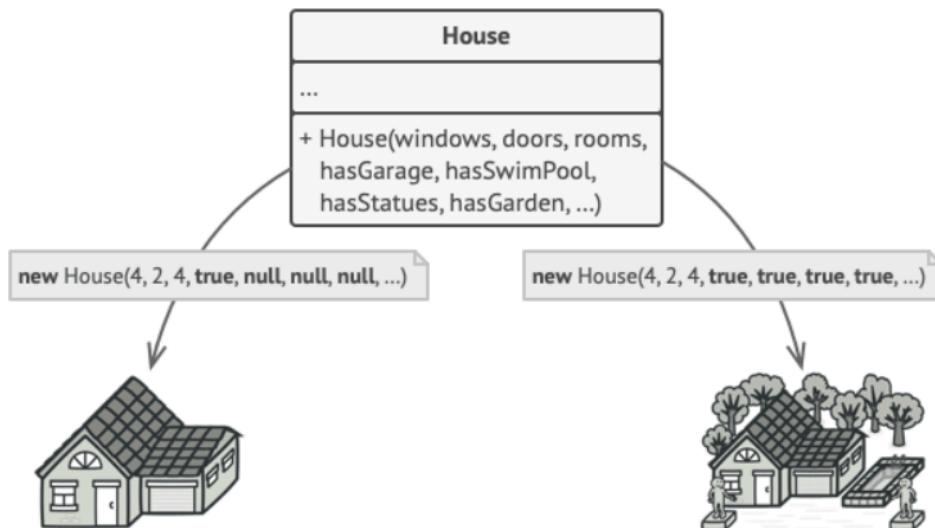
Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones del objeto.



¿Por qué se dice que el proceso de construcción puede crear diferentes representaciones del objeto?

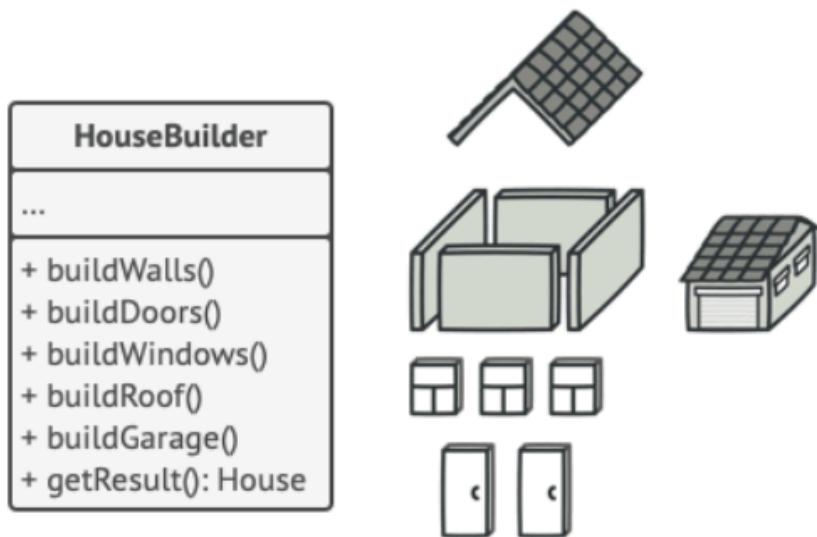
Pongámonos en contexto, queremos construir una casa, pero claro esta casa puede tener jardín, tejado...

Nuestra primera opción sería tener una constructora de la casa con un número ingente de parámetros:



Pero esto tiene un problema, y es que la mayor parte de las veces muchos de los parámetros serán null y quedará una llamada bastante fea y que no nos dejaría dormir por las noches.

Aquí es donde entra este patrón, que sugiere que saques el código de construcción del objeto de su propia clase y lo coloques dentro de objetos independientes llamados constructores, y te permite construir objetos paso a paso.



De esta forma, se van a ir construyendo los objetos según se necesiten, y aunque al fin y al cabo seguirán siendo casas, tenemos una forma común de construir una casa estándar y luego agregarles tejado, garaje....

