2

(VIII)

Patrón Command

(Patrones de diseño)

Diseño del Software

Grado en Ingeniería Informática del Software

Curso 2022-2023

Escuela de Ingeniería Informática
Universidad de Oviedo

César Acebal

Command (Orden)

- **Patrón de comportamiento (ámbito de objetos)**
- **Propósito:**

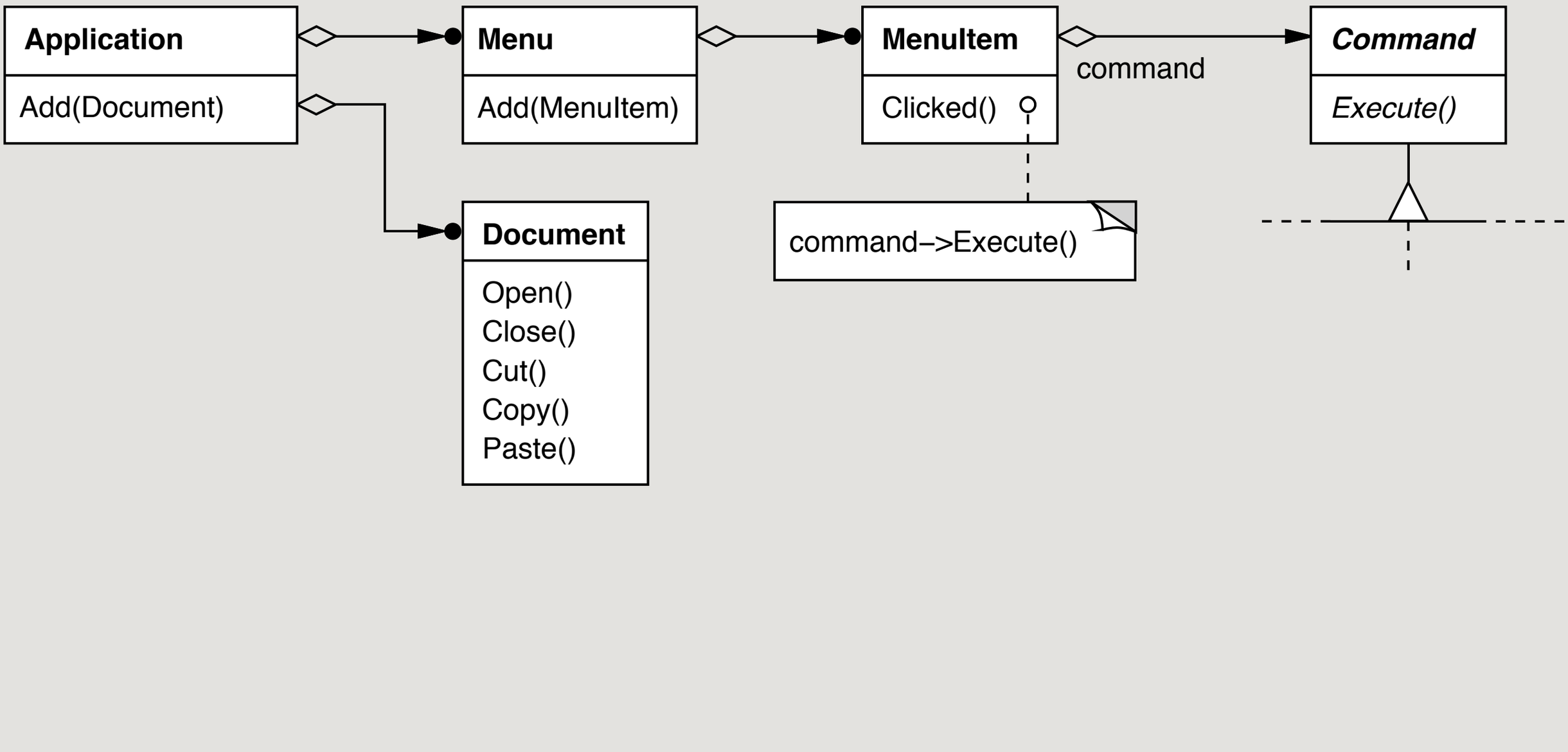
> *Encapsula una petición dentro de un objeto, permitiendo parametrizar a los clientes con distintas peticiones, encolarlas, guardarlas en un registro de sucesos o implementar un mecanismo de deshacer/repetir.*

