Laboratorio de Ingeniería de Software II

Patrones de Comportamiento

Objetivo

Entender el patrón comando y complementar con un conjunto pequeño de patrones de comportamiento: MVC (Micro-arquitectura) y Observador

Patrón Comando

Utilizando el patrón command, un objeto **invocador,** emite una solicitud en representación de un cliente de tal forma que el conjunto de servicios de los **objetos receptores**, pueden ser **desacoplados**. El patrón sugiere crear una abstracción para el procesamiento o acción a ejecutar, en respuesta a la solicitud del cliente.

Problema

En general una aplicación orientada a objetos consiste en la interacción de objetos que ofrecen una funcionalidad. En respuesta a la interacción del usuario, la aplicación ejecuta algún tipo de procesamiento. Para este propósito la aplicación utiliza servicios de diferentes objetos para los requisitos de procesamiento. En términos de implementación, la aplicación puede depender de un objeto designado que *invoca* los métodos de esos objetos mediante el paso de argumentos.

A este objeto se lo denomina *invoker*, y se encarga de invocar las operaciones de los diferentes objetos. El invocador puede ser tratado como parte de la aplicación cliente. Los objetos que ofrecen los servicios para procesar solicitudes se denominan objetos *receiver* (receptores).

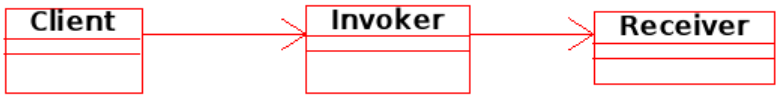

Figura 1. Interacción de objetos: antes de aplicar el patrón Command

En este diseño el objeto *invocador* podría tener muchas instrucciones if:

```
if (RequestType=TypeA){
  //do something
}
if (RequestType=TypeB){
  //do something
}
…
```

Cuando un nuevo tipo de característica se adicione a la aplicación, el código existente debe ser modificado y de esta forma se violaría el principio *open-closed*.

```
...
if (RequestType=NewType){
  //do something
}
…
```

Solución

Utilizando el patrón command, es el **invocador,** quien emite una solicitud en representación de un cliente, y el conjunto de servicios de los **objetos receptores**, pueden ser **desacoplados**. El patrón sugiere crear una abstracción para el procesamiento o acción a ejecutar, en respuesta a la solicitud del cliente.

Esta abstracción puede ser diseñada como una interfaz común a ser implementada por diferentes implementadores concretos, denominados como *Objetos de comando* (Commandobjets). Cada objeto *command* representa un tipo diferente de **solicitudcliente** y el correspondiente **procesamiento**. En la Figura 2, la interface *Command* representa la abstracción. Su método execute, es implementado por dos clases implementadoras (*ConcreteCommand_1* y *ConcrteCommand_2*).

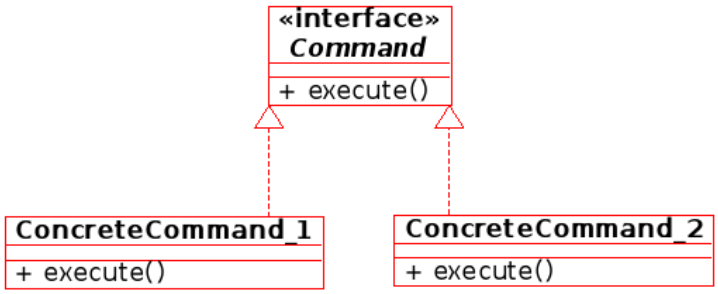

Figura 2. Jerarquía de objetos comando

Un objeto *Command* es responsable de ofrecer una funcionalidad requerida para procesar la solicitud que representa, pero no contiene la implementación de la funcionalidad. Los objetos *command* hacen uso de los objetos *Receiver* para ofrecer sus funcionalidades (Ver Figura 3).

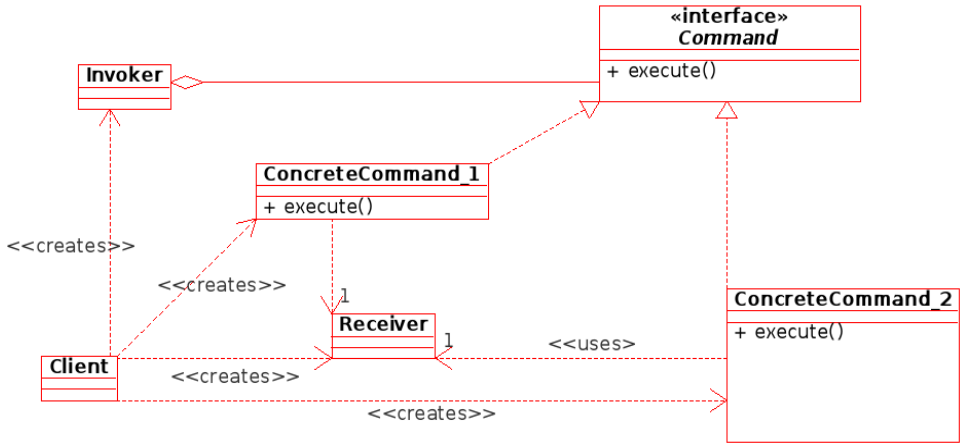

Figura 3. Diseño después de aplicar el patróncomado

A continuación, se muestra la interacción que ocurre cuando la aplicación cliente necesita un servicio:

1. Se crean los objetos *Receiver* necesarios.
2. Se crea el objeto *Command* apropiado y se configura con sus objetos *Receiver* del paso 1.
3. Se crea una instancia del *invocador* y se configura con su objeto *Command* del paso 2.
4. El *invocador* ejecuta el método *execute()* sobre el objeto *Command*.

