Patrones de Comportamiento

- Propósito:
  - Encapsular “lo que varía”
    - Cuando **un comportamiento** varía con frecuencia se encapsula con un objeto
  - Asignación de responsabilidad ➔ Distribuir el comportamiento.
  - Comunicación entre instancias.
  - Se usa mas la composición que la herencia
- ¿Cuáles veremos?
  - Command
  - State
  - Observer
  - Chain of Responsibility
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Template Method
  - Visitor

Command – Patrón de Comportamiento
Orden, Action (Acción), Transaction (Transacción)

- Propósito
  - Necesito invocar operaciones pero…
    - El objeto (Emisor) que crea la petición no es el que lo ejecuta
    - El Emisor pasa la solicitud a otro objeto (Receptor)
      - El Receptor no necesita saber que operación es
      - Sólo sabe como lanzarla
      - Y no tiene porque ejecutar inmediatamente la operación
    - **La misma operación** se puede ejecutar en diferentes Receptores
      - Ej: Una misma opción disponible desde el menú y la botonera
    - Las operaciones a ejecutar son varias y queremos **variar el orden de ejecución (sin recompilar)**

Command – Escenario 1

cliente

```
ordenA ( ArgsA )
ordenB ( ArgsB )
Metodo(...){
ordenC ( ArgsC )
}
```

Sabe que operaciones hay que ejecutar

Receptor1

```
Metodo1(...){

}
```

¿Quién y Cuándo se van a ejecutar?

Receptor2

```
Metodo2(...){

}
```

Command – Escenario 2

cliente

```
Metodo(...){
  ordenA
  ordenB
  ordenC
}
```

SOLUCIÓN
MANEJAR CÓDIGO COMO SI FUERAN DATOS
(Patrón Command)

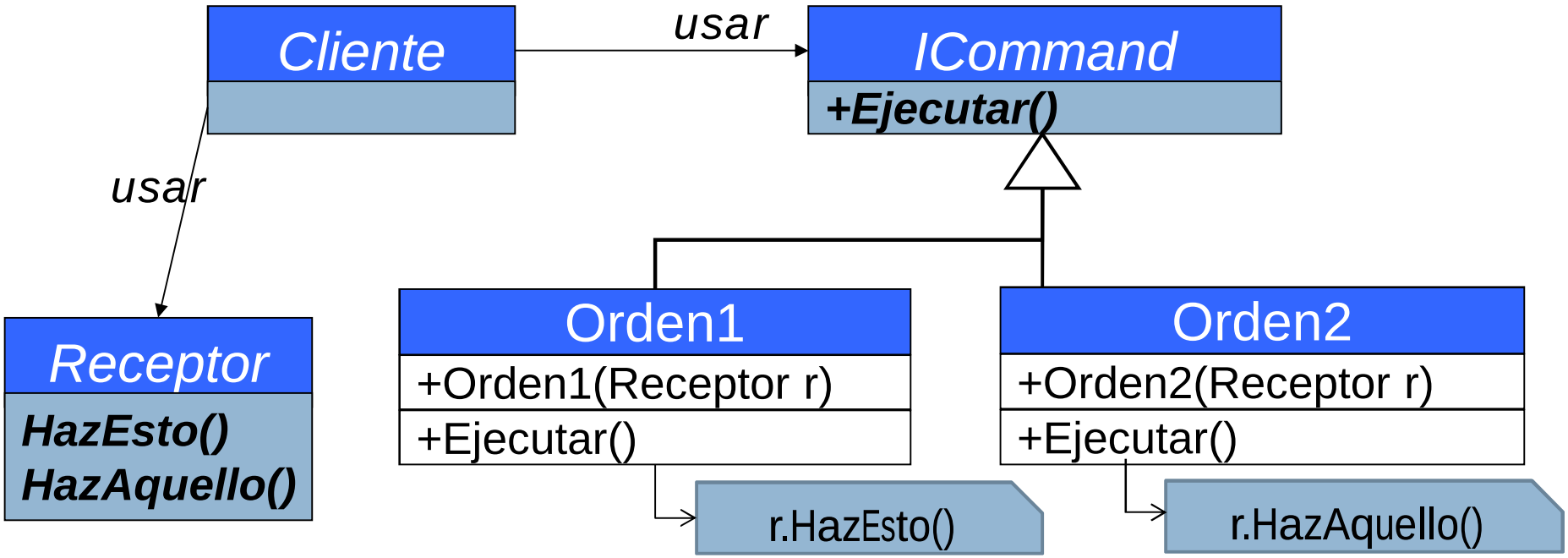
Cliente es el responsable de ejecutarlas y cuando …