- **También conocido como:**
  - Action, Transaction

Motivación

- **Una biblioteca de clases para interfaces de usuario tendrá objetos como botones y elementos de menú responsables de realizar alguna operación en respuesta a una entrada del usuario**
- **La biblioteca no puede implementar dichas operaciones directamente en el botón o el menú**
  - Sólo las aplicaciones que usan la biblioteca saben qué hay que hacer y a qué operaciones de otros objetos hay que llamar

Motivación


Application
Add(Document)
Menu
Add(MenuItem)
MenuItem
Clicked()
command
Command
Execute()
Document
Open()
Close()
Cut()
Copy()
Paste()
command->Execute()

Motivación

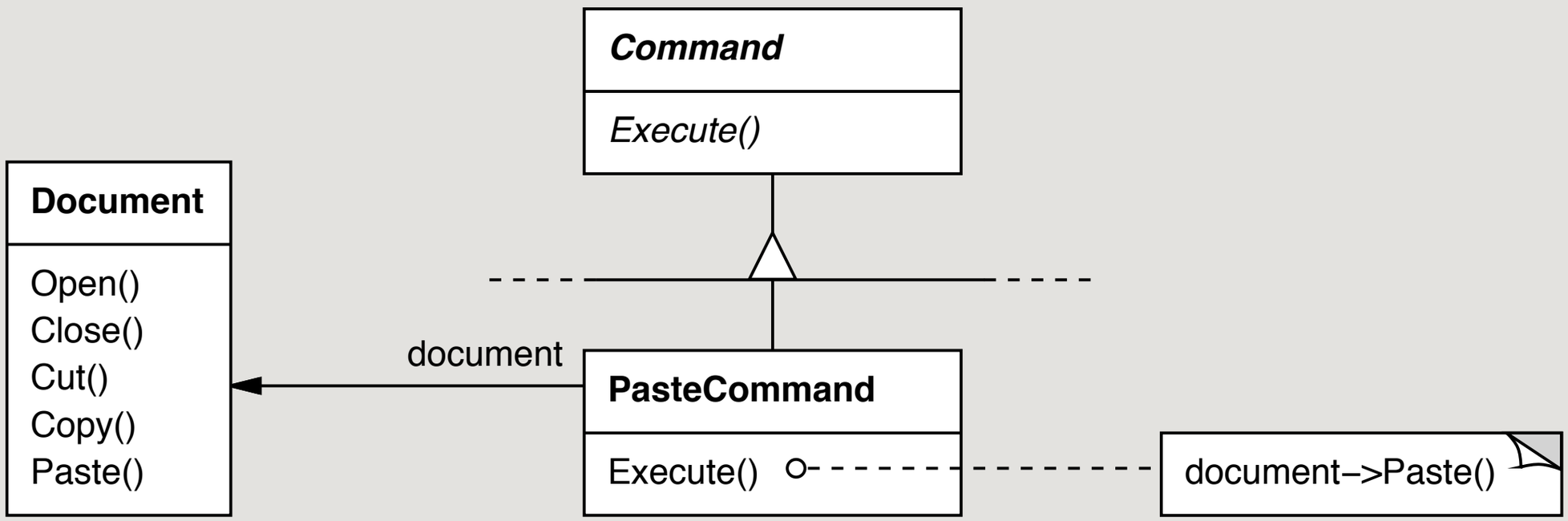
- La clave de este patrón es una interfaz `Command` que define una operación `execute`
- Son las subclases concretas quienes implementan la operación y especifican el receptor de la orden

Motivación

- Podemos configurar cada elemento del menú, `MenuItem`, con un objeto `Command`
- Los elementos del menú no saben qué objeto concreto están usando (simplemente llaman a su método `execute`)