Como parte de su implementación el método *execute*, un típico objeto *Command,* invoca los métodos necesarios sobre los objetos *Receiver*.

En este nuevo diseño se debe aclarar:

- El cliente/invoker no interactúa direcatamente con los objetos *Receiver*, logrando el desacople entre el uno y el otro.
- Cuando la aplicación necesita ofrecer una nueva característica, un nuevo objeto *Command* puede ser adicionado. Esto no requiere hacer cambios en el código del *invoker*. Por lo tanto, se cumple con el principio *open-closed*.
- Debido a que la solicitud es diseñada en forma de un objeto, se abre un conjunto de posibilidades como:
    - Almacenar un objeto *Command* como mecanismo de persistencia para ser ejecutado posteriormente, y además, aplicar procesamiento inverso y lograr la operación deshacer.
    - Agrupar varios objetos *Command* para ser ejecutados como una unidad (Transacción).

El patrón de micro-arquitectura MVC

La micro-arquitectura Model-View-Controller es un marco orientado a objetos y un patrón de diseño bien conocido para crear aplicaciones y componentes GUI reutilizables.MVC prescribe una forma de dividir una aplicación o componente en tres partes: el modelo, la vista y el controlador. La motivación

original para esta separación fue mapear los roles tradicionales de entrada, procesamiento y salida en el ámbito de la GUI:

Entrada --> Procesamiento --> Salida

Controlador --> Modelo --> Vista

La entrada del usuario, la función y el estado del sistema y la retroalimentación visual para el usuario están separados y manejados por el controlador, el modelo y la vista, respectivamente. La Figura 4 representa la tríada básica de MVC y las líneas de comunicación.

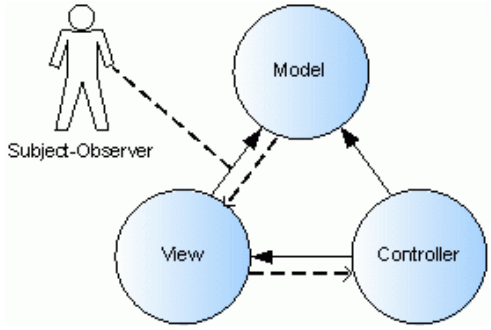

Figura 4. Elementos del Patrón Modelo Vista Controlador. Los círculos representan componentes, las líneas continuas invocaciones explícitas y las líneas discontinuas invocaciones implícitas.

El modelo es la piedra angular de la tríada MVC. Como su nombre lo indica, su trabajo es modelar sistema del mundo real emulando su estado y funcionalidad. Los modelos definen consultas para informar sobre el estado, comandos para modificar el estado y notificaciones para informar a los observadores (vistas, por ejemplo) que se ha producido un cambio en el estado. El controlador es responsable de definir el comportamiento de la tríada. Su trabajo es recibir la entrada del mouse (En java MouseListener, ActionListener) y del teclado (KeyListener) y mapear este estímulo del usuario en la respuesta de la aplicación, por ejemplo, ejecutando los cambios del modelo. La vista administra un área rectangular de la pantalla y es responsable de la presentación de datos. Y debido a su relación de observador con el modelo, se pueden definir y adjuntar nuevas vistas a un modelo mientras se mantiene constante la interfaz del modelo.

Relación Modelo-Vista-Controlador

La figura 4 muestra las relaciones entre modelo, vista y controlador en una tríada.Las líneas discontinuas representan una agregación débilmente tipada (en lo más abstracto de la jerarquía) y las líneas continuas representan una agregación fuertemente tipada (en lo más concreto de la jerarquía). El modelo mantiene una referencia a la vista, lo que le permite enviar notificaciones de

cambio de tipo débil a los controladores. Dado que es una relación débilmente tipada, el modelo hace referencia al controlador solo a través de una clase base que le permite enviar notificaciones a el controlador. Por el contrario, la Vista sabe exactamente qué tipo de modelo observa. Tiene una referencia fuertemente tipada al modelo específico que le permite llamar a cualquiera de las funciones del modelo. La ventana gráfica también tiene una relación débilmente tipada con el controlador. La ventana gráfica no está vinculada a un tipo específico de controlador, lo que significa que se pueden usar diferentes tipos de controladores con la misma vista. El controlador tiene referencias tanto para el modelo como para la vista y conoce el tipo de ambos. Debido a que el controlador define el comportamiento de la tríada, necesita conocer el tipo de modelo y de vista para traducir la entrada del usuario en la respuesta de la aplicación.

El Patrón Sujeto-Observador en MVC

La relación entre el modelo y la vista en realidad está definida por otro patrón de diseño. El patrón observador que define una dependencia de uno a muchos entre objetos, de modo que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente. En el caso de MVC, el modelo es Observado y las Vistas son observadores.

Lo que debo entregar (Ejercicios)

A partir del código brindado para los comandos de OpenMarket se solicita complementar el CRUD producto con la siguiente funcionalidad:

- Agregar las opciones de Hacer, deshacer y rehacer asociadas a las operaciones de adición, edición y borrado de un producto.
- Haciendo uso del MVC agregar una interface gráfica de usuario que permita al crear una categoría poder agregar productos a la misma de una manera más interactiva. Es decir, tener una lista de productos resultado de una consulta, seleccionar productos que se quieren categorizar, agregarlos a la categoría y visualizar inmediatamente los productos de esa categoría. Agregar un conjunto de productos a una categoría debe poderse deshacer.
- Condiciones de entrega:
    - Entrega en repositorio GitHub
    - Pruebas unitarias a las clases de servicio y al invocador de comandos.