Instituto Politécnico Nacional

Escuela Superior de Cómputo

PATRONES DE DISEÑO

Investigación

Análisis y Diseño Orientado a Objetos

Profesor: Chadwick Carreto Arellano

Grupo: 2CM7

Fecha: 27 / Mayo /2018

Alumno:

Calva Hernández José Manuel 2017630201

Contenido

Desarrollo Histórico

El término patrón se utiliza inicialmente en el campo de la arquitectura, por Christopher Alexander, a finales de los 70's. Este conocimiento es transportado al ámbito del desarrollo de software orientado a objetos y se aplica al diseño. De allí es extrapolado al desarrollo en general y a las demás etapas.

En 1987, Ward Cunningham y Kent Beck trabajaron con Smaltalk y diseñaron interfaces de usuario. Decidieron, para ello, utilizar alguna de las ideas de Alexander para desarrollar un lenguaje pequeño de patrones para servir de guía a los programadores de Smaltalk. Así dieron lugar al libro "Using Pattern Languajes for Object-Oriented Programs"

Mark Grand publica en 1998 "Patterns in Java (volume 1)" que es un catálogo de patrones de diseño ilustrados con UML. En 1999 el mismo autor publica "Patterns in Java (volume 2)" en donde se añaden algunos patrones de diseño más y otros tipos de patrones tales como patrones de organización de código, patrones de optimización de código, etc.

¿Qué es un patrón de diseño?

En la ingeniería de software, un patrón de diseño es una solución reutilizable general a un problema que ocurre comúnmente en el diseño de software. Un patrón de diseño no es un diseño acabado que se puede implementar directamente en código, sino que hay que aplicarlo a un problema real y adaptarlo en la medida de lo posible.

Los patrones de diseño son arquitecturas probadas para construir software orientado a objetos flexible y que pueda mantenerse. El campo de los patrones de diseño trata de enumerar a los patrones recurrentes, y de alentar a los diseñadores de software para que los reutilicen y puedan desarrollar un software de mejor calidad con menos tiempo, dinero y esfuerzo.

En general, un patrón tiene cuatro elementos esenciales:

1. El nombre del patrón se utiliza para describir un problema de diseño, su solución, y consecuencias en una o dos palabras. Nombrar un patrón incrementa inmediatamente nuestro vocabulario de diseño. Esto nos permite diseños a un alto nivel de abstracción. Tener un vocabulario de patrones nos permite hablar sobre ellos con nuestros amigos, en nuestra documentación, e incluso a nosotros mismos.

2. El problema describe cuando aplicar el patrón. Se explica el problema y su contexto. Esto podría describir problemas de diseño específicos tales como algoritmos como objetos. Podría describir estructuras de clases u objetos que son sintomáticas de un diseño inflexible. Algunas veces el problema incluirá una lista de condiciones que deben cumplirse para poder aplicar el patrón.

3. La solución describe los elementos que forma el diseño, sus relaciones, responsabilidades y colaboraciones. La solución no describe un diseño particular o implementación, porque un patrón es como una plantilla que puede ser aplicada en diferentes situaciones. En cambio, los patrones proveen una

descripción abstracta de un problema de diseño y como una disposición general de los elementos (clases y objetos en nuestro caso) lo soluciona.

4. Las consecuencias son los resultados de aplicar el patrón. Estas son muy importantes para la evaluación de diseños alternativos y para comprender los costes y beneficios de la aplicación del patrón.

Los patrones de diseño tienen un cierto nivel de abstracción. Los patrones de diseño no son diseños tales como la realización de listas y tablas hash que pueden ser codificadas en clases y reutilizadas. Un algoritmo puede ser un ejemplo de implementación de un patrón, pero es demasiado incompleto, específico y rígido para ser un patrón. Una regla o heurística puede participar en los efectos de un patrón, pero un patrón es mucho más. Los patrones de diseño son descripciones de las comunicaciones de objetos y clases que son personalizadas para resolver un problema general de diseño en un contexto particular.

