DOCUMENTO FINAL
PROGRAMACION AVANZADA

NESTOR CASTRO

JAVIER BECERRA

DOCENTE

INGENIERO DARWIN MARTINEZ

UNIVERSIDAD CENTRAL

FACULTAD DE INGENIERIA

DEPARTAMENTO DE INGENIERIA DE SISTEMAS

JULIO 2015

BOGOTA D.C.

PATRONES DE DISEÑO

Los patrones de diseño son un conjunto de prácticas de óptimo diseño que se utilizan para abordar problemas recurrentes en la programación orientada a objetos.

El concepto de patrones de diseño fue el resultado de un trabajo realizado por un grupo de 4 personas (Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, conocidos como "la pandilla de los cuatro") que se publicó en 1995 en un libro titulado "Patrones de diseño: Elementos de software orientado a objetos reutilizables" en el que se esbozaban 23 patrones de diseño.

Un patrón de diseño puede considerarse como un documento que define una estructura de clases que aborda una situación particular.

Dentro del curso hemos visto los siguientes patrones los cuales a continuación describiremos en el por qué se utilizaron o no dependiendo del desarrollo del problema planteado.

PATRON MVC

Problema: Acoplamiento de la solución del problema con la visualización e intervención con el usuario.

Solución: Separa la visualización de la lógica del problema.

Implementación en el proyecto: Si, Es la base de la solución al problema mediante este patrón separamos los datos de la aplicación, la interfaz de usuario, y la lógica de control.

Separamos la capa visual gráfica de su correspondiente programación y acceso a datos, algo que mejora el desarrollo y mantenimiento de la Vista y el Controlador en paralelo, ya que ambos cumplen ciclos de vida muy distintos entre sí.

PATRON VISITOR

Tipo: Patrón de comportamiento

Problema: Proporcionar una forma fácil y sostenible de ejecutar acciones en una familia de clases.

Solución: Una nueva funcionalidad puede ser fácilmente añadida a la jerarquía de herencia original, mediante la creación de una nueva subclase de Visitantes.

Implementación en el proyecto: Si, nos permite realizar operaciones adicionales sobre la jerarquía de clases centralizando los comportamientos permitiendo modificarlos sin cambiar las clases que visita.

PATRON ITERATOR

Tipo: Patrón de comportamiento

Problema: Un objeto agregado, tal como una lista, debería proveer un modo de brindar acceso a sus elementos sin exponer su estructura interna. Más aún, quizás se desea recorrer la lista en diferentes formas, dependiendo de lo que se quiera realizar. Pero, probablemente, la idea no es aumentar la interfaz de la lista con operaciones para recorridos diferentes, aun anticipando los que se necesitarán. Tal vez, también se necesite tener más de un recorrido en la misma lista

Solución: Proporcionar una forma coherente de acceder secuencialmente a los elementos de una colección, independientemente del tipo de colección subyacente.

Implementación en el proyecto: No, no se posee una lista de objetos en un contenedor al que deseemos acceder o recorrer por lo cual se descrto en la solución final.

PATRON MEMENTO

Tipo: Patrón de comportamiento

Problema: Necesidad de restaurar un objeto a su estado anterior.

Solución: Se quiere poder restaurar el sistema desde estados pasados y por otra parte, es usado cuando se desea facilitar el hacer y deshacer de determinadas operaciones, para lo que habrá que guardar los estados anteriores de los objetos sobre los que se opere (o bien recordar los cambios de forma incremental).

Implementación en el proyecto: No, A pesar de ser altamente necesario en cualquier aplicación poder volver a un estado previo en nuestra aplicación no se implementó.

PATRON CADENA DE RESPONSABILIDAD

Tipo: Patrón de comportamiento

Problema: Establecer una cadena en un sistema, para que un mensaje pueda ser manejado en el nivel en el que se recibe en primer lugar, o ser redirigido a un objeto que pueda manejarlo.

Solución: La petición debe ser procesada por los receptores, lo cual quiere decir que, ésta petición queda al margen del uso exclusivo.

Pretendemos dar un mayor detalle y especificación a las peticiones generadas. Las peticiones serán filtradas por todos los receptores a medida que se van generando los resultados esperados.

Implementación en el proyecto: No, la responsabilidad radica en la clase reserva desde donde se asocian los datos de la clase usuario y se reciben los que crea y envía la factoría con respecto a la zona común solicitada.

PATRON COSTRUCTOR

Tipo: Patrón de creacional

Problema: Cómo separar los objetos complejos

Solución: Una clase que construye los objetos complejos. Se sugiere crear otra clase que controle el proceso de creación, además se sugiere que el constructor implemente los métodos de construcción definidos en una interface.

Implementación en el proyecto: No, el proceso de creación el patrón factoría nos soluciona de manera adecuada el requerimiento planteado.

PATRON PROTOTIPO

Tipo: Patrón de creacional

Problema: Crear a partir de un modelo.

