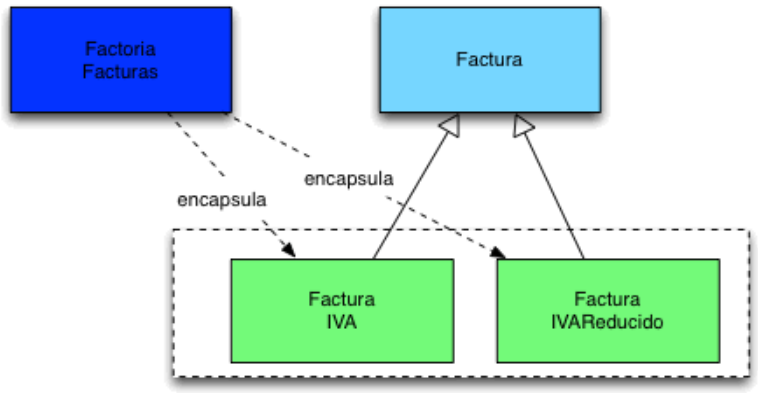
Fecha: 15/marzo/2018 Nombre: Mario Iván Montes Vidal Clave única: 156167

Análisis y Diseño de Sistemas de Información – Tarea 6

1) Ejemplificación y descripción de algunos patrones con código real

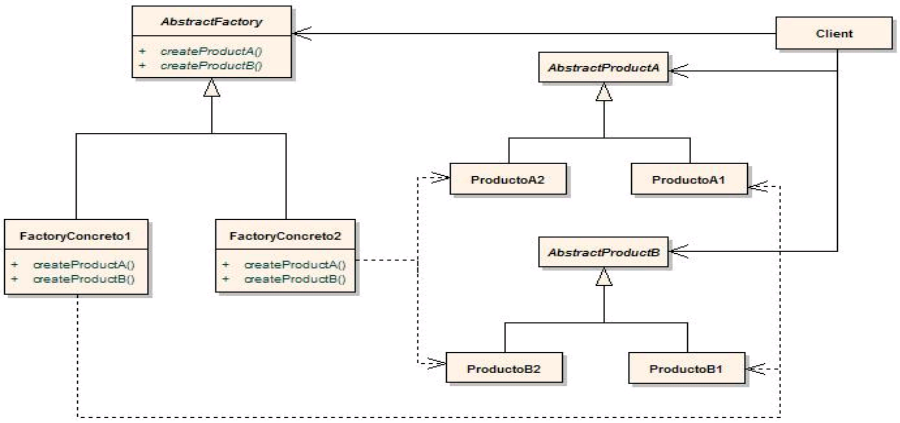
Factory

Es un patrón de diseño creacional, es decir, proporciona lo necesario para crear una jerarquía de clases, pero sin la necesidad de exponer la lógica de creación al usuario. Todos los objetos dentro del código son referenciados a través de una interfaz común. Este patrón permite al usuario trabajar con una clase, sin necesidad de preocuparse por las subclases que hereda la clase principal. Es flexible al aislar las clases y al momento de actualizar atributos en ellas.



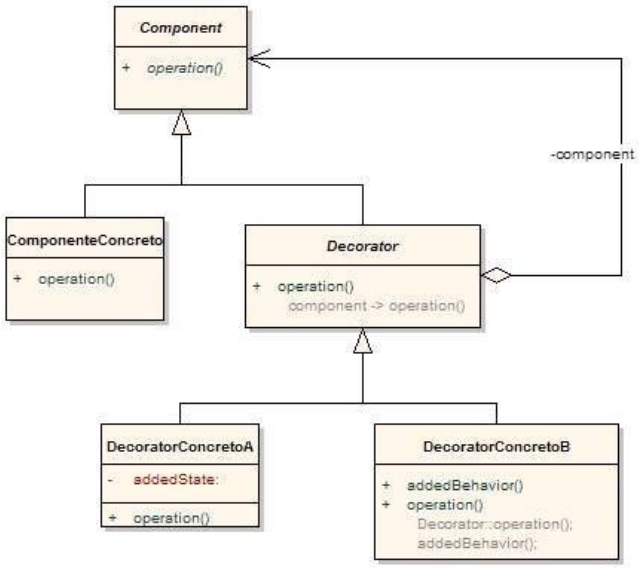
Abstract Factory

Este patrón funciona igual que el patrón *factory*, solo que es mucho más extenso. Se usa cuando se requiere de un sistema con una jerarquía de clases bastante grande; el usuario, mediante una interfaz, tiene acceso a conjuntos de familias u objetos dentro de la jerarquía, pero sin especificar cual es el objeto concreto. Al igual que el patrón *factory*, este patrón proporciona más flexibilidad al aislar las clases y facilita la actualización de atributos en toda la familia de clases.