… cuando …

Sabe que operaciones hay que ejecutar

…y el "qué" ejecutar

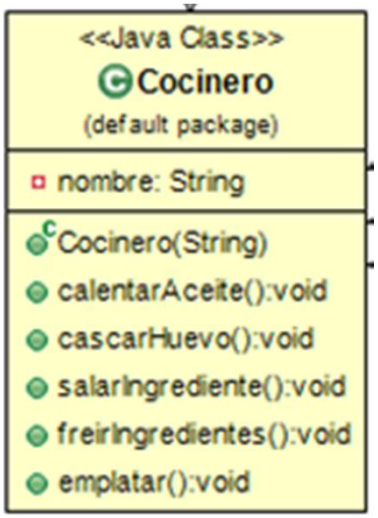
Command – Estructura



- Los objetos Orden (Command) reciben en su constructor el *Receptor* actual del problema sobre el que ejecutar la acción
- El Cliente que crea la Orden **puede ser distinto** del que finalmente la ejecuta

Command – Ejemplo: Recetas

- Cocinando huevos fritos




```
/**
 * Método directo para que un cocinero fría un huevo
 * @param elCocinero - el cocinero responsable de la receta
 */
private static void freirHuevo(Cocinero elCocinero)
{
    elCocinero.calentarAceite();
    elCocinero.cascarHuevo();
    elCocinero.salarIngrediente();
    elCocinero.freirIngredientes();
    elCocinero.emplatar();
}
```