24. Factory Method

Proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

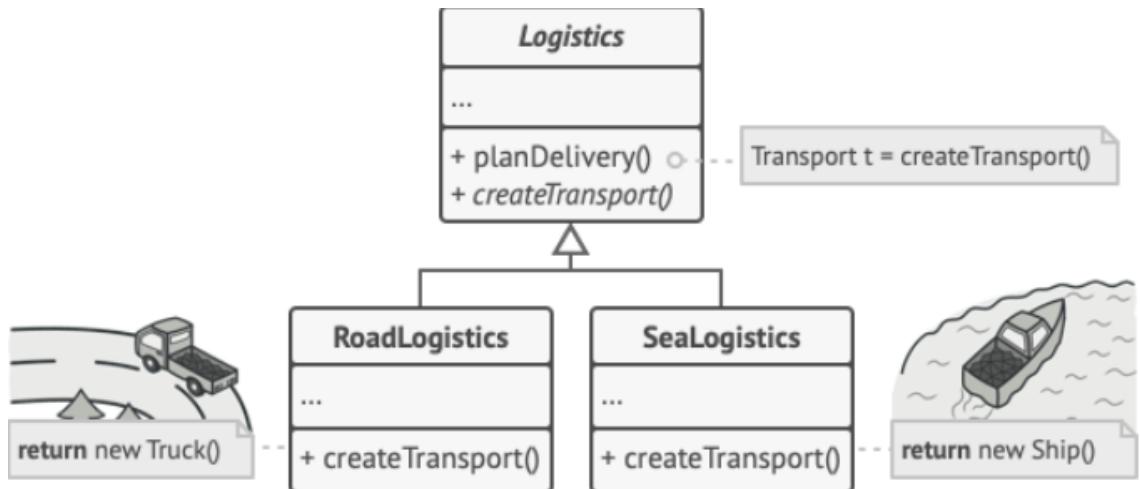
Contexto:

Imagina que estás creando una aplicación de gestión logística. La primera versión de tu aplicación sólo es capaz de manejar el transporte en camión, por lo que la mayor parte de tu código se encuentra dentro de la clase Camión.

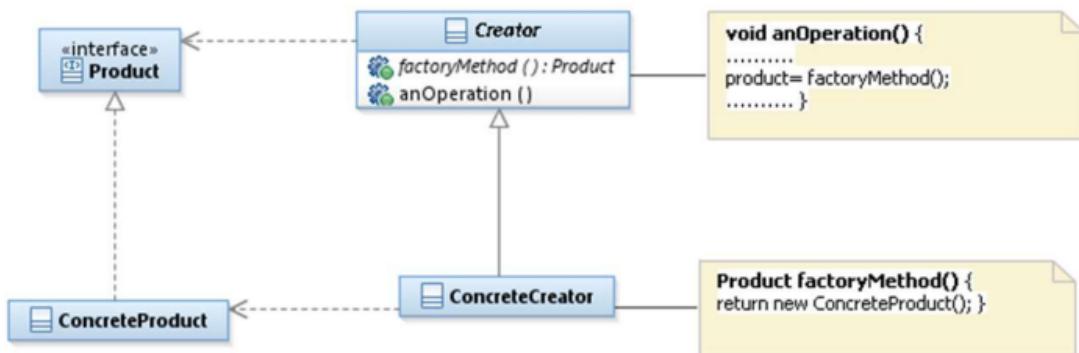
Al cabo de un tiempo, tu aplicación se vuelve super famosa. Cada día recibes decenas de peticiones de empresas de transporte marítimo para que incorpores la logística por mar a la aplicación.

¿Qué pasa ahora? Que tu código está acoplado en la clase Camión, y para añadir los transportes marítimos tendrías que hacer unos cambios brutales en toda la estructura.

Aquí es donde entra el Factory Method, que en vez de hacer un new para crear objetos, ahora vas a llamar a un método fábrica



Estructura:



25. Prototype

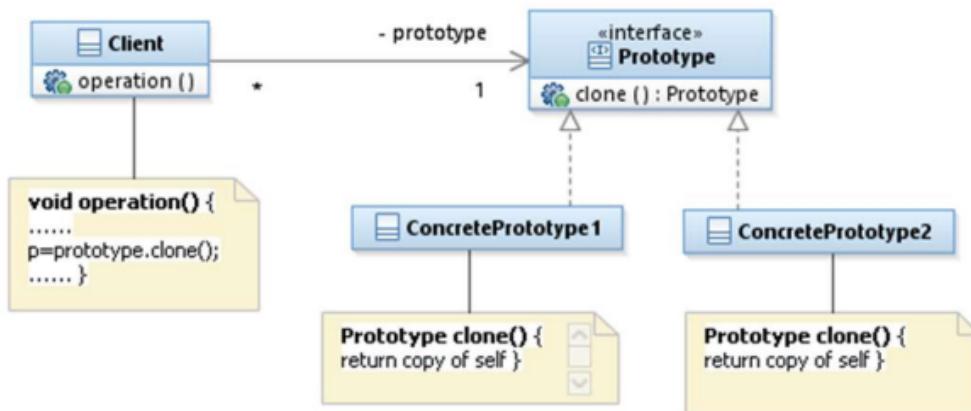
Es un patrón de diseño creacional que nos permite copiar objetos existentes sin que el código dependa de sus clases.