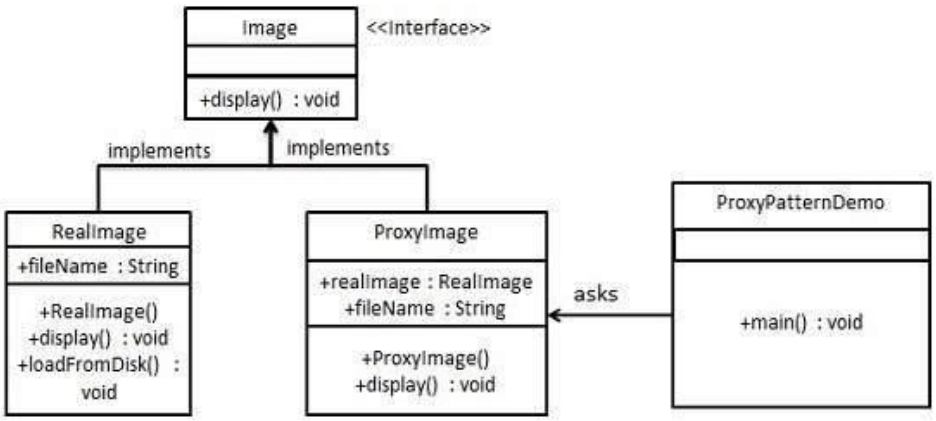
Decorator

Este parón estructural permite al usuario agregar nuevas funcionalidades a un objeto existente sin alterar su estructura. Su estructura consiste en la creación una clase *decorator* que envuelve la clase original y proporciona una funcionalidad adicional manteniendo la firma de los métodos de clase intactos.



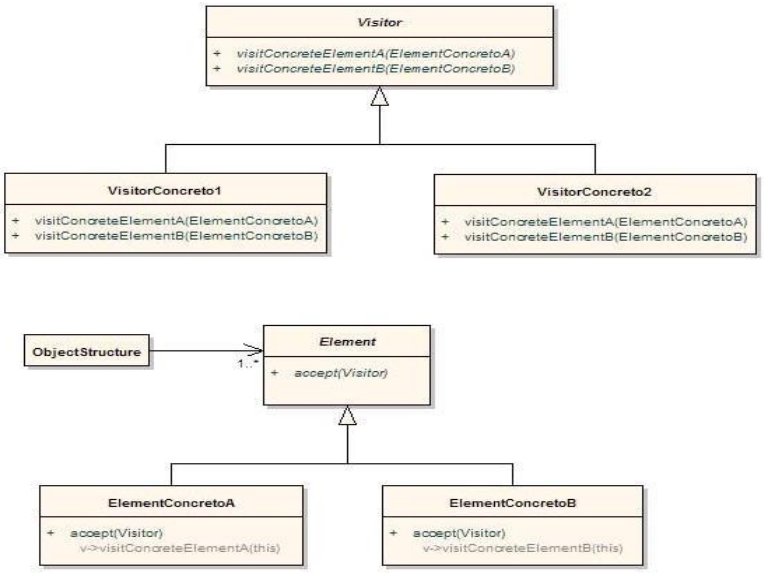
Proxy

Este patrón de tipo estructural permite crear objetos y clases normales, pero agrega la funcionalidad de estos en otra clase, es decir, puedo controlar la funcionalidad de cualquier objeto, instanciado de acuerdo con el patrón, desde otra clase, y esta a su vez, funciona como puente entre el objeto y el usuario.