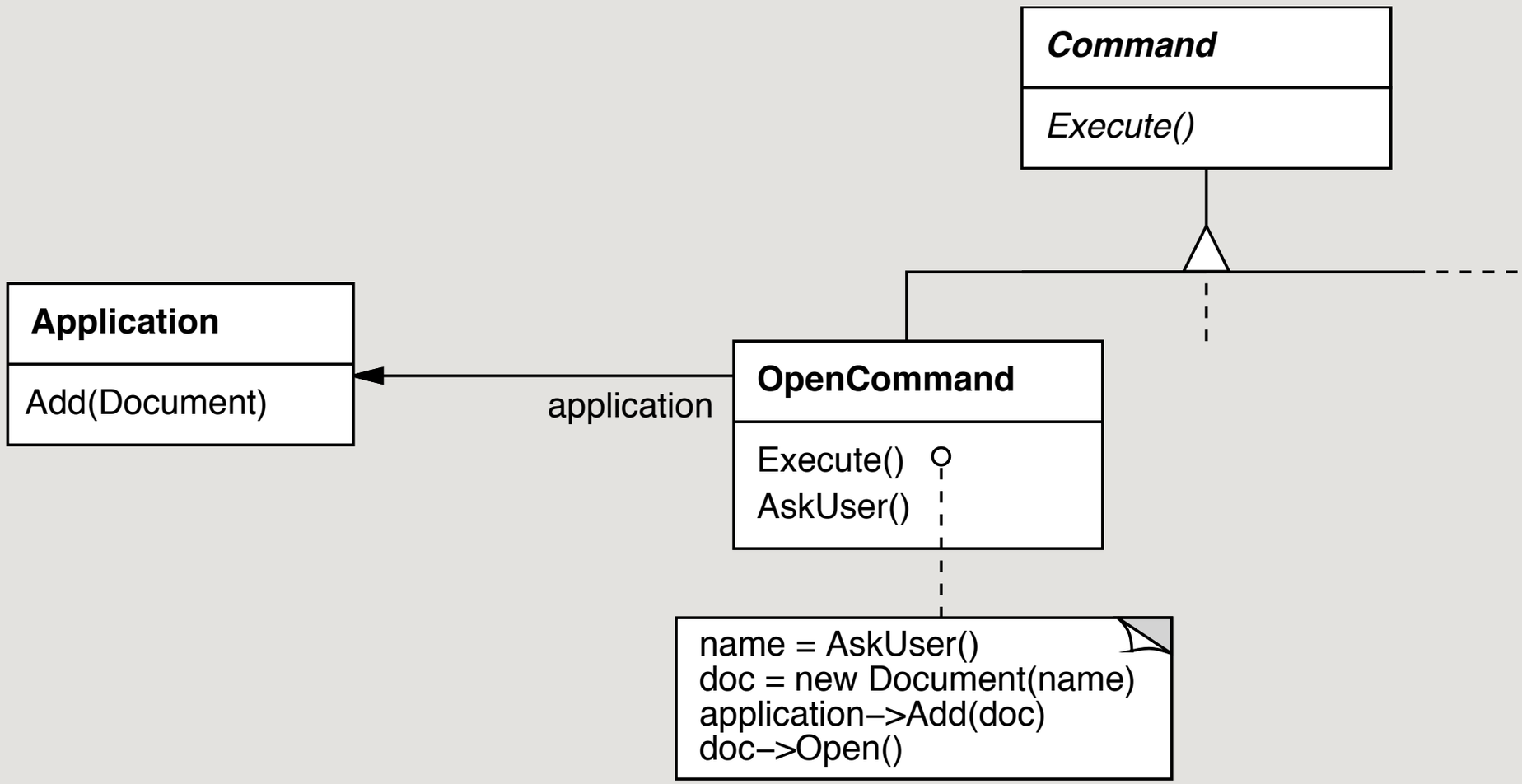
Motivación

Por ejemplo, «pegar»



Motivación

0 «abrir documento»



Motivación

- **En resumen, lo que permite el patrón *Command* es desacoplar al objeto que invoca a la operación de aquél que tiene el conocimiento necesario para realizarla**
- **Esto nos otorga muchísima flexibilidad**
  - Podemos hacer, por ejemplo, que una aplicación proporcione tanto un elemento de menú como un botón para hacer una determinada acción
  - Podemos cambiar dinámicamente los objetos `Command`

