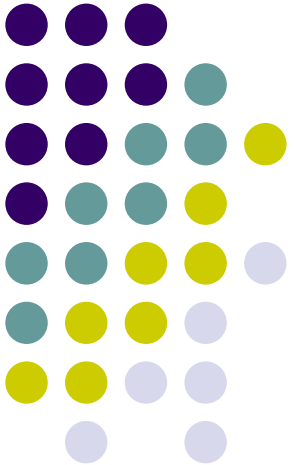
Patrones de Diseño

Patrón de comportamiento *Command*



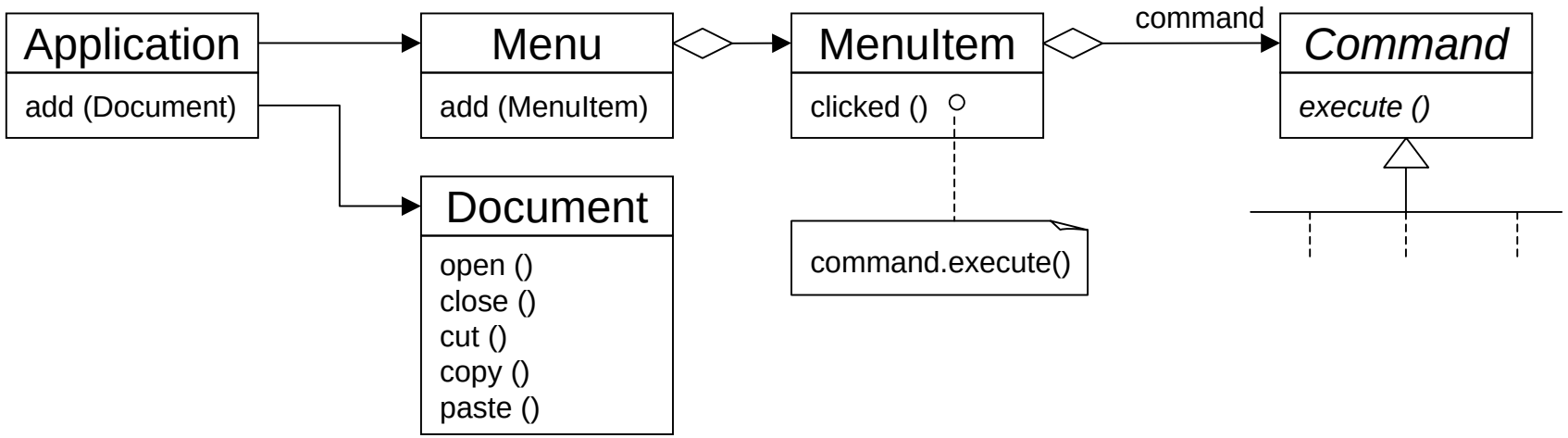
Command
Propósito

- Encapsular peticiones en forma de objetos.
- Permite parametrizar a los clientes con distintas peticiones, hacer colas de peticiones, llevar un registro de la peticiones realizadas, y deshacer el efecto de las peticiones.
- También conocido como *action*, *transaction*