Contexto:

Digamos que tienes un objeto y quieres crear una copia exacta de él. ¿Cómo lo harías? En primer lugar, debes crear un nuevo objeto de la misma clase. Después debes recorrer todos los campos del objeto original y copiar sus valores en el nuevo objeto.

Pero, ¡caray! No todos los objetos se pueden copiar de este modo, porque algunos de los campos de los objetos puede ser privado y además, dado que tienes que conocer la clase del objeto para crear el duplicado, el código se vuelve dependiente de esa clase.

Aquí es donde entra el Prototype, que delega el proceso de clonación a los propios objetos que están siendo clonados, declarando una interfaz común para todos los objetos que van a soportar la clonación.

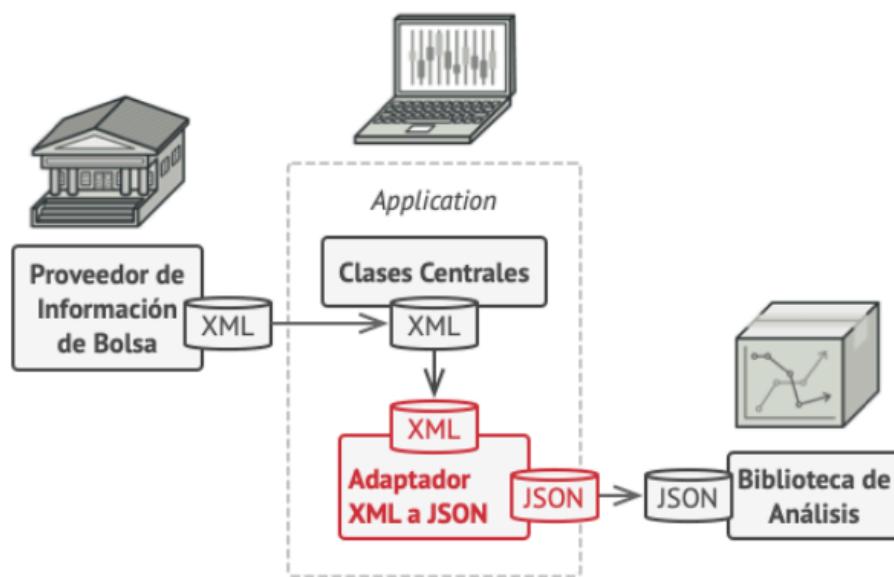


Estructurales:

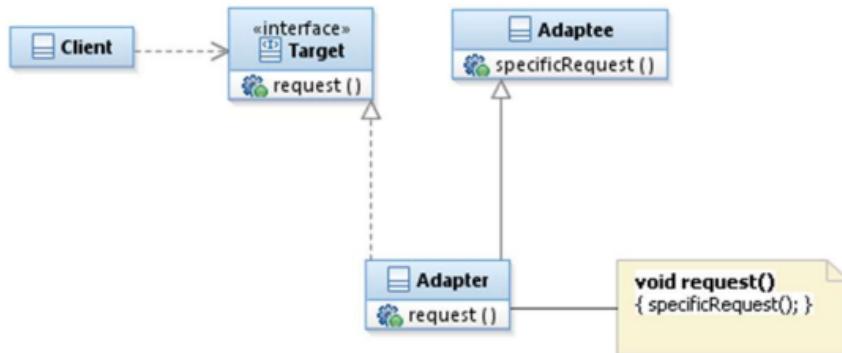
26. Adapter

Convierte la interfaz de una clase en otra interfaz que es la que esperan los clientes, y permite que cooperen clases con interfaces incompatibles.

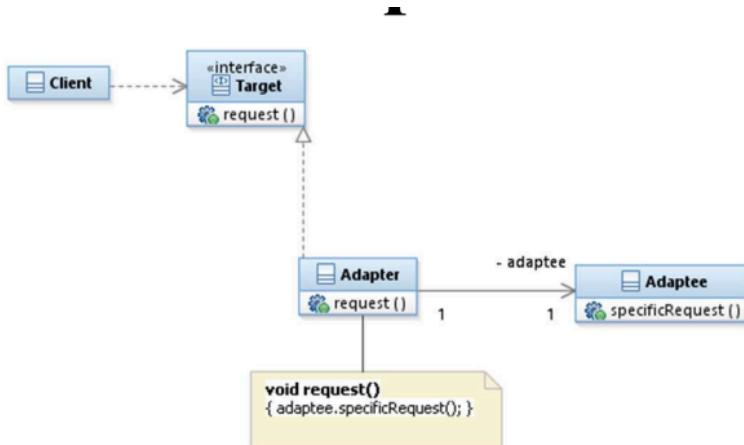
Contexto:



Esquema adaptador de clases:



Esquema adaptador de objetos:

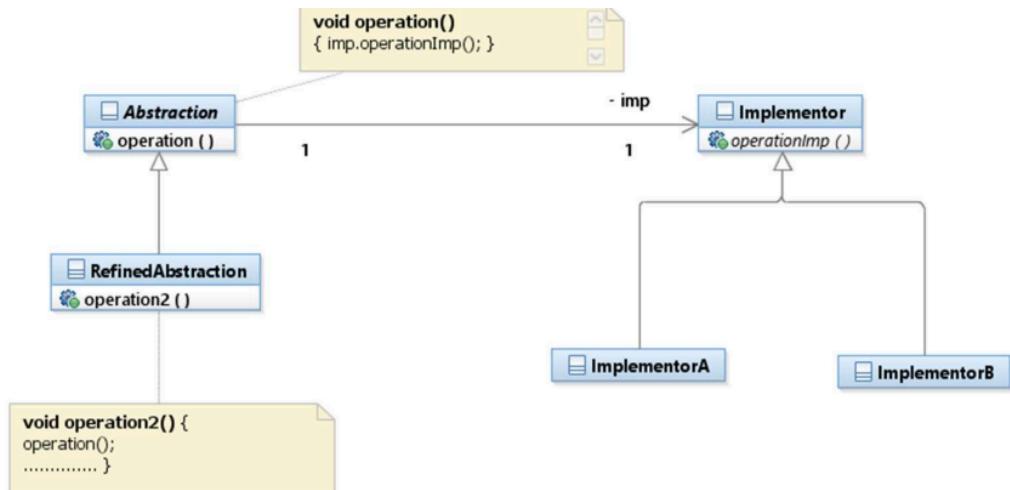


¿Se usa el patrón Adapter en el proyecto?

No se usa *ad hoc*, pero sí que se puede llegar a medio considerar que Transaction hace la función de llegar a adaptar al SA para realizar acciones contra el SGBDR

27. Bridge

Permite dividir una clase grande (o un grupo de clases estrechamente relacionadas) en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.



Ventajas:

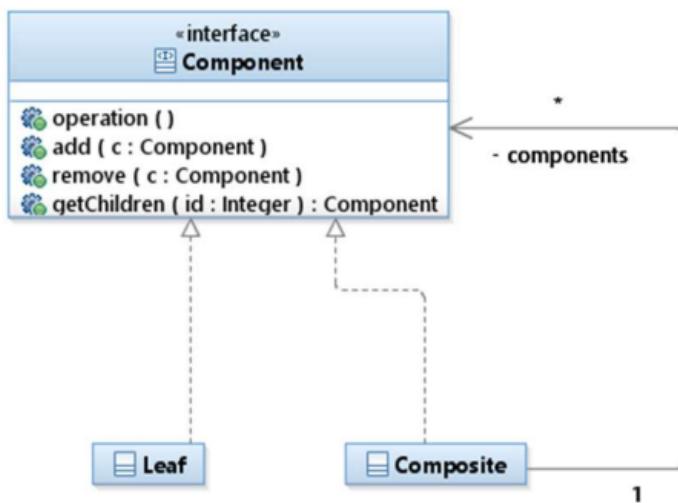
- Desacopla la interfaz de la implementación
- Oculta detalles de implementación a los clientes

Desventajas:

- Nuevas subclases de la abstracción podrían necesitar nuevas subclases de la implementación

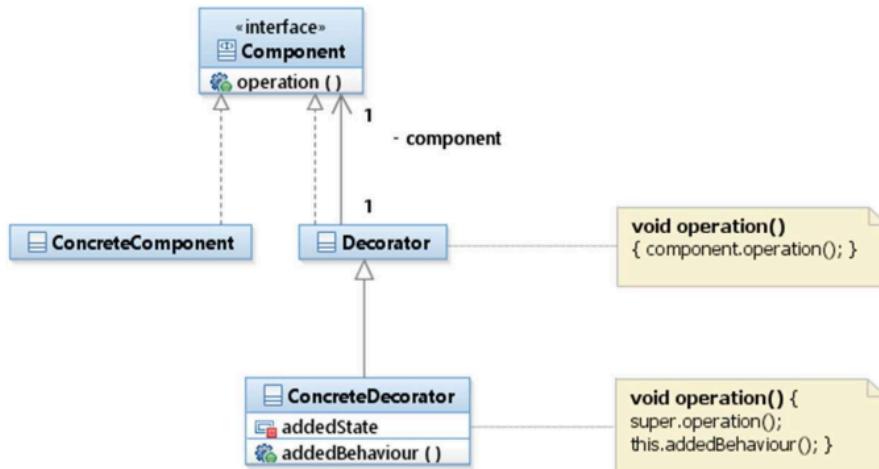
28. Composite

Compone objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.