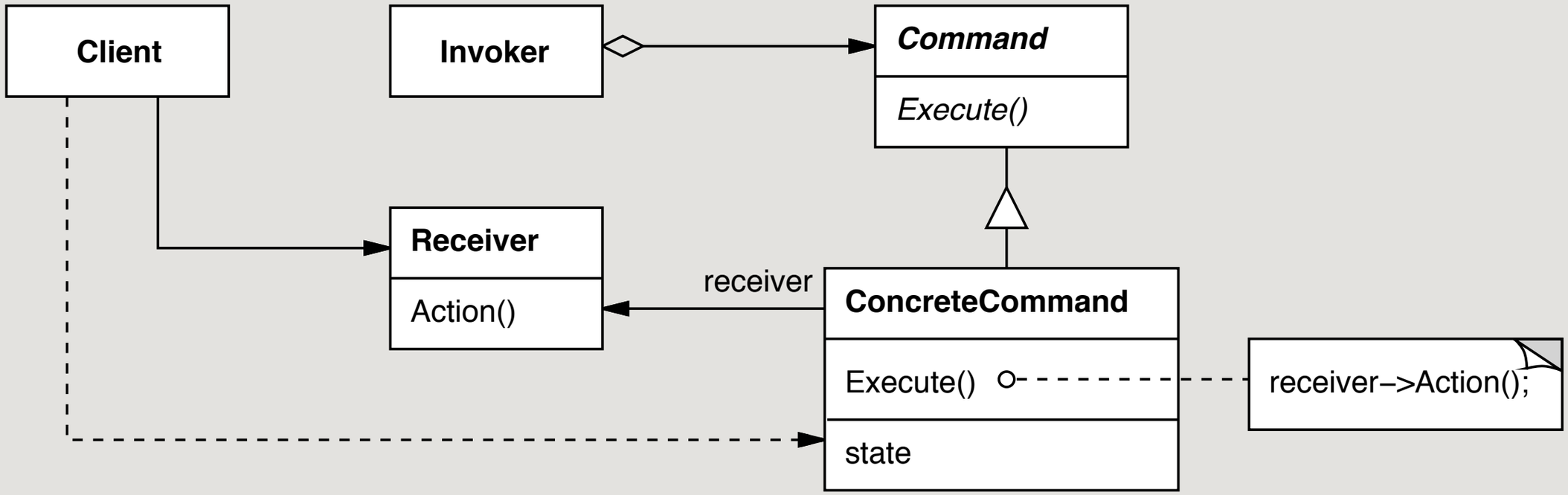
Aplicabilidad

- **Úsese el patrón *Command* cuando se quiera:**
  - Parametrizar objetos con una determinada acción
    - Lo que haríamos en un lenguaje de procedimientos con una función callback (como un puntero a función, que será llamada más tarde)
- **Especificar, guardar y ejecutar la petición en distintos momentos**
  - Es decir, que la acción a realizar y el objeto que la crea tengan ciclos de vida distintos («desacoplamiento temporal»)
- **Permitir deshacer/repetir («undo/redo»)**
  - En ese caso, `execute` deberá guardar el estado para poder revertir los efectos de ejecutar la operación
  - Y hará falta una operación añadida, `unexecute`

Aplicabilidad

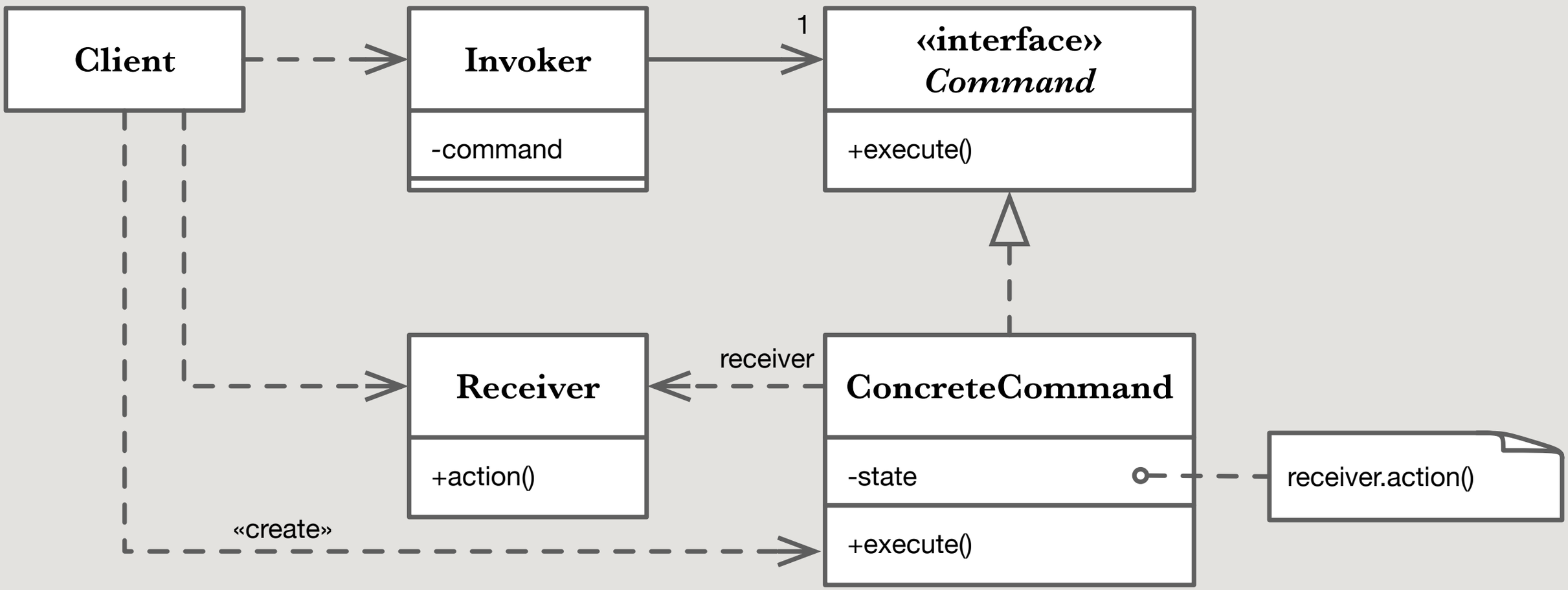
- **Guardar todas las operaciones ejecutadas en un registro («log»)**
  - Proporcionando un par de operaciones `store` y `load`
  - El sistema podría recuperarse de un error cargando las operaciones guardadas en disco y volviendo a llamar al método `execute` de cada una de ellas
- **Usar transacciones**