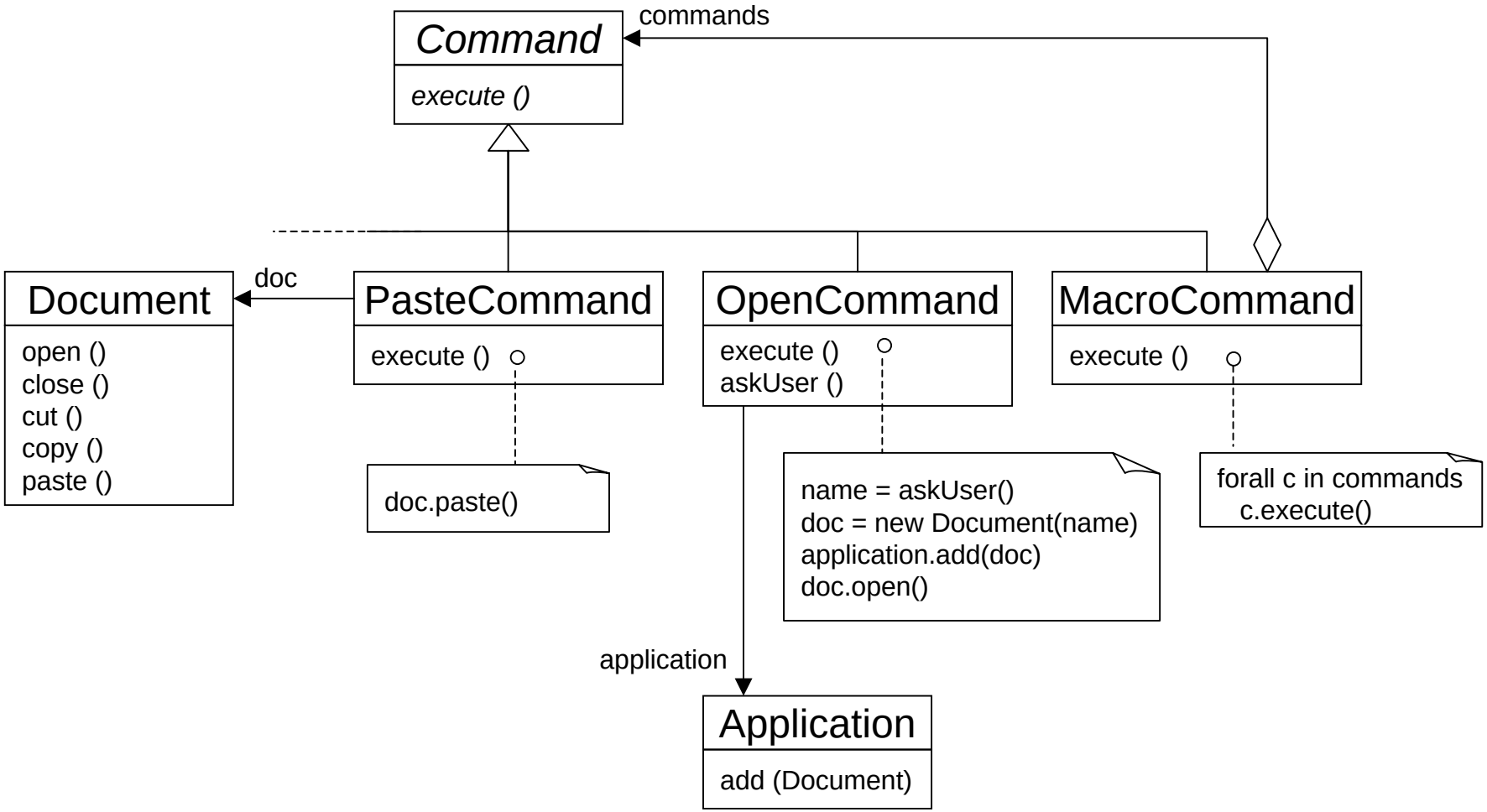
Command
Motivación

- Ej: framework gráfico con objetos como botones, menús ... que realizan operaciones en base a eventos de usuario. El comportamiento del objeto depende de la aplicación