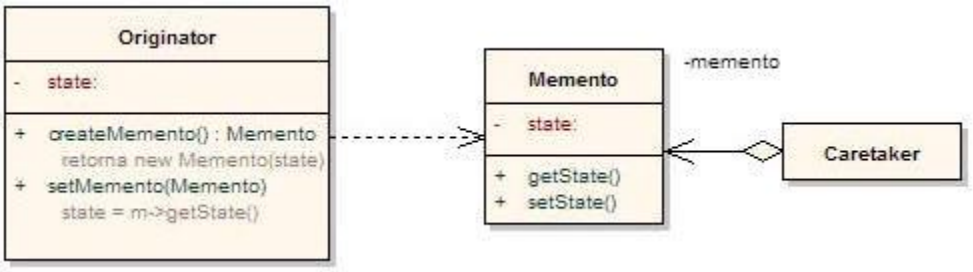
Visitor

Este patrón de comportamiento permite añadir funcionalidades a una clase, sin tener que modificarla, mediante el uso de una clase *visitor* que cambia el algoritmo de ejecución de otra clase de elemento. De esta forma, el algoritmo de ejecución del elemento puede variar a medida que varía el visitante. Según el patrón, el objeto de elemento tiene que aceptar el objeto de visitante para que el objeto de visitante maneje la operación en el objeto de elemento. Este patrón es muy parecido al patrón *decorator*, la principal diferencia es que aquel agrega funcionalidades a objetos, mientras que este las agrega a clases.



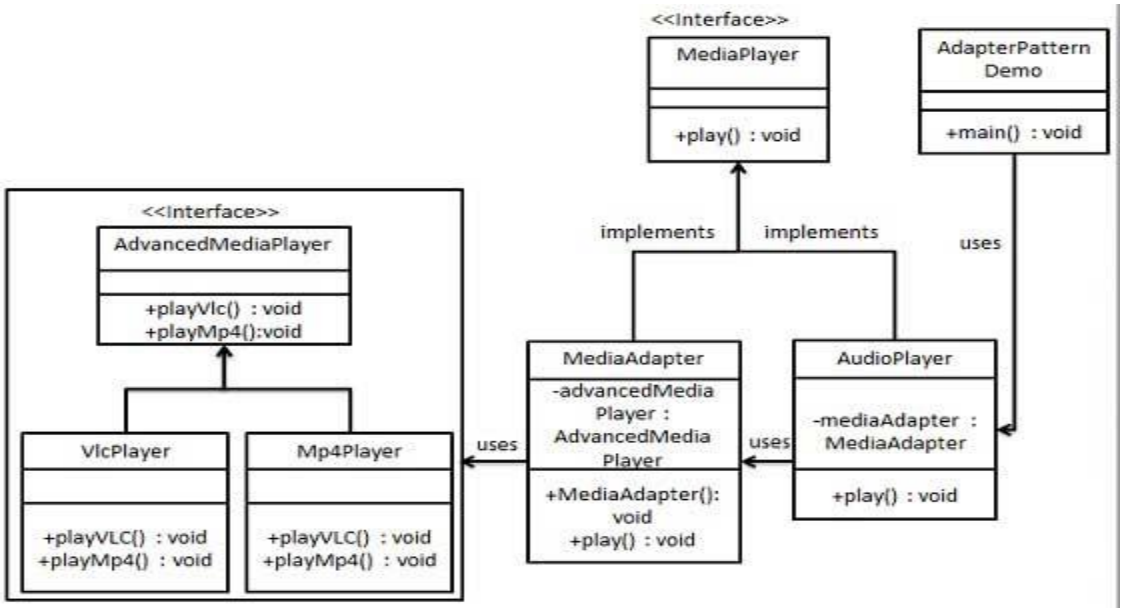
Memento

Este patrón es una especia de "recordador" de los estados anteriores de los objetos, es decir, permite capturar y exportar el estado interno de un objeto para que luego se pueda restaurar una vez que ha sido actualizado, todo gracias a tres clases principales; el estado de un objeto que se restaurará, guardado en la clase *memento()*; un *originator()* que crea y almacena estados en objetos *memento()*; y finalmente, el objeto *caretaker*, el cual, es responsable de restaurar el estado del objeto guardado en *memento()*. Este patrón es muy útil cuando, al trabajar con aplicativos, se desea "hacer" y "deshacer" de manera sencilla y sin ningún temor a pérdida de estados.