Prepara Huevo

Fríe

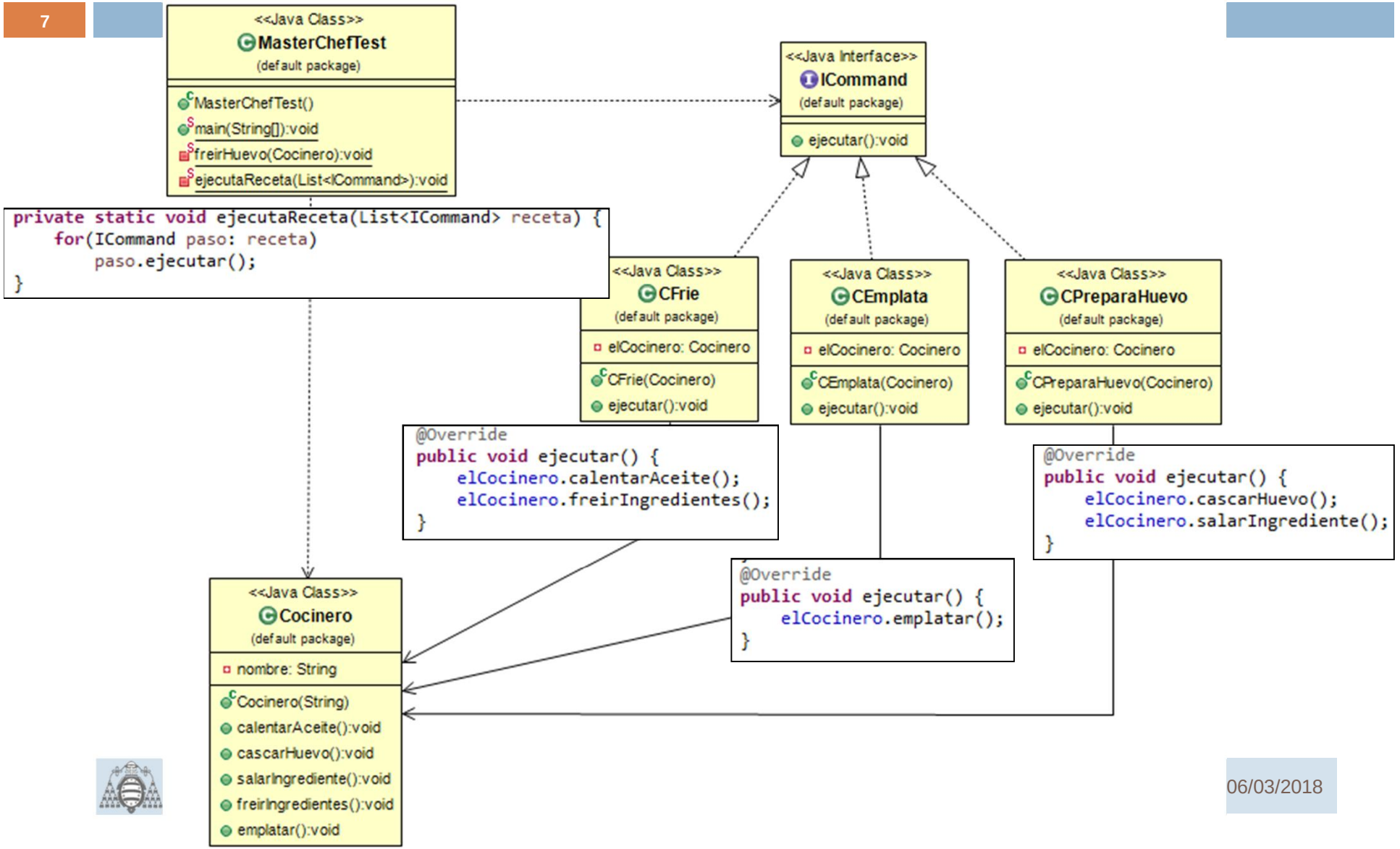
Emplatar

¿Orden Correcto?

¿Solo 1 cocinero?

```
Cocinero pinche1 = new Cocinero("pinche1");
// Un huevo frito: el pinche 1 hace todo (prepara, frie y emplata)
List<ICommand> recetaHuevoFrito = new ArrayList<ICommand>();
// pasos de la receta
recetaHuevoFrito.add(new CPreparaHuevo(pinche1));
recetaHuevoFrito.add(new CFrie(pinche1));
recetaHuevoFrito.add(new CEmplata(pinche1));
```

```
// Un huevo frito
ejecutaReceta(recetaHuevoFrito);
```

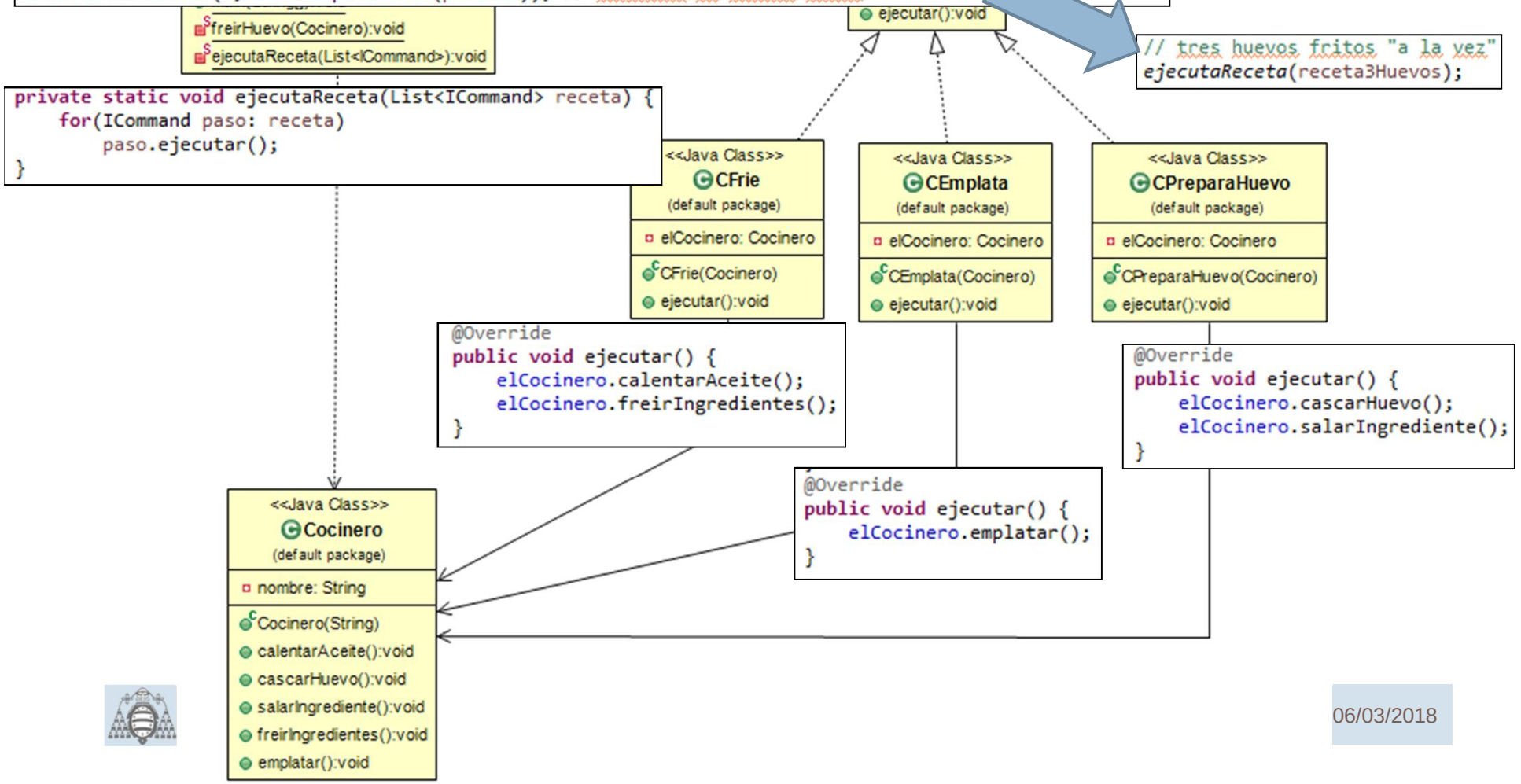
plo: Recetas

```
Cocinero pinche1 = new Cocinero("pinche1");
// Un huevo frito: el pinche 1 hace todo (prepara, frie y emplata)
List<ICommand> recetaHuevoFrito = new ArrayList<ICommand>();
// pasos de la receta
recetaHuevoFrito.add(new CPreparaHuevo(pinche1));
recetaHuevoFrito.add(new CFrie(pinche1));
recetaHuevoFrito.add(new CEmplata(pinche1));
```

```
Cocinero pinche2 = new Cocinero("pinche2");
Cocinero pinche3 = new Cocinero("pinche3");
List<ICommand> receta3Huevos = new ArrayList<ICommand>(recetaHuevoFrito); // copio la receta de 1 huevo
// nuevos pasos de la receta
receta3Huevos.add(0,new CPreparaHuevo(pinche2)); // añadimos un segundo huevo
receta3Huevos.add(0,new CPreparaHuevo(pinche3)); // añadimos un tercer huevo
```

```
// tres huevos fritos "a la vez"
ejecutaReceta(receta3Huevos);
```

