POLITÉCNICA

"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL

MiW

Patrones de Diseño

10. *Command*

Luis Fernández Muñoz

https://www.linkedin.com/in/luisfernandezmunyoz

setillofm@gmail.com

INDICE

1. Problema

- *Una posible organización en un restaurante sería que cuando un camarero anota una comanda de un cliente, se dirija a la cocina, busque/encargue a un cocinero adecuado (p.e. sabe cocinar el plato, tiene tiempo, tiene fogones libres, …) y le consulta si está disponible por cada plato para servir al cliente. O sea, una caos donde todo camarero interactúa con todo cocinero potencialmente.*
- *Una alternativa es que los camareros dejen la comanda en un lugar acordado al que acuden los cocineros para escoger la siguiente según sus condiciones y, tras la preparación, dejan los platos para servir a los clientes. Con dos equipos totalmente independientes y mucho menos trabajo para el cocinero.*

- ***Encapsula un petición como un objeto, permitiendo de ese modo parametrizar clientes con diferentes peticiones, colas o solicitudes registradas, y apoyar las operaciones de deshacer.***

1. Problema

- Úsese cuando:
  - Parametrizar objetos con una acción a realizar. Los objetos Orden son un sustituto orientado a objetos para las funciones *callback*.
  - Especificar, poner en cola y ejecutar peticiones en diferentes instantes del tiempo. Un objeto Orden puede tener un tiempo de vida independiente de la petición original. Si se puede representar el receptor de una petición en una forma independiente del espacio de direcciones, entonces se puede transferir un objeto orden con la petición a un proceso diferente y llevar a cabo la petición allí.