Adapter

Este patrón funciona como un puente entre dos o más interfaces incompatibles. Gracias a su característica de ser un patrón estructural, permite combinar la capacidad de dos o más interfaces independientes. Su estructura implica una sola clase (clase principal) que es responsable de unir funcionalidades de interfaces independientes, es decir, si una interfaz requiere datos provenientes de otras interfaces, o la propia estructura del programa combina varios procesos entre clases, la clase principal es la responsable de gestionar que el resto de las clases puedan comunicarse entre sí.



2) *Ensayo del video "Managing State with RxJava by Jake Wharton"*

Funcionamiento y ventajas de Reactive X

Jake Wharton nos introduce a las complejidades que rodean a Reactive X, una API que facilita el manejo de flujos de datos y eventos, a partir de una combinación de varios patrones de diseño; entre ellos Observer e Iterator. Wharton nos habla sobre la programación asíncrona, esto es, la estructuración de los programas que permitan ejecutar llamadas antes de que otras llamadas previas hayan concluido; esto permite acelerar el proceso de ejecución considerablemente. Según Wharton, la opción de uso de este tipo de programación se presenta cuando se deben manipular datos desde el back-end, escritos en un sistema de archivos, guardados en la red o en una base de datos. Cuando uno programa pensando en los posibles errores, trata de que su código sea lo más optimo posible, esto con él fin de que ningún error pueda perjudicar la aplicación; muchas veces el propio programador se muestra optimista de que todo salga bien, esto es imaginarse siempre el mejor de los casos. Sin embargo, como dice Wharton, existen problemas que están fuera del control del programador, tales como fallas de red o actualizaciones simultáneas del sistema; cuando estas se presentan, la aplicación se ralentiza y corre el riesgo de fallar al momento de procesar los

datos. Una solución parcial al problema sería el uso de un *listener* (escuchador) como alerta en caso de éxito o fracaso de los procesos; sin embargo, esto a la larga solo dificulta exponencialmente el problema, ya que, si trabajamos por ejemplo con dispositivos móviles, estos son eminentemente asíncronos; todo el tiempo un móvil puede estar ejecutando actividades simultáneas, las cuales, al saturarse, pueden detener las aplicaciones e incluso trabar el dispositivo.

La pre-solución consiste en quitar responsabilidades a nuestro código conectando directamente los componentes (interfaz gráfica, base de datos, red etc.) entre sí, sin necesidad de usar un bus. Comúnmente, dentro del código de las aplicaciones existen partes llamadas código imperativo, el cual se encarga de manipular los datos y luego decidir qué hacer con ellos. Wharton sustituye este código con uno más elaborado y fragmentado en cinco secciones principales: empuje (*push-based updates*), cuando los datos son actualizados inmediatamente el programa los "empuja" en la base de datos; declarativo (*declarative threading*), esta instrucción envía los datos al *flatMap()*, el cual transforma los elementos en observables de una sola secuencia a realizar según la actividad correspondiente; error de línea (*erase error-handing*), esta instrucción controla los posibles casos de error y despliega las acciones a realizar; llamadas especializadas, estas nos dicen si tuvo éxito o no la acción; y finalmente, cancelación de ciclo, este simplemente termina el ciclo de ejecución de una actividad. Una de las ventajas claras de Reactive X son su falta de errores terminales, es decir, si se hace una solicitud y esta falla no se desplegará el mensaje de error, sino que habilitará la interfaz gráfica para nuevo ingreso de datos. Aplicando esta pre-solución nos encontramos con una aplicación que funciona de la siguiente manera: primero, los datos de entrada son transformados en eventos (`Observable<SubmitUiEvent>`) y enviados al *flatMap()*, esto mientras la interfaz gráfica es deshabilitada; segundo; el nuevo código mencionado anteriormente se encarga de ordenar por prioridad las actividades y enviar los observables, recién creados, al gestor (red, base de datos etc.) correspondiente; tercero, ya que en los datos fueron almacenados o actualizados, se emite una notificación de "en curso" donde se está evaluando el caso de éxito o fracaso de la operación; cuarto, para regresar la respuesta, se crea un *subscriber* en el cual se contienen los resultados de la operación, estos son enviados de regreso a la interfaz gráfica.