Un patrón de diseño nombra, abstrae e identifica los aspectos clave de un diseño estructurado, común, que lo hace útil para la creación de diseños orientados a objetos reutilizables. Los patrones de diseño identifican las clases participantes y las instancias, sus papeles y colaboraciones, y la distribución de responsabilidades. Cada patrón de diseño se enfoca sobre un particular diseño orientado a objetos. Se describe cuando se aplica, las características de otros diseños y las consecuencias y ventajas de su uso.

Los patrones de diseño se pueden utilizar en cualquier lenguaje de programación orientado a objetos, adaptando los diseños generales a las características de la implementación particular.

Descripción de patrones de diseño

¿Cómo describimos los patrones de diseño? Las notaciones gráficas, aunque importantes y útiles, no son suficientes. Estás solo recogen el producto final del proceso de diseño como las relaciones entre clases y objetos. Para reutilizar el diseño, nosotros debemos también guardar las decisiones, alternativas y ventajas que nos llevaron a ese diseño. Los ejemplos concretos son también muy importantes, porque ellos nos ayudan a ver el funcionamiento del diseño.

Para describir los patrones de diseño se utiliza un formato consistente. Cada patrón es dividido en secciones de acuerdo con la siguiente plantilla. La plantilla nos muestra una estructura uniforme para la información, de tal forma que los patrones de diseño sean fáciles de aprender, comparar y utilizar.

a) Nombre del patrón: Esta sección consiste en un nombre del patrón y una referencia bibliografía que indica de donde procede el patrón. El nombre es significativo y corto, fácil de recordar y asociar a la información que sigue.

b) Objetivo: Esta sección contiene unas pocas frases describiendo el patrón. El objetivo aporta la esencia de la solución que es proporcionada por el patrón. El objetivo está dirigido a programadores con experiencia que pueden reconocer el patrón como uno que ellos ya conocen, pero para el cual ellos no le han dado

un nombre. Después de reconocer el patrón por su nombre y objetivo, esto podría ser suficiente para comprender el resto de la descripción del patrón.

c) Contexto: La sección de Contexto describe el problema que el patrón soluciona. Este problema suele ser introducido en términos de un ejemplo concreto. Después de presentar el problema en el ejemplo, la sección de Contexto sugiere una solución de diseño a ese problema.

d) Aplicabilidad: La sección Aplicabilidad resume las consideraciones que guían a la solución general presentada en la sección Solución. En que situaciones es aplicable el patrón.

e) Solución: La sección Solución es el núcleo del patrón. Se describe una solución general al problema que el patrón soluciona. Esta descripción puede incluir, diagramas y texto que identifique la estructura del patrón, sus participantes y sus colaboraciones para mostrar cómo se soluciona el problema. Debe describir tanto la estructura dinámica como el comportamiento estático.

f) Consecuencias: La sección Consecuencias explica las implicaciones, buenas y malas, del uso de la solución.

g) Implementación: La sección de Implementación describe las consideraciones importantes que se han de tener en cuenta cuando se codifica la solución. También puede contener algunas variaciones o simplificaciones de la solución

h) Usos en el API de Java: Cuando hay un ejemplo apropiado del patrón en el núcleo del API de Java es comentado en esta sección. Los patrones que no se utilizan en el núcleo del API de Java no contienen esta sección.

i) Código del ejemplo: Esta sección contiene el código del ejemplo que enseña una muestra de la implementación para un diseño que utiliza el patrón. En la mayoría de estos casos, este será el diseño descrito en la sección de Contexto.

j) Patrones relacionados: Esta sección contiene una lista de los patrones que están relacionados con el patrón que se describe.

Clasificación de los patrones de diseño

Dado que hay muchos patrones de diseño necesitamos un modo de organizarlos. En esta sección clasificamos los patrones de diseño de tal forma que podamos referirnos a familias de patrones relacionados. La clasificación nos ayuda a saber lo que hace un patrón. Según el libro “Patterns in Java (Volume 1)” existen seis categorías:

Patrones de diseño fundamentales

Los patrones de esta categoría son los más fundamentales e importantes patrones de diseño conocidos. Estos patrones son utilizados extensivamente en otros patrones de diseño.