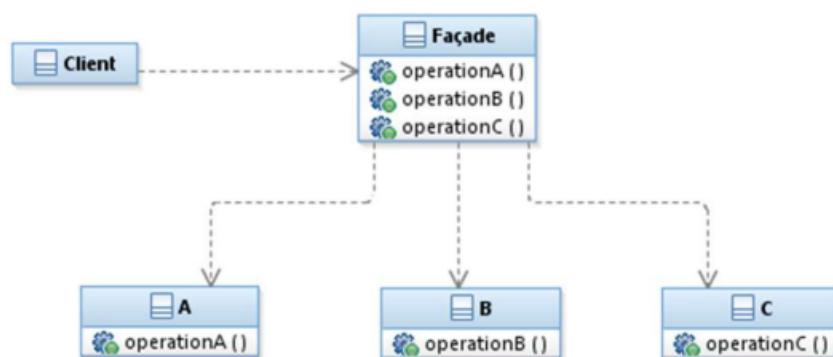
29. Decorator

Asigna responsabilidades adicionales a un objeto dinámicamente, proporcionando una alternativa flexible a la herencia para extender la funcionalidad



30. Façade

Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Hace que sea más fácil de usar.



¿Se usa el patrón Fachada en los proyectos de la asignatura?

No, porque nuestros DAOs y SAs tienen la granularidad correcta.

31. Flyweight

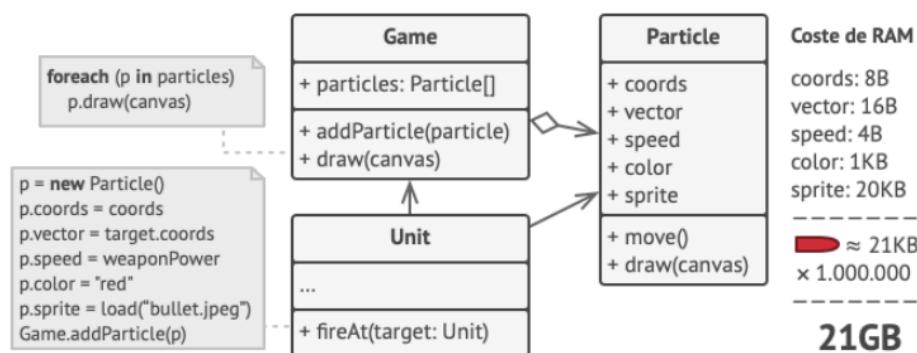
Comparte objetos de grano fino para permitir la existencia de un gran número de objetos de forma eficiente.

Contexto:

Suponemos que tenemos un juego en el que los jugadores se disparan entre sí.

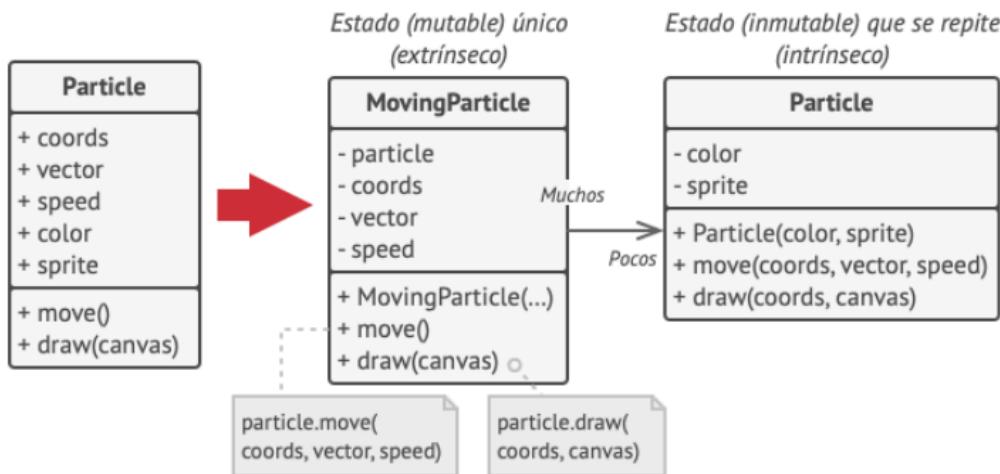
En tu pc va flying, pero se lo pasas a un amigo (su ordenador no es tan bueno) y a él se le paraba tras escasos minutos jugando.