Ahora, ¿qué pasa cuando se tiene múltiples peticiones en la interfaz gráfica? Como el código está diseñado para crear eventos a partir los datos de entrada, siempre se crea un observable por cada evento que surja a partir de la petición del usuario; a través de un *merge*, todos los observables se concatenan en uno solo, lo mismo que los resultados que regresan a la interfaz después del proceso. Wharton concluye su presentación recalcando que este modelo no necesariamente requiere de una interfaz o de un gestor para funcionar, si quitamos cualquiera de las dos partes el código sigue funcionando, gracias a su fragmentación, por lo que se presta para realizar pruebas en caso de que no se cuente con alguno de los dos componentes. Este estilo de programación soluciona el problema de posibles errores secundarios y fallas en la red durante el proceso; posee un código lógicamente fragmentado pero muy compacto.

3) Ejemplificación y descripción de una expresión lambda en Java

Expresiones lambda

Las expresiones lambda son funciones compuestas por bloques de código ejecutable, definido en una interfaz, pero no implementado, mediante las cuales se pueden referenciar métodos anónimos o métodos sin nombre, lo que nos permite escribir código más claro y conciso que cuando usamos clases anónimas. Una expresión lambda se compone de: un listado de parámetros separados por comas y encerrados en paréntesis, por ejemplo: (a,b); el símbolo de flecha hacia la derecha: >; y finalmente, un cuerpo que puede ser un bloque de código encerrado entre llaves o una sola expresión.

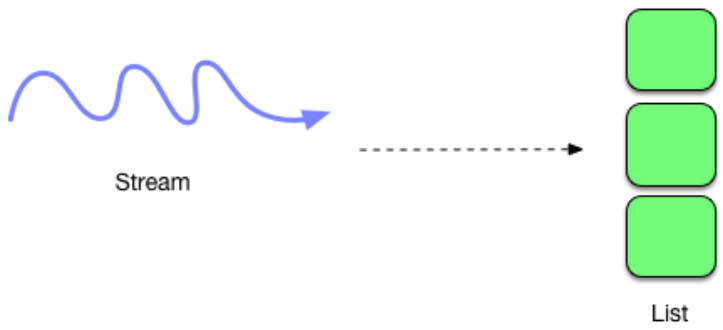
Estructura

```
(argumentos)->{cuerpo} //Estructura de una expresión lambda.
(int a, int b) -> a >b; //Ejemplo: dados dos enteros a y b, se comparan si a es mayor que b.
(int a, int b) -> System.out.println(a + b); return a + b;}
//Dados dos entresos ay b, la función imprime la suma (a+b) y también regresa el resultado.
```

4) Ejemplificación y descripción de un Stream (map-reduce) en Java

Stream

Streams son medios utilizados como lectores de datos de una fuente, para ser escritos en una dirección de destino (archivos, bases de datos, memoria etc.); caracterizan por ser unidireccionales, es decir, se utilizará solo para leer, solo para escribir, pero no ambas acciones al mismo tiempo. La mayoría de las instrucciones para leer datos en los lenguajes de programación poseen una entrada (input) y una salida (output), la estructura de los stream se relaciona directamente con la fuente de destino.



NOTA: La información mostrada en los incisos 1), 3) y 4) proviene de proyectos reales hechos en Maven, anexados junto con este documento. También se encuentran en el GitHub

Bibliografía consultada:

https://www.arquitecturajava.com/usando-el-patron-factory/
https://translate.google.com.mx/translate?hl=es&sl=en&u=https://www.tutorialspoint.com/design_pattern/&prev=search
https://translate.google.com.mx/translate?hl=es&sl=en&u=https://www.tutorialspoint.com/design_pattern/&prev=search
http://www.oracle.com/technetwork/es/articles/java/expresiones-lambda-api-stream-java-2633852-esa.html
http://migranitodejava.blogspot.mx/2011/05/abstract-factory.html
http://migranitodejava.blogspot.mx/2011/06/decorator.html
https://migranitodejava.blogspot.mx/2011/06/visitor.html
http://migranitodejava.blogspot.mx/2011/06/memento.html