- Solución:
  - Encapsular las peticiones como objetos



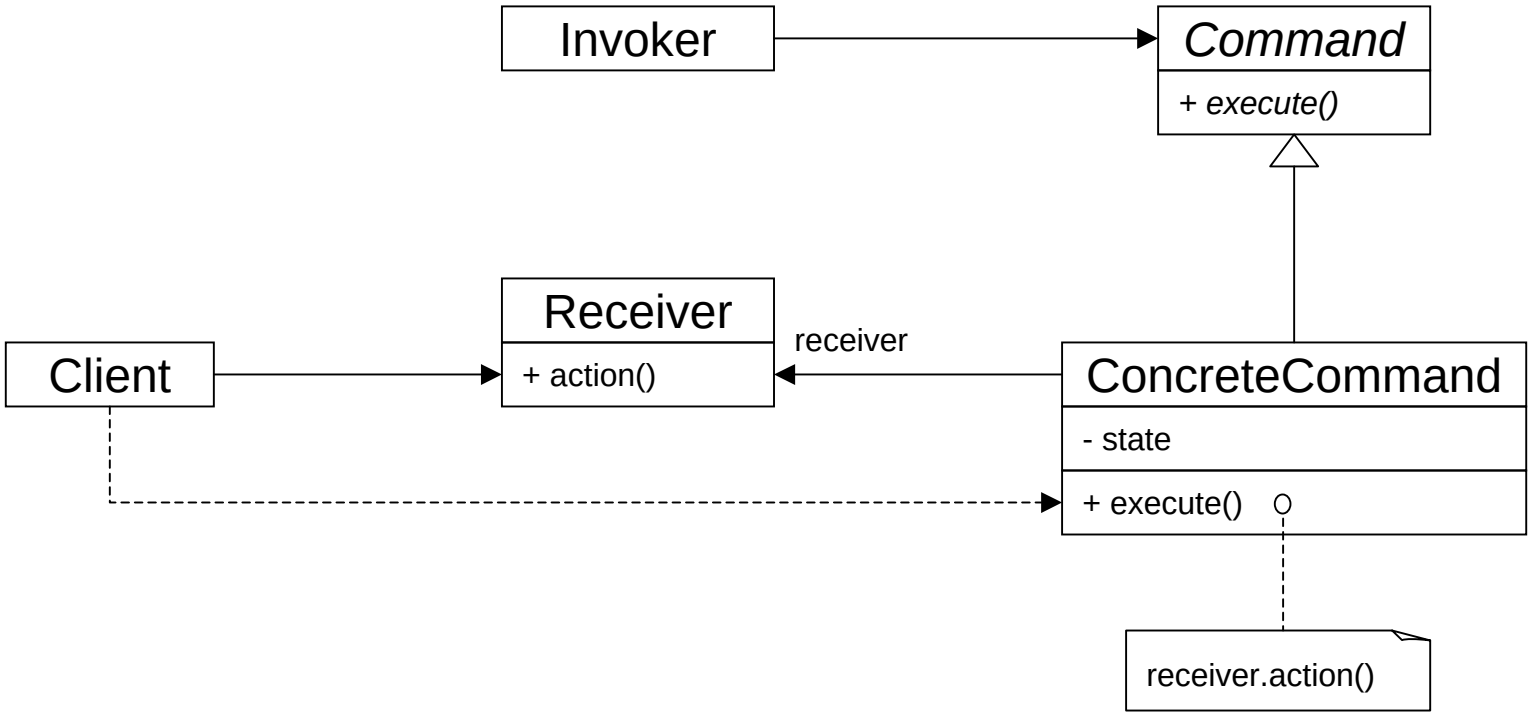
Command
Motivación



Command
Aplicabilidad

- Usa el patrón *Command* si quieres:
  - Parametrizar objetos mediante una acción
  - Especificar, encolar y ejecutar peticiones en distintos momentos (el objeto *command* tiene un tiempo de vida distinto de la petición)
  - Operaciones deshacer (método adicional en *command*)
  - Acciones de recuperación del sistema (métodos adicionales *salvar* y *recuperar* en *command*)
  - Interfaz común que permita invocar las acciones de modo uniforme, y extender el sistema con nuevas acciones de forma sencilla