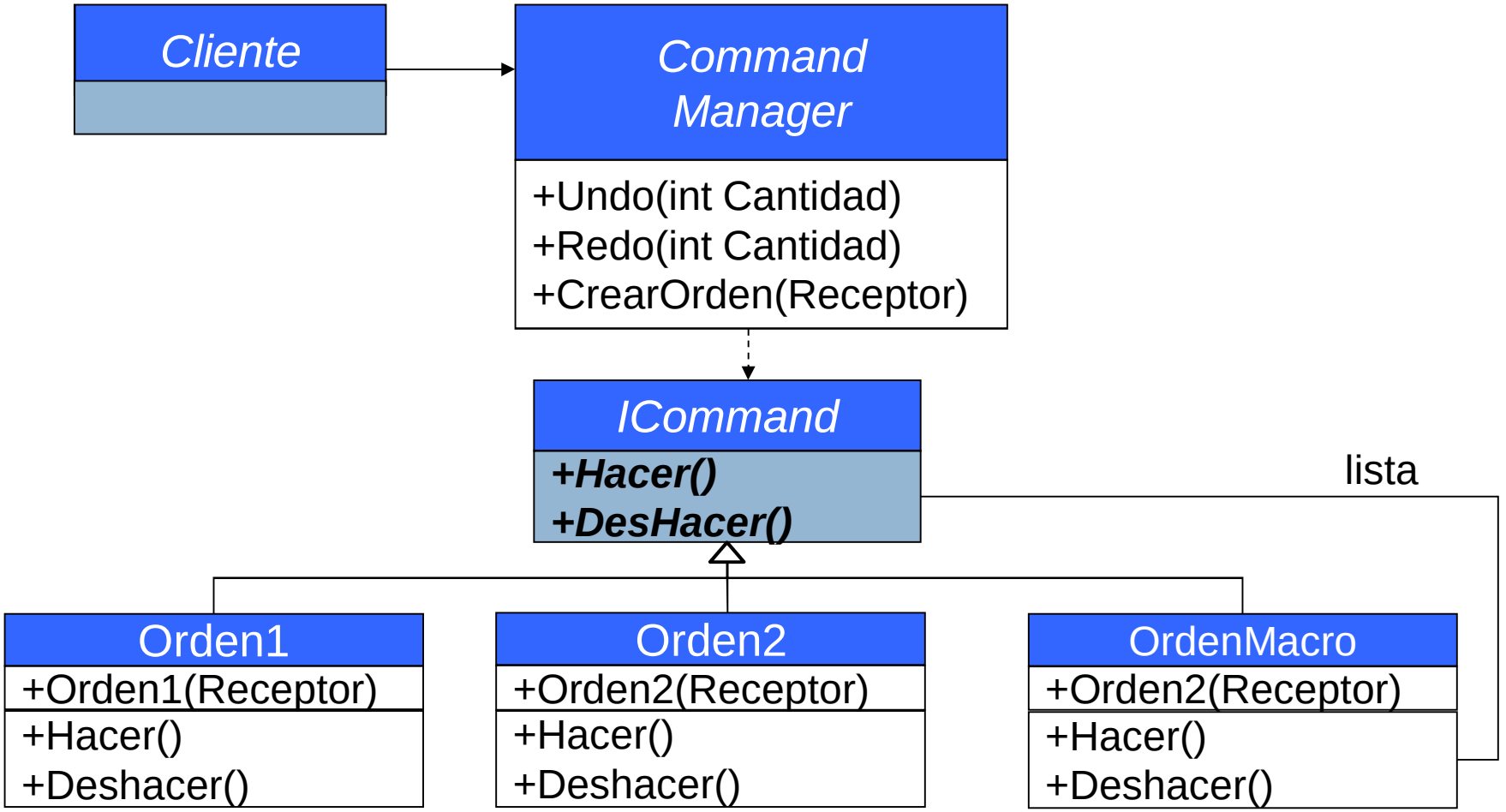
```
private static void ejecutaReceta(List<ICommand> receta) {
    for(ICommand paso: receta)
        paso.ejecutar();
}
```