Patrones de creación

Los patrones de creación muestran la guía de cómo crear objetos cuando sus creaciones requieren tomar decisiones. Estas decisiones normalmente serán resueltas dinámicamente decidiendo que clases instanciar o sobre que objetos un objeto delegará responsabilidades. La valía de los patrones de creación nos dice como estructurar y encapsular estas decisiones.

A menudo hay varios patrones de creación que puedes aplicar en una situación. Algunas veces se pueden combinar múltiples patrones ventajosamente. En otros casos se debe elegir entre los patrones que compiten. Por estas razones es importante conocer los seis patrones descritos en esta categoría.

Patrones de partición

En la etapa de análisis, tu problema para identificar los actores, casos de uso, requerimientos y las relaciones que constituyen el problema. Los patrones de esta categoría proveen la guía sobre como dividir actores complejos y casos de uso en múltiples clases.

Patrones estructurales

Los patrones de esta categoría describen las formas comunes en que diferentes tipos de objetos pueden ser organizados para trabajar unos con otros.

Patrones de comportamiento

Los patrones de este tipo son utilizados para organizar, manejar y combinar comportamientos.

Patrones de concurrencia

Los patrones de esta categoría permiten coordinar las operaciones concurrentes. Estos patrones se dirigen principalmente a dos tipos diferentes de problemas:

1. Recursos compartidos: Cuando las operaciones concurrentes acceden a los mismos datos o otros tipos de recursos compartidos, podría darse la posibilidad de que las operaciones interfirieran unas con otras si ellas acceden a los recursos al mismo tiempo. Para garantizar que cada operación se ejecuta correctamente, la operación debe ser protegida para acceder a los recursos compartidos en solitario. Sin embargo, si las operaciones están completamente protegidas, entonces podrían bloquearse y no ser capaces de finalizar su ejecución.
   El bloqueo es una situación en la cual una operación espera por otra para realizar algo antes de que esta proceda. Porque cada operación está esperando por la otra para hacer algo, entonces ambas esperan para siempre y nunca hacen nada.

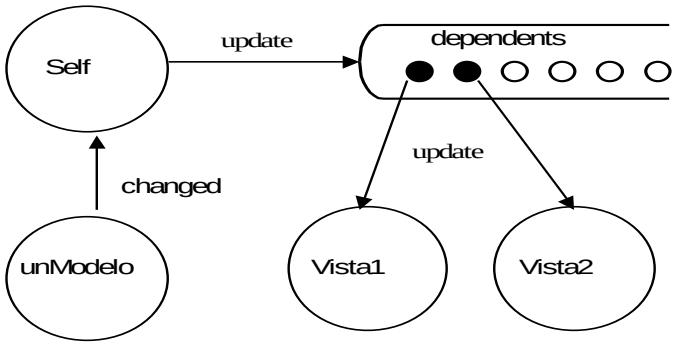
2. Secuencia de operaciones: Si las operaciones son protegidas para acceder a un recurso compartido una cada vez, entonces podría ser necesario garantizar que ellas acceden a los recursos compartidos en un orden particular. Por ejemplo, un objeto nunca será borrado de una estructura de datos antes de que esté sea añadido a la estructura de datos.

Ejemplos de patrones de diseño

Patrón de diseño OBSERVER

MVC separa vistas y modelos estableciendo un protocolo de suscripción /notificación entre ellos. Una vista debe asegurarse de que su apariencia refleja el estado del modelo. Siempre que los datos del modelo cambian, el modelo notifica a las vistas que dependen de este. En respuesta, cada vista tiene una oportunidad para actualizarse ella misma.