Estructura



La estructura del patrón *Command*, tal como aparece en el GoF

Estructura



La estructura del patrón *Command*, en UML (y corregida)

Participantes

- **Command**
  - Define una interfaz para ejecutar una operación
- **ConcreteCommand (PasteCommand, OpenCommand)**
  - Asocia un objeto Receiver con una acción
  - Implementa execute llamando a las operaciones de dicho objeto receptor

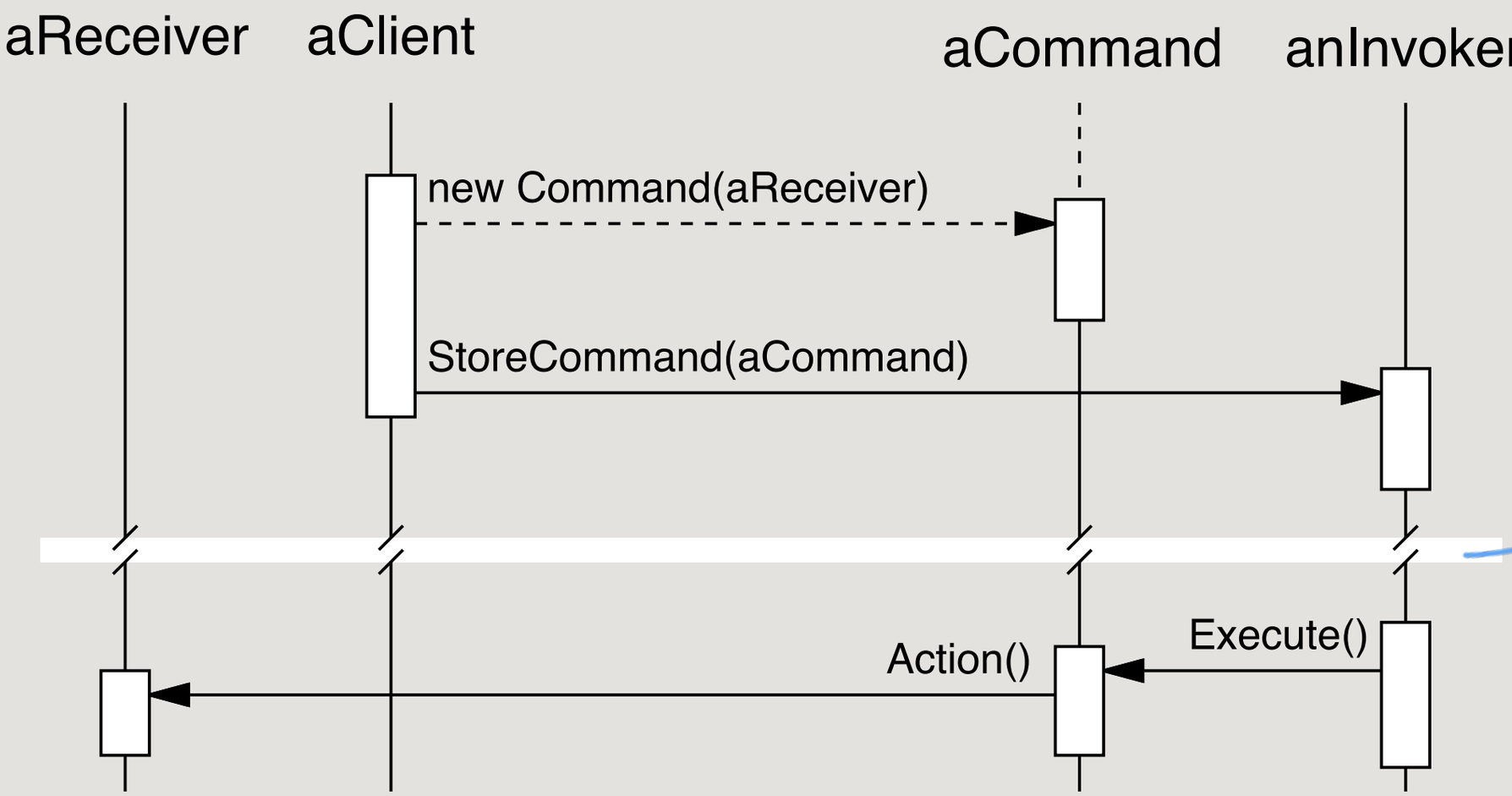
Participantes

- **Client (Application)**
  - Crea un objeto ConcreteCommand y establece su receptor
- **Invoker (MenuItem)**
  - Le pide al Command que se ejecute
- **Receiver (Document, Application)**
  - Quien realmente lleva a cabo la acción

Colaboraciones

- **El cliente crea un objeto `ConcreteCommand` y especifica su receptor**
- **Un objeto `Invoker` guarda el objeto `ConcreteCommand`**
- **Aquél llama a la operación de este último**
  - Quien antes guarda el estado para luego poder deshacer la operación (si son operaciones que se pueden deshacer)
- **El objeto `ConcreteCommand` se vale de las operaciones de su receptor para llevar a cabo la acción**

Colaboraciones



Este desacoplamiento temporal es fundamental, la esencia del patrón: es necesario guardarlo en algún sitio (el «invoker», normalmente distinto de quien lo crea) para ejecutarlo en algún momento posterior.

Consecuencias

- Desacopla el objeto que llama a la operación del que sabe cómo llevarla a cabo
- Son ciudadanos «de primera clase» (objetos)
- Se pueden ensamblar (*Composite*)
- Resulta sencillo añadir nuevas acciones, al no tener que tocar las clases existentes

Implementación

- **¿Cómo de inteligente debería ser?**
  - Desde un mero enlace entre el receptor y las operaciones a realizar en él hasta implementarlo todo él solo sin especificar un receptor