Command - Alternativas

- También puedo:
  - Crear un método Deshacer en la Interfaz.
  - Puedo crear una pila de las últimos Órdenes que se ejecutaron.
  - Puedo sacar de la pila de Órdenes ejecutadas y llamar al método Deshacer.
  - Combinando varios Command puedo generar una Orden Macro.
    - Sería útil utilizar el patrón estructural Composite

Command – Ejemplo con Composite



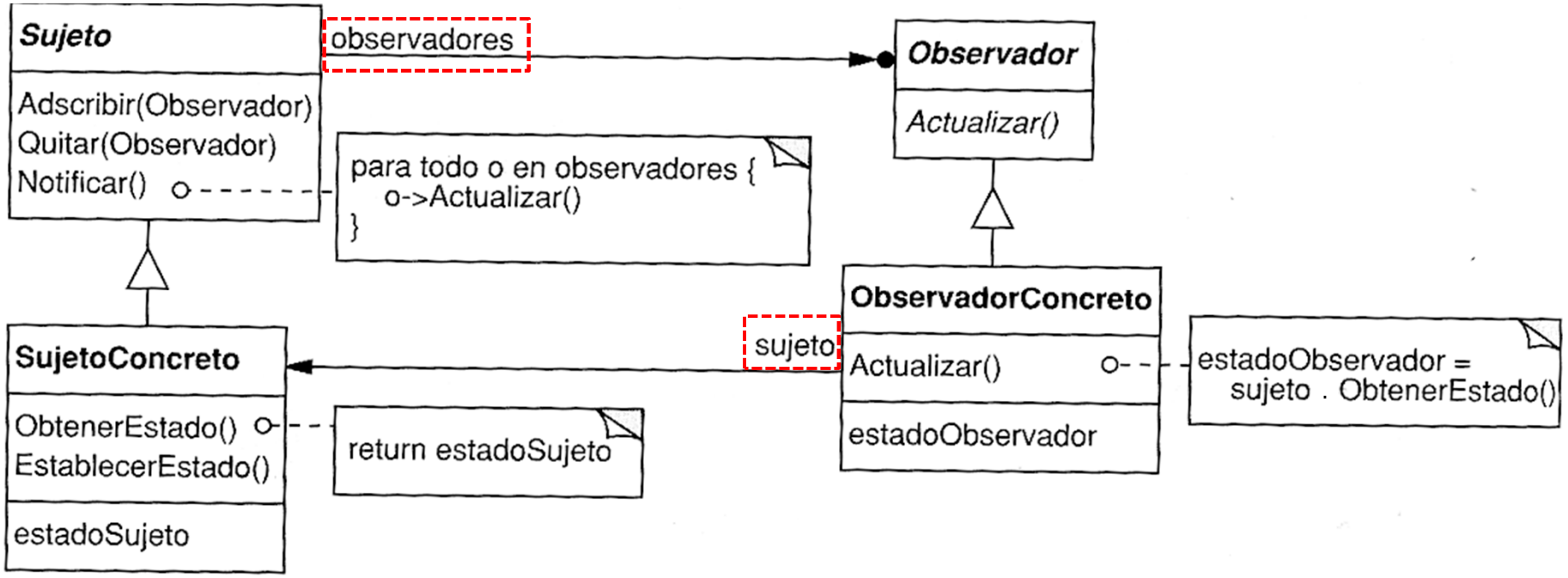
Command - Consecuencias

- Command desacopla el objeto que invoca la acción de aquel que sabe como realizarla.
- Las órdenes son ***objetos de primera clase***
  - Pueden manipularse y extenderse como otro objeto
- Se pueden crear Macro-Ordenes combinando varios objetos de tipo Command
  - Sería útil el patrón Composite
- Es fácil crear nuevas órdenes, sin tocar las clases existentes
  - Utilizamos para ello el mecanismo de herencia

Observer – Patrón de Comportamiento
(Observador, Dependents, Dependientes, Publish-Subscribe, Publicar-Suscribir)

- Propósito
  - Define una dependencia de uno-a-muchos entre objetos
  - Al cambiar un objeto automáticamente se notifica a todos los objetos que dependen de él: **sus observadores**
- Aplicabilidad
  - Una abstraccion tiene dos aspectos y uno depende del otro. Observer facilita que se puedan modificar y reutilizar por separado: **desacopla** el observador y el observado.
  - Cuando un cambio en un objeto requiera cambiar a otros, y a priori no sabemos cuantos necesitan cambiarse.
  - Cuando un objeto debería notificar a otros sin presuponer que objetos concretos son: **desacoplarlos**

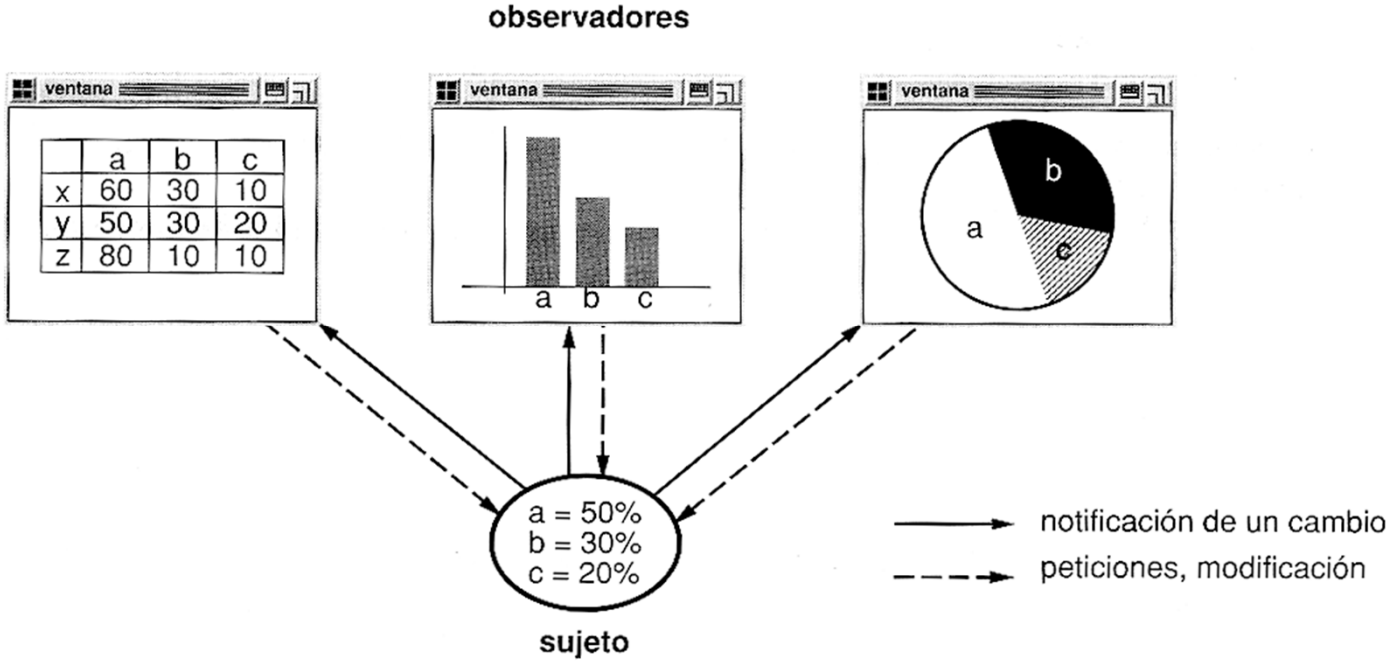
Observer - Estructura



- Sujeto tiene una colección de **observadores** modificable: Adscribir() y Quitar()
- Un ObservadorConcreto tiene una referencia **sujeto** al Sujeto observado. Así poder consultar: sujeto.ObtenerEstado()