He aquí como trabaja: la clase Objeto, en la raíz de la jerarquía de clases, mantiene una colección de dependientes, permitiendo a cualquier objeto registrarse el mismo como dependiente de otro objeto. Objeto también implementa un método changed, así cualquier objeto puede enviar el mensaje changed así mismo. Lo que el método changed hace es enviar un mensaje update a cada uno de los dependientes del objeto. Así cuando un objeto envía un mensaje changed al mismo, sus dependientes reciben un mensaje update: automáticamente. Los dependientes deban implementar el método update correspondiente, en el cual ellos mismos se redespliegan o toman otras acciones de actualización.



El mecanismo de dependencia

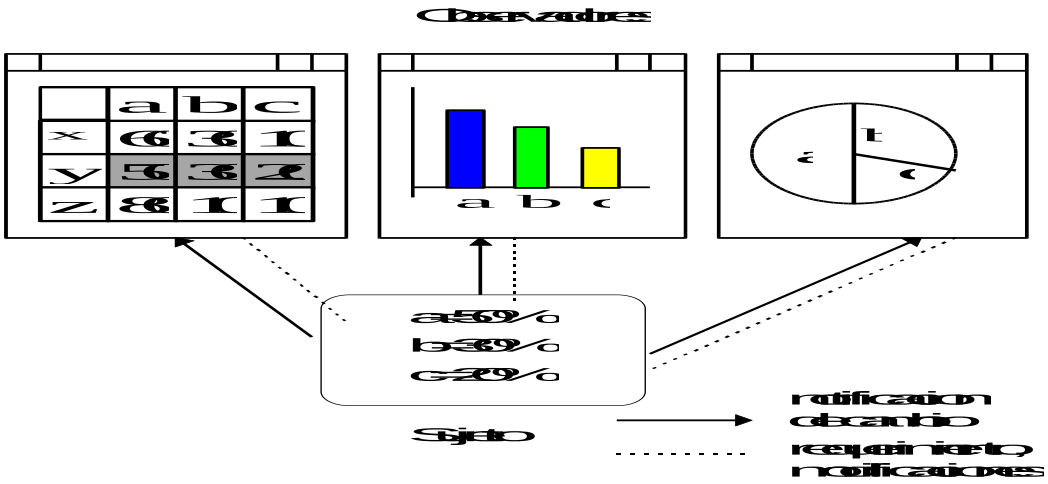


Figura 3.1

La aproximación anterior permite asignar múltiples vistas a un modelo para proveer diferentes presentaciones. También se pueden crear nuevas vistas para un modelo sin tener que reescribirlo.

El modelo se comunica con sus vistas cuando sus valores cambian y las vistas se comunican con el modelo para acceder a aquellos valores.

El patrón de diseño observador describe el diseño más general donde: se separan objetos para que los cambios que afectan a uno puedan afectar cualquier número de otros sin requerir que el objeto cambiado conozca detalles de los otros. Los objetos principales en este patrón son sujeto y observador, ver figura 3.2. Un sujeto puede tener cualquier número de observadores dependientes, ver figuras 3.1 y 3.2.