Depurando te das cuenta de que cada partícula (como una bala), estaba representada por un objeto separado que contenía gran cantidad de datos. En el momento en que la masacre estaba asegurada, se estaban creando tantas balas que al final la RAM petaba.

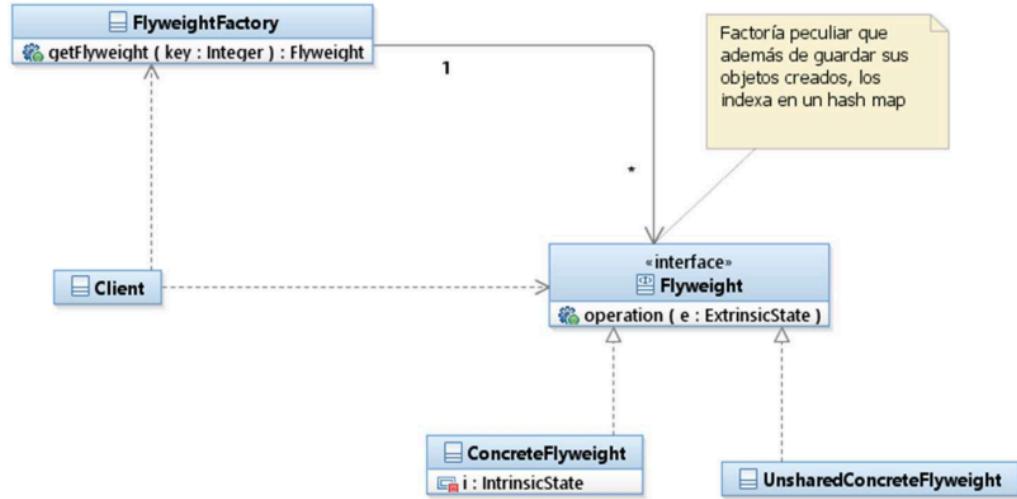


Solución:

Observando la clase Partícula te das cuenta de que color y sprite ocupan mucho en RAM y que además va a ser casi igual en todas las balas por ejemplo, por lo que tenemos que diferenciar entre **estado extrínseco (aquel que es alterable)** y **estado intrínseco (aquel que se mantiene constante)**.



Aquí es donde el patrón Flyweight te dice que almacenes en la clase particle solamente el estado intrínseco, y muevas el extrínseco a otra que dependa de él, normalmente el objeto contenedor.

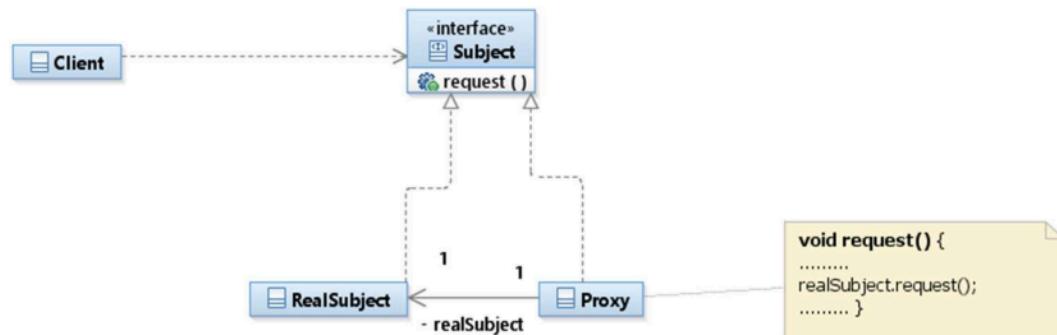


32. Proxy

Proporciona un representante o sustituto a otro objeto para acceder a éste.

Hay tres tipos:

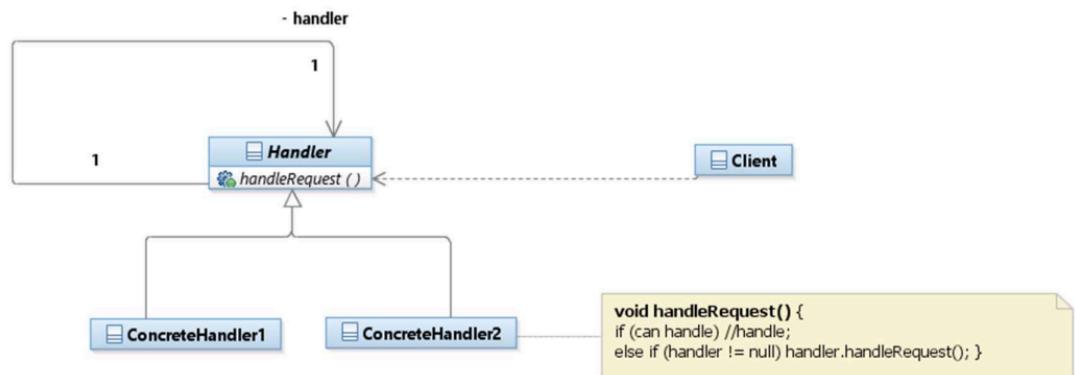
- Remoto
- Virtual
- De protección



De comportamiento:

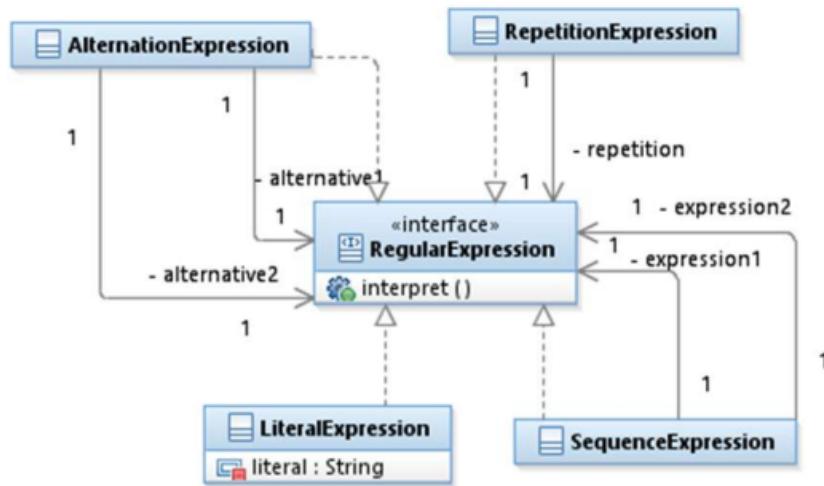
33. Cadena de responsabilidad

Es un patrón de diseño de comportamiento que te permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.



34. Interpreter

Dado un lenguaje, define una representación de su gramática junto con un intérprete que usa dicha representación para interpretar sentencias del lenguaje.

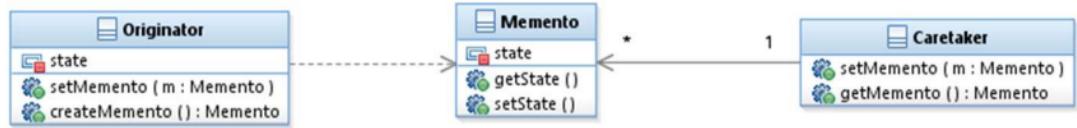


35. Memento

Te permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación.

Se debe usar cuando haya que guardar una instantánea del estado de un objeto (para que se pueda volver a este estado).

Además, hay que tener en cuenta que poner una interfaz directa para solucionar este problema podría romper la encapsulación y mostrar detalles del objeto.



¿Por qué no tiene sentido usar memento en aplicaciones empresariales?

Para empezar, las aplicaciones empresariales necesitan persistir los datos, algo de lo que se ocupa las bases de datos, por lo que no te interesaría guardar estados de objetos en mementos (por ejemplo para deshacer).

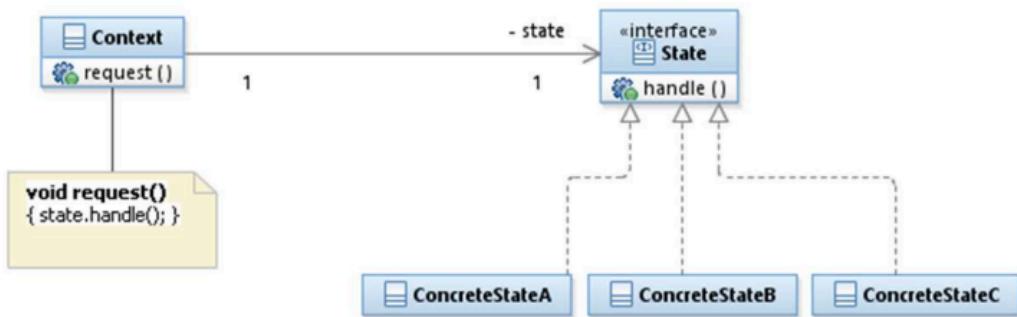
Pero, si aún así se decidiera guardarlos, tendríamos otro problema, y es que al almacenar estados complejos de objetos, ocuparía una gran cantidad de memoria y la aplicación iría muy lenta.

36. State

Permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara su clase.

Está estrechamente relacionado con la máquina de estados.

Por ejemplo, imagina que tienes la clase Documento con 3 estados: borrado, publicado y en cola; pues para gestionar esto deberías usar un State.