Observer - Ejemplo

- Datos estadísticos
  - Hoja de cálculo (observador/modificador)
  - Gráfica de barras (observador)
  - Gráfico de tarta (observador)



Observer - Consecuencias

- Observer permite modificar los sujetos y sus observadores de forma independiente.
  - Podemos añadir Observadores sin modificar el Sujeto.
  - Podemos reutilizar un mismo Observador con otros Sujetos y viceversa.
- Comunicación por difusión
  - Al Sujeto le da igual cuantos o qué Observadores tiene a la hora de informar de cambios.
- Actualizaciones inesperadas
  - Los Observadores pueden modificar al Sujeto.
  - Una pequeña modificación puede disparar la "difusión" del cambio a múltiples Observadores, provocando una reacción en cadena con un alto coste computacional
- Sujetos complejos complican identificar "qué" cambió en ellos
  - El protocolo sólo indica que cambió el Sujeto, pero no el qué.

Back To The Future: 1955

Zenith Flash Matic: Primer mando a distancia TV

Evolución Mandos a distancia TV

TV clásica

TV (info en pantalla)

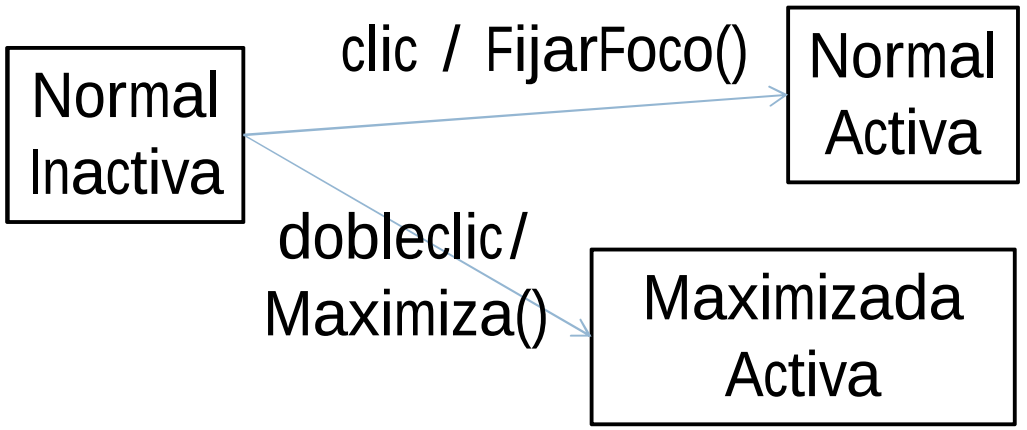
Smart TV

State – Patrón de Comportamiento (Objects for States, Estados como Objetos)