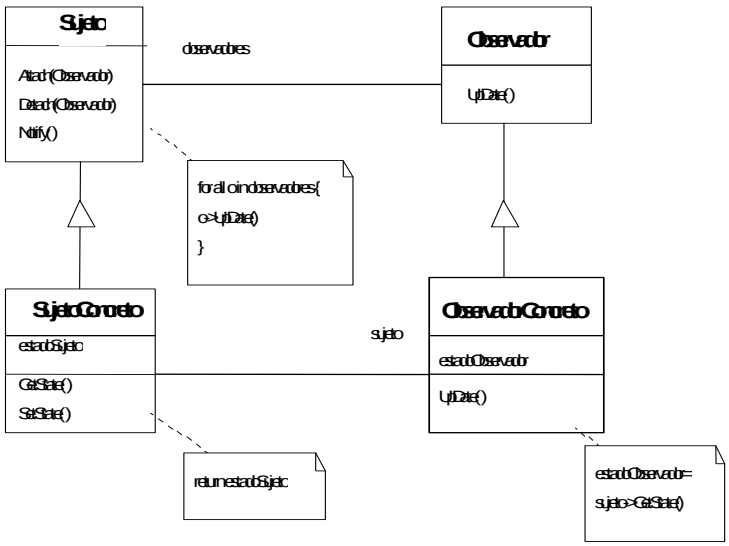


Figura 3.2

Patrón de diseño COMPOUND

Otra característica de MVC es que las vistas se pueden anidar. MVC soporta vistas anidadas con la clase CompositeView, una subclase de DependentComposite. Este es el equivalente compuesto de una vista. Los objetos CompositeView actúan como los objetos View; una vista compuesta puede ser usada siempre que una vista pueda ser usada, pero esta también contiene y maneja vistas anidadas.

El patrón de diseño Compuesto describe el diseño más general donde: se agrupan objetos y se trata el grupo como un objeto individual. Este permite crear una jerarquía de clases en la cual algunas subclases definen objetos compuestos que ensamblan los primitivos en objetos más complejos. L a clave del patrón de diseño Compuesto es una clase abstracta que representa tanto a primitivos como a sus contenedores, ver figura 3.3.

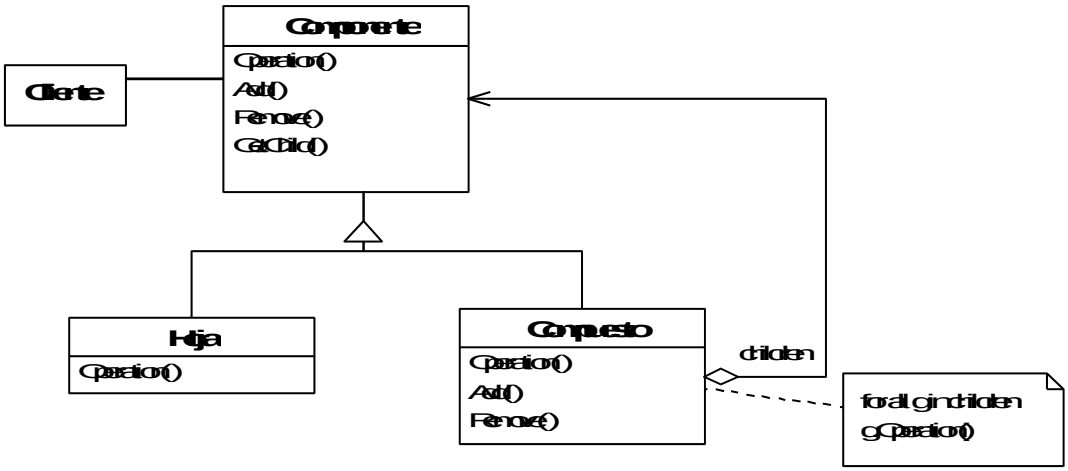


Figura 3.3

Patrón de diseño STRATEGY

MVC permite cambiar la forma como una vista responde a la entrada del usuario sin cambiar su presentación visual. MVC encapsula el mecanismo de respuesta en un objeto controlador. Hay una jerarquía de controladores, que hacen fácil crear un controlador nuevo como una variación de uno existente.

Una vista usa una instancia de una subclase controlador para implementar una estrategia de respuesta en particular; para implementar una estrategia diferente, simplemente se reemplaza la instancia con otra instancia de una subclase de controlador diferente. Es aun posible cambiar el controlador en tiempo de ejecución para permitir que la vista cambie la forma en que esta responde a la entrada de usuario. Por ejemplo, una vista puede ser deshabilitada para que esta no acepte entrada simplemente dándole un controlador que ignore los eventos de entrada.

La relación Vista-Controlador es un ejemplo del patrón de diseño estrategia. Una estrategia es un objeto que representa un algoritmo. Esto es útil cuando se quiere reemplazar el algoritmo ya sea estáticamente o dinámicamente, cuando se tienen muchas variantes del algoritmo, o cuando el algoritmo tiene estructuras complejas que se quieren encapsular, ver figura 3.4.