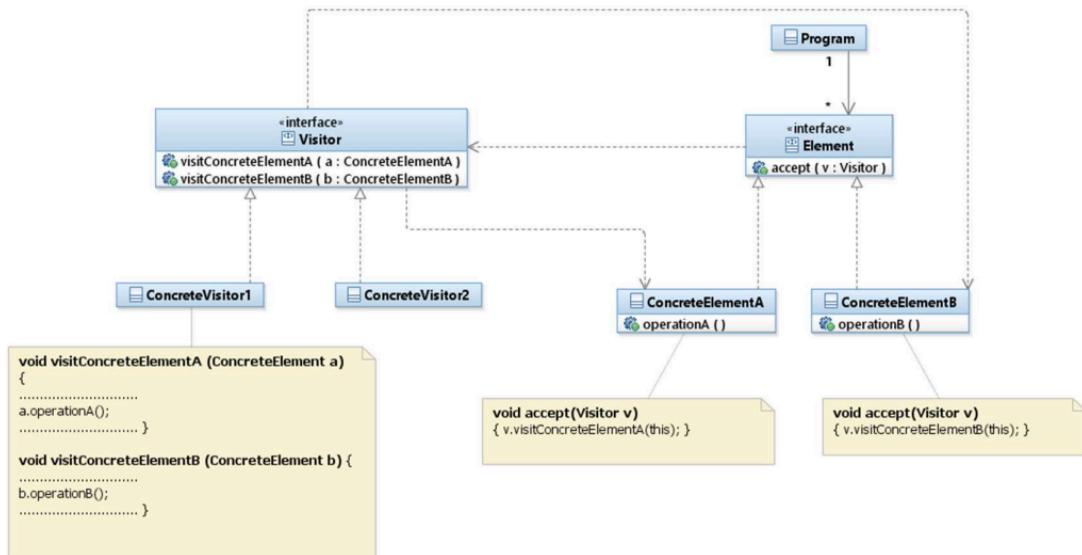
37. Template Method

Define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura.

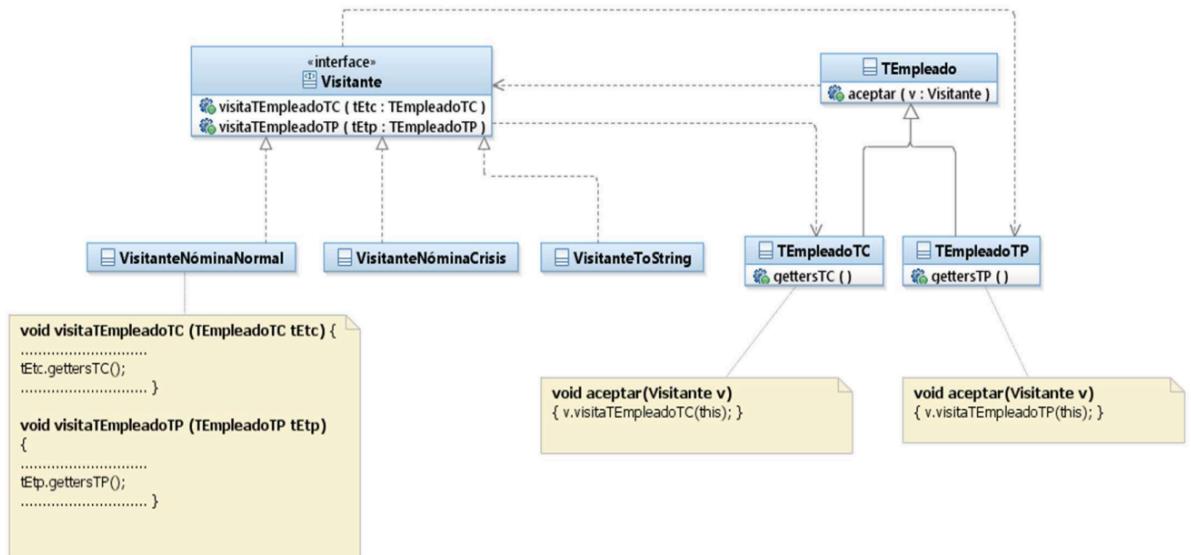


38. Visitor

Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.



Ejemplo:



(Lo único que faltaría del temario sería lo de transaccionalidad y gestión de concurrencia y el resto del examen es parte práctica conforme al proyecto).

Agradecimientos y fuentes:

Transparencias de Antonio Navarro (MS).
 Refactoring.guru.
 Apuntes de @cocodrift en Wuolah.

Agradecimiento especial al Señor Don Javier Aceituno.

Una entrega más, y con absoluto placer:

David Castro García.