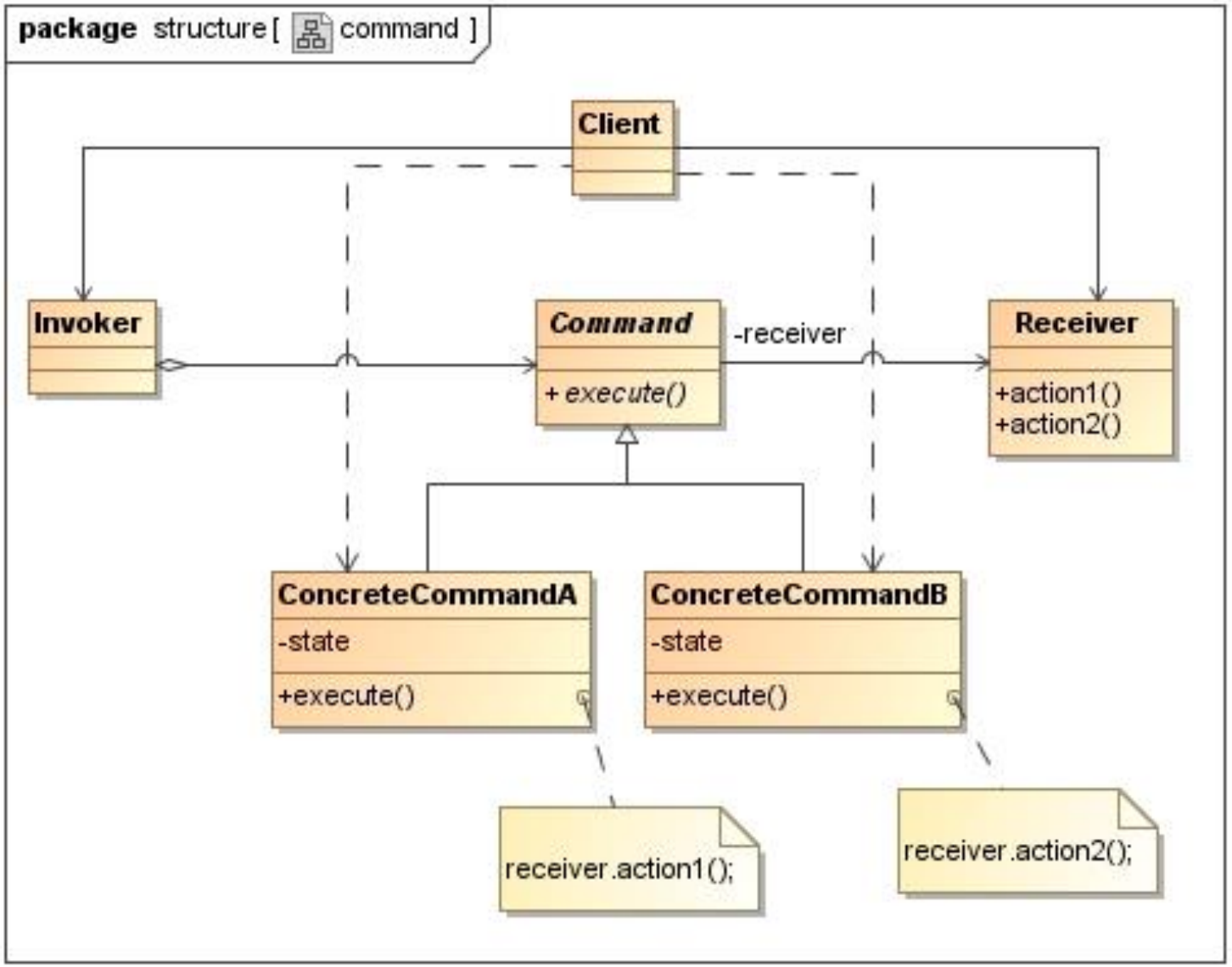
1. Problema

- Permitir deshacer. La operación Ejecutar de Orden puede guardar en la propia orden el estado que anule sus efectos. Debe añadirse a la interfaz Orden un operación Deshacer() que anule los efectos de una llamada anterior a Ejecutar(). Las órdenes ejecutadas se guardan en una lista que hace las veces de historial. Se pueden lograr niveles ilimitados de deshacer y repetir recorriendo dicha lista hacia atrás y hacia adelante llamando respectivamente a Deshacer() y Ejecutar()
- Permitir registrar los cambios de manera que se puedan volver a aplicar en caso de caída del sistema. Aumentando la interfaz de Orden con operaciones para cargar y guardas se puede mantener un registro persistente de los cambios. Recuperarse de una caída implica volver a cargar desde el disco la órdenes guardadas y volver a ejecutarlas con la operación Ejecutar()

1. Problema

- Estructurar un sistema alrededor de operaciones de alto nivel construidas sobre operaciones básicas. Dicha estructura es común en los sistemas de información que permiten transacciones.
  - Una transacción encapsula un conjunto de cambios sobre unos datos. El patrón *Command* ofrece un modo de modelar transacciones. Las órdenes tienen una interfaz común, permitiendo así invocar a todas la transacciones del mismo modo. El patrón también facilita extender el sistema con nuevas transacciones.

2. Solución


package structure [command]
Client
Invoker
Command
+ execute()
-receiver
Receiver
+action1()
+action2()
ConcreteCommandA
-state
+execute()
ConcreteCommandB
-state
+execute()
receiver.action1();
receiver.action2();

2. Solución

- Una orden puede tener un amplio conjunto de habilidades. Por un lado, simplemente define un enlace entre un receptor y las acciones que lleva a cabo la petición. Por el otro, lo implementa todo ella misma sin delegar para nada en el receptor. Este último extremo resulta útil cuando queremos definir órdenes que sean independientes de las clases existentes, cuando no existe ningún receptor adecuado o cuando una orden conoce implícitamente a su receptor. En algún punto entre estos dos extremos se encuentran las órdenes que tienen el conocimiento suficiente para encontrar dinámicamente sus receptores

2. Solución

- Las órdenes pueden permitir capacidades de deshacer y repetir si proveen un modo de revertir su ejecución. Una clase OrdenConcreta podría necesitar almacenar información de estado adicional para hacer esto. Este estado puede incluir:
  - El objeto Receptor, el cual es quien realmente realiza las operaciones en respuesta a la petición;
  - Los argumentos de la operación llevada a cabo por el receptor; y
  - Cualquier valor original del receptor que pueda cambiar como resultado de manejar la petición. El receptor debe proporcionar operaciones que permitan a la orden devolver el receptor a su estado anterior

2. Solución

- Para permitir un nivel de deshacer, una aplicación sólo necesita guardar la última orden que se ejecutó. Para múltiples niveles de deshacer/rehacer, la aplicación necesita guardar un historial de las órdenes que han sido ejecutadas, donde la máxima longitud de la lista determina el número de niveles de deshacer/rehacer. Recorrer la lista hacia atrás deshaciendo las órdenes cancela sus efectos; recorrerla hacia delante ejecutando las órdenes las rehace.
  - Una orden anulable puede que deba ser copiada antes de que se guarde en el historial. Eso es debido a que el objeto orden que llevó a cabo la petición original más tarde ejecutará otra peticiones. La copia es necesaria para distinguir entre diferentes invocaciones de la misma orden si su estado puede variar entre invocaciones sucesivas. En caso de que el estado de la orden no cambie tras su ejecución no es necesario realizar la copia, basta con guardar en el historial una referencia a la orden.

2. Solución

- La histéresis puede ser un problema a la hora de garantizar un mecanismo de deshacer/repetir fiable, que preserve la semántica. Los errores se pueden acumular a medida que se ejecutan y deshacen las órdenes repetidamente, de manera que es estado de una aplicación finalmente difiera de sus valores originales. Por tanto, puede ser necesario guardar más información con la orden para asegurar que los objetos son devueltos a su estado original.

3. Consecuencias

- Orden desacopla el objeto que invoca la operación de aquél que sabe cómo realizarla
- Las órdenes son objetos de primera clase. Pueden ser manipulados y extendidos como cualquier otro objeto
- Se pueden ensamblar órdenes en una orden compuesta.
- Es fácil añadir nuevas órdenes, ya que no hay que cambiar las clases existentes.

4. Relaciones

- ***Prototype*** para copiar los *Command* antes de ser guardados en el historial
- ***Memento*** para guardar el estado de las operaciones que pueden deshacerse
- ***Composite*** para implementar *Command* que sean macros de varios otros comandos.

5. Comparativa

- ***Command*** *vs* ***Memento***.
  - *Command* representa un elemento como una petición. *Memento* representa el estado interno de un objeto en un momento concreto.
  - En ambos casos, el elemento puede tener una representación interna compleja, pero los clientes nunca llegan a percibirla.
  - El polimorfismo es importante en *Command* , ya que ejecutar el *Command* es una operación polimórfica. Por el contrario, la interfaz del *Memento* es tan limitada que éste sólo puede pasarse como un valor. Por tanto, es probable que no presente ninguna operación polimórfica a sus clientes.