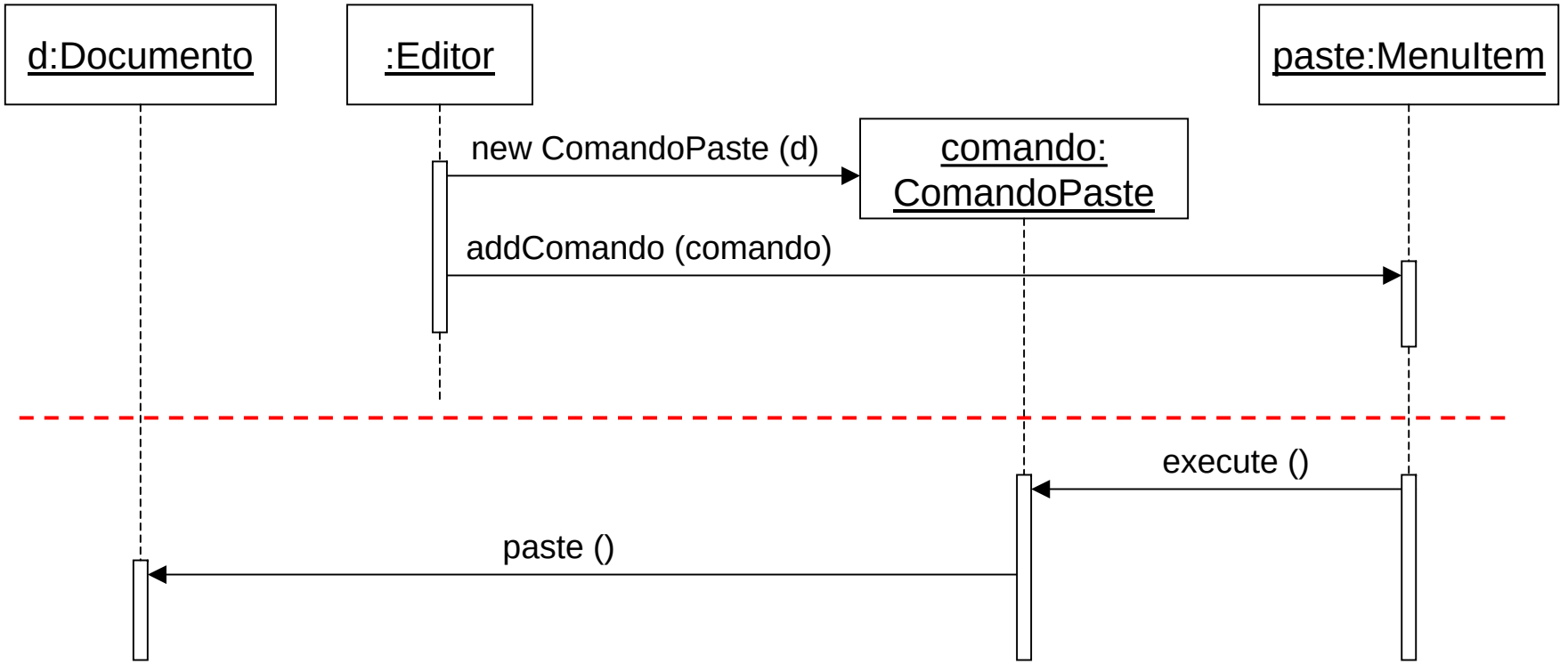
Command
Estructura



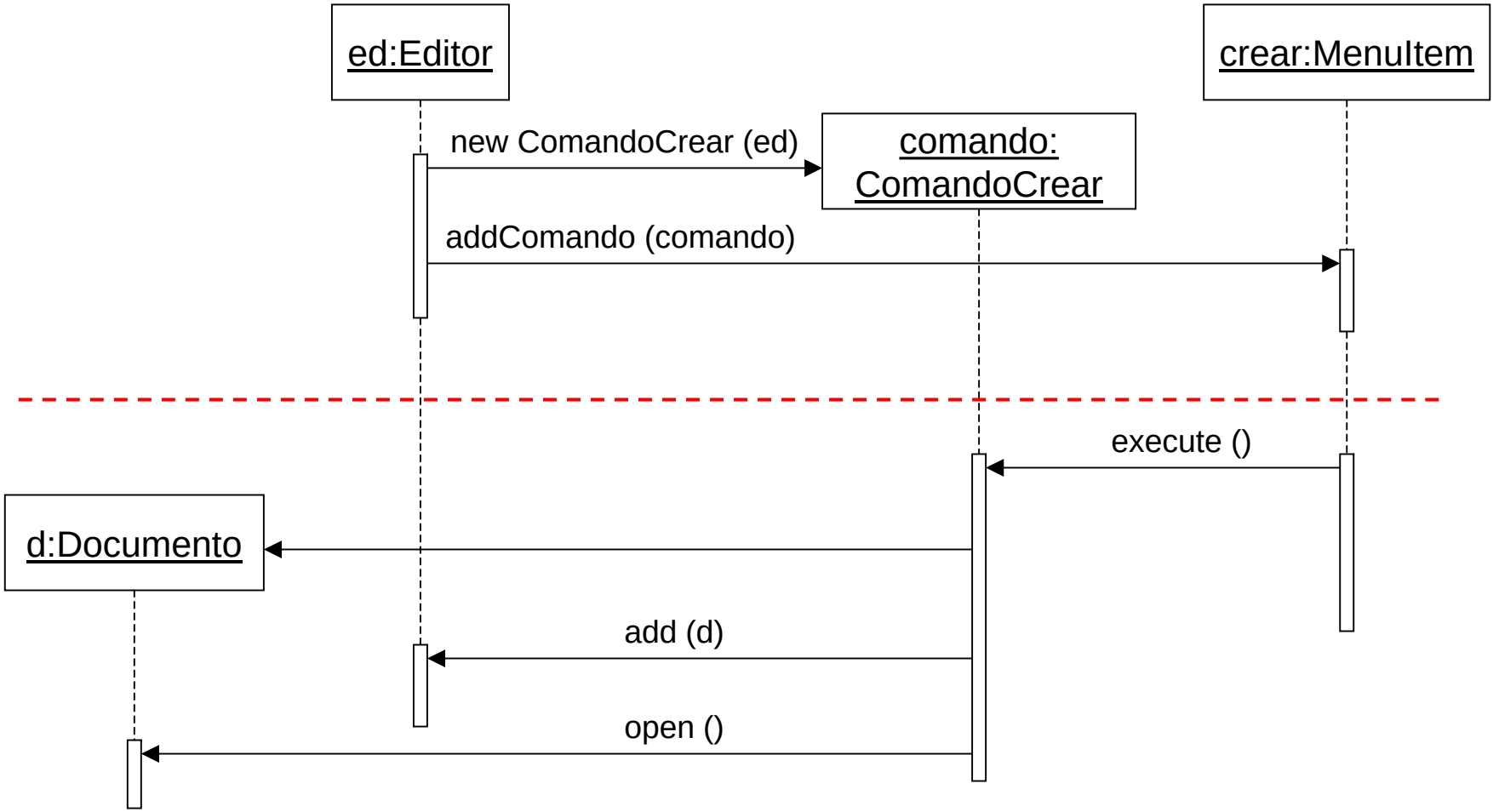
Command
Participantes

- **Command**: define la interfaz de ejecución de operaciones
- **ConcreteCommand** (*PasteCommand*, *OpenCommand*): Implementa la interfaz de ejecución invocando operaciones del receptor. De este modo relaciona una acción con un receptor
- **Client** (*Application*)*:* crea un comando concreto e indica a quién va dirigido (receptor)
- **Invoker** (*MenuItem*)*:* contiene el comando asociado a la petición
- **Receiver** (*Document*, *Application*): sabe cómo realizar las operaciones asociadas a una petición. Cualquier clase puede actuar como receptor.

Command
Colaboraciones (ejemplo)



Command
Colaboraciones (ejemplo)



Command
Consecuencias

- Desacopla el objeto que invoca la operación del que sabe cómo llevarla a cabo
- Los comandos son entidades de “primer orden”: se pueden manipular y extender como cualquier otro objeto
- Se pueden crear macro-comandos (patrón *composite*)
- Es fácil añadir nuevos comandos ya que no es necesario cambiar las clases existentes

Command
Implementación

- ¿Cómo de inteligente debe ser un comando? 2 extremos:
  - Invocan una acción en el receptor
  - Implementan el comportamiento sin delegar: útil para comandos independientes de un receptor, o cuando el receptor está implícito

- ¿Cómo implementar las operaciones *undo* y *redo*?