- **Diferentes niveles de deshacer/repetir**
  - Por ejemplo, mediante una lista que haga las veces de historial, recorriéndola en ambos sentidos
  - A veces será necesario crear una copia del objeto «command» antes de guardarlo en el historial (si su estado puede variar entre distintas invocaciones al método ejecutar)
    - En este caso los *Command* serían también *Prototype*, otro patrón que ya veremos

Ejemplo en Swing

Sin Command

```
public void actionPerformed(ActionEvent e)
{
  Object o = e.getSource();
  if (o = fileNewMenuItem)
    doFileNewAction();
  else if (o = fileOpenMenuItem)
    doFileOpenAction();
  else if (o = fileOpenRecentMenuItem)
    doFileOpenRecentAction();
  else if (o = fileSaveMenuItem)
    doFileSaveAction();
  // and more ...
}
```

Mal

Ejemplo en Swing

Usando «Action»

```
// create our actions
cutAction = new CutAction("Cut", cutIcon, "Cut stuff onto the clipboard",
                          new Integer(KeyEvent.VK_CUT));
copyAction = new CopyAction("Copy", copyIcon, "Copy stuff to the clipboard",
                          new Integer(KeyEvent.VK_COPY));
pasteAction = new PasteAction("Paste", pasteIcon,
                          "Paste whatever is on the clipboard",
                          new Integer(KeyEvent.VK_PASTE));

// create our main menu
JMenu fileMenu = new JMenu("File");
JMenu editMenu = new JMenu("Edit");

// create our menu items, using the same actions the toolbar buttons use
JMenuItem cutMenuItem = new JMenuItem(cutAction);
JMenuItem copyMenuItem = new JMenuItem(copyAction);
JMenuItem pasteMenuItem = new JMenuItem(pasteAction);

// add the menu items to the Edit menu
editMenu.add(cutMenuItem);
editMenu.add(copyMenuItem);
editMenu.add(pasteMenuItem);
```

http://alvinalexander.com/java/java-action-abstractaction-actionlistener