Solución: Los nuevos objetos que se crearán de los prototipos, son clonados. Vale decir, tiene como finalidad crear nuevos objetos duplicándolos, clonando una instancia creada previamente.

Implementación en el proyecto: No, a pesar de que nos ahorrar tiempo en la creación de objetos con la clonación nos inclinamos por una solución creacional que nos brinda la factoría.

PATRON SINGLETON

Tipo: Patrón de creacional

Problema: Múltiples clientes que referencian objetos de la misma clase, los cuales tienen diferentes valores en sus atributos, causando errores en la ejecución porque todas las instancias deben tener el mismo valor en sus atributos.

Solución: Crear una sola instancia haciendo el método constructor privado y creando un método que retorna la instancia creada/existente de la clase.

Implementación en el proyecto: Si, nos asegura que la clase BD solo tiene una instancia y nos provee un punto de acceso global a esa instancia.

PATRON INMUTABLE

Tipo: Patrón de creacional

Problema: La no modificación del contenido de un objeto

Solución: Indicar/asegurar que el objeto solo puede tomar valores una vez, normalmente durante su creación.

Implementación en el proyecto: No, No existe un acceso simultáneo a los objetos que permita el cambio constante de estos y así afecte de manera directa los objetos.

PATRON FACTORIA

Tipo: Patrón de creacional

Problema: Delegar la responsabilidad de crear objetos; la clase no debe tener la responsabilidad de conocer e instanciar los objetos de otras clases permitiendo aumentando la flexibilidad a desacoplar las relaciones a otras clases.

Solución: Delegar a creación de los objetos a otra clase que ofrece un método que devuelve la instancia requerida.

Implementación en el proyecto: Si, se implementa en la creación de los tipos de zonas que dispone de la universidad para poder realizar la reserva.

PATRON FACTORIA ABSTRACTA

Tipo: Patrón de creacional

Problema: Un marco para estandarizar el modelo arquitectónico para una gama de aplicaciones, permitir aplicaciones individuales para definir sus propios objetos de dominio y prevén su instanciación.

Solución: Nos permite crear, mediante una interfaz, conjuntos o familias de objetos (denominados productos) que dependen mutuamente y todo esto sin especificar cuál es el objeto concreto

Implementación en el proyecto: No, el patrón factoría nos soluciona de manera efectiva el requerimiento de los tipos de zona solicitados

PATRON FLYWEIGHT

Tipo: Patrón estructural

Problema: Cómo hacer para eliminar a redundancia

Solución: Separar las clases diferentes, la información intrínseca y la información extrínseca. La implementación intrínseca la agrupó en un objeto de peso ligero, sea un singleton o un objeto inmutable. La información extrínseca se define en otra clase que además tiene una referencia al objeto de peso ligero.

Implementación en el proyecto: No, no es necesario crear objetos a tan bajo nivel ya que la flexibilidad del sistema se da en que la factoría genera únicamente las zonas requeridas por la reserva, ni más ni menos.

PATRON ADAPTER

Tipo: Patrón estructural

Problema: Cómo integrar una clase que soluciona el problema a un cliente, el cual además a provisto la interface que quiere que la clase cumpla.

Solución: Crear un adaptador que implemente la interface del cliente y solucione el problema de este haciendo uso de la clase que sabe cómo resolver el problema.

Implementación en el proyecto: No, la aplicación a desarrollar no contiene clases incompatibles entre ellas que nos lleven a utilizar este patrón.

PATRON FACADE

Tipo: Patrón estructural

Problema: Cómo uso los servicios de un sistema

Solución: Crear una interfaz que indique cómo se hace uso de los subsistemas.

Implementación en el proyecto: Si, es la puerta de entrada a todo nuestro sistema enfocando en un solo punto el acceso a este, se reduce la complejidad y minimizamos las dependencias.

PATRON DECORATOR

Tipo: Patrón estructural

Problema: Cómo evitar la utilización de las subclases, generacionales o extensiones.

Solución: Crear una nueva clase que tenga la nueva funcionalidad y tenga como atributo a la clase de la cual desea extender o agregar las nuevas funcionalidades.

Implementación en el proyecto: No, las funcionalidades planteadas inicialmente en el aplicativo son gestionadas por la clase reserva no se manejan herencias ante lo cual las responsabilidades son de cada clase.

En versiones posteriores al aplicativo se podría implementar a fin de integrar nuevas funcionalidades, ya que es compatible con el diseño elegido hasta el momento.

PATRON COMPOSITE:

Tipo: Patrón estructural

Problema: Cómo agrupar los objetos individuales a partir de objetos más sencillos para crear objetos complejos

Solución: Agrupar en componentes funcionales.

Implementación en el proyecto: No, no es necesaria la representación de objetos compuestos a través de jerarquías como parte de un todo.

CONCLUSIONES

Existen muchos patrones de diseño que solucionan diferentes problemas planteados por ello, para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.