  - Operaciones adicionales en *command*
  - Almacenar el estado previo a la ejecución del comando: objeto receptor, argumentos de la operación, valores del receptor que puedan cambiar tras ejecutar el comando
    - Almacenar el estado anterior a la ejecución del comando permite hacer un *undo*
    - Almacenar todos los estados anteriores permite hacer n *undo*

Command
Código de ejemplo


```
public interface Command {
   public void execute();
}

public class CommandGenerarNominas implements Command {
   private Universidad _universidad;
   public CommandGenerarNominas (Universidad universidad) {
      _universidad = universidad;
   }
   public void execute() {
      _universidad.generarNominas();
   }
}

public class MenuUniversidad {
   public boolean menuPrincipal {
      ...
      case 3: // generar nóminas
              // _universidad.generarNominas();
         Command comando = new CommandGenerarNominas(_universidad);
         comando.execute();
         break;
   }
}
```

Command
Ejercicio

- **Se quiere desarrollar una clase que ejecute periódicamente tareas de mantenimiento del sistema (ej. realizar copias de seguridad del sistema, ejecutar el antivirus, etc.)**
- **Cada tarea está implementada como un método de alguna clase existente**
- **Se quiere que la nueva clase tenga un bajo acoplamiento con las clases que implementan dichas tareas**

Command
Solución