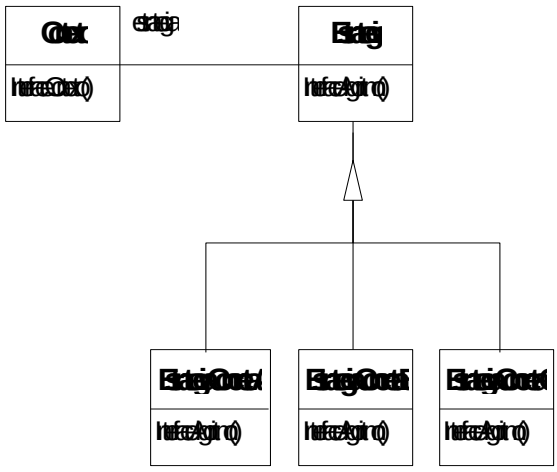


Figura 3.4

Patrón de diseño ADAPTER

Una clase es más reutilizable cuando se minimizan las suposiciones que otras clases deben hacer para usarla. Construyendo la adaptación a la interfaz de una clase, se elimina la suposición de que otras clases vean la misma interfaz. Por ejemplo, cuando se espera que una vista sea usada con diferentes modelos, o todos los modelos deben hablar el lenguaje de la vista o la vista debe ser capaz de adaptarse ella misma a diferentes mensajes. Hacer que los modelos se adapten viola el principio de que ningún modelo debe ser dependiente de sus vistas.

Un adaptador provee un conjunto standard de mensajes de acceso para vistas y controladores (value y value:) mientras permanece muy flexible en cuanto a comunicación con sus modelos. Un adaptador es como un traductor universal por que puede ser entrenado para hablar el lenguaje de cualquier modelo.

La clase abstracta ValueModel, definida para vistas que muestran un solo valor, provee el mecanismo de obtención de valor (value) y almacenamiento (value:). Esta tiene dos subclases, ValueHolder y PluggableAdaptor

Un ValueHolder traduce (para un objeto simple, guardado en su variable de instancia value, que no se comporta normalmente como modelo) los mensajes estándar value y value: que recibe en la devolución del objeto que guarda y su reemplazo (con notificación a sus dependientes del cambio).

Los adaptadores que se enchufan (pluggable adapters) son comunes en ObjectWorks /SmallTalk.

Un PluggableAdaptor traduce (para un objeto más complicado que tiene un vocabulario diferente que la vista, y posiblemente una semántica diferente) los mensajes standard value y value: que recibe (de las vistas y controladores) en acciones arbitrarias (definidas por bloques) que sirven para obtener y almacenar el valor deseado.

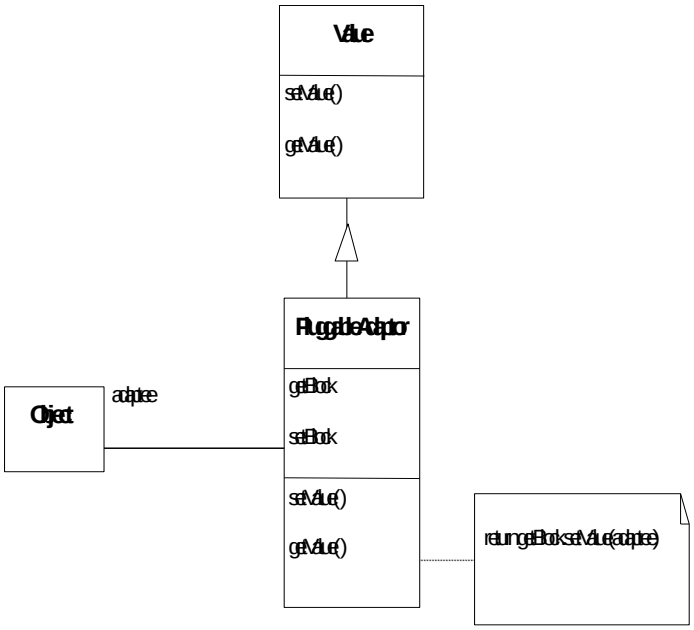


Figura 3.5

Los escritores de aplicaciones accesan al valor con nombres de dominio específicos al dominio como width y width:, pero ellos no deberían tener que subclasar ValueModel para traducir dichos nombres específicos a la aplicación a la interfaz de ValueModel.

En su lugar PluggableAdaptor permite pasar nombres de selector (como width y width:) directamente para el caso en que el modelo solo usa palabras diferentes para significar la misma cosa que value y value:. Este convierte aquellos selectores en bloques correspondientes automáticamente, ver figura 3.5.

El patrón de diseño Adaptor describe el diseño más general donde: se convierte la interfaz de una clase en otra interfaz que los clientes esperan. Adaptor permite que trabajen juntas clases que de otra manera no podrían hacerlo a causa de sus interfaces incompatibles. Dicho de otra forma, la adaptación a una interfaz nos permite incorporar nuestra clase en sistemas existentes que pueden esperar interfaces diferentes a la interfaz de la clase, ver figura 3.6.

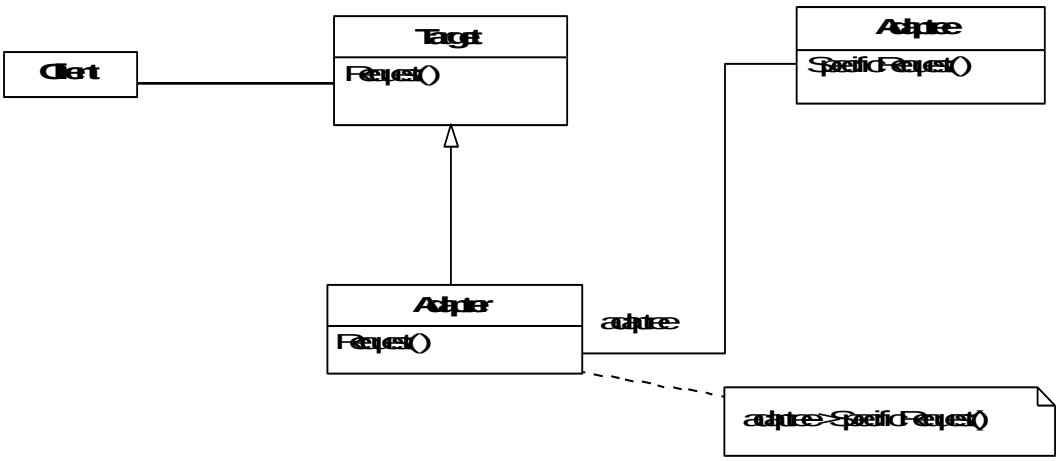


Figura 3.6

Patrón de diseño VALUE OBJECT (VO)

Consiste básicamente en la agrupación de datos dentro de un objeto, estos datos representan los campos de una tabla o entidad de la BD y facilitan su mantenimiento y transporte dentro del sistema.

Problema: Al momento de pasar argumentos de un objeto a un método puede ocasionar que el método reciba gran cantidad de datos pasados como parámetros, si posteriormente se requiere la modificación de uno de esos argumentos se obliga a que todos los métodos que reciban estos argumentos sean cambiados, presentándose problemas de acoplamiento y mantenibilidad.

Solución: Se crean clases que representan las tablas de la BD que utiliza el sistema, de esta manera las propiedades o atributos de la clase serán los campos de la entidad, permitiendo encapsular la información y facilitando la manera en que estos son transportados, así al momento de enviar los parámetros a un método, se envía un solo objeto que los contiene.

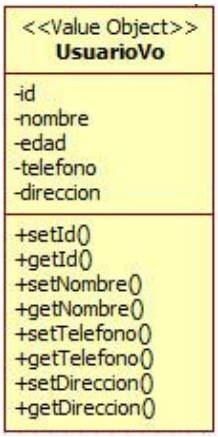


Figura 3.7