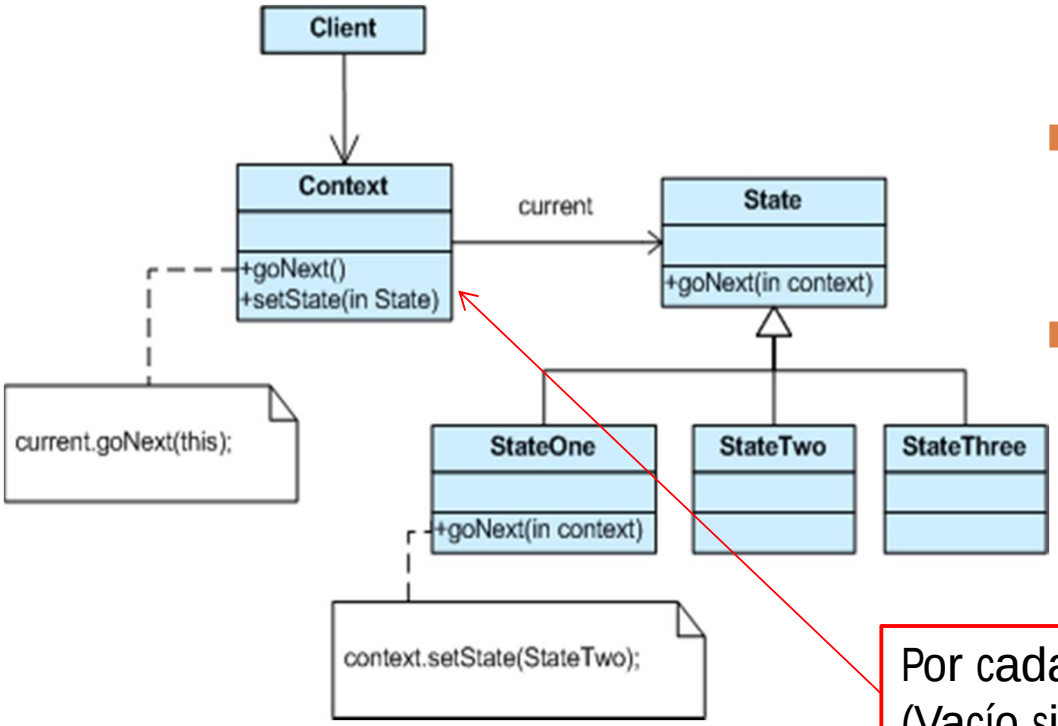
- Propósito y Aplicabilidad
  - Permite a un objeto modificar su comportamiento cada vez que cambie su estado interno: ***máquina de estados***
  - ***La complejidad de las transiciones hace inmanejable los cambios mediante sentencias if-then-else***
  - *No sólo el comportamiento cambia con cada estado sino que las transiciones implican realizar acciones especificas.*

Ventana GUI

Estado	Cambio	Acción	Nuevo Estado
Normal Inactiva	Clic	Fijar foco	Normal Activa
Normal Inactiva	Dobleclic	Maximizar	Maxim. Activa



State – Estructura

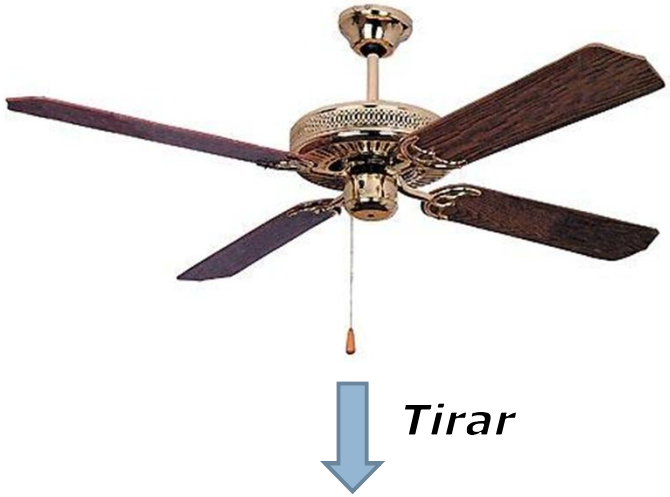


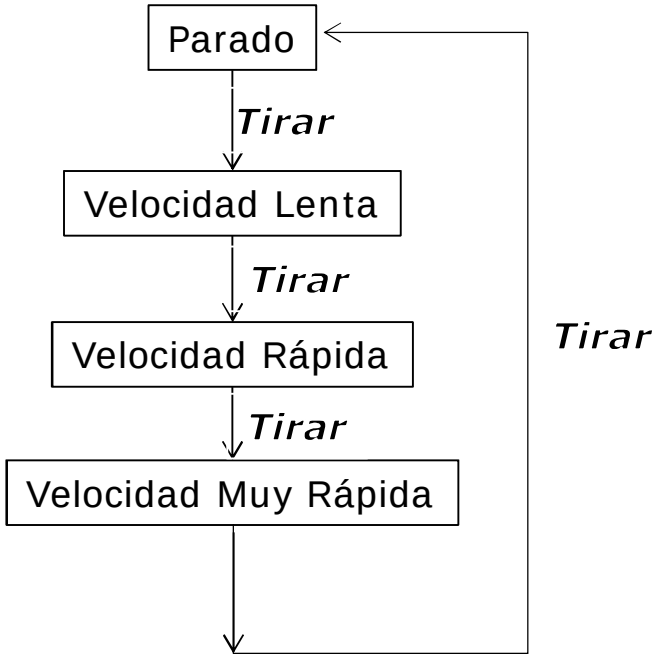
- Participantes
  - Context:
    - Único objeto manejado por el Cliente
  - State:
    - Define interfaz de comportamiento de un estado
  - Subclases de State:
    - Implementación de un comportamiento de un estado concreto

Por cada posible evento N habría un goNext-N()
(Vacío si no produce cambio en el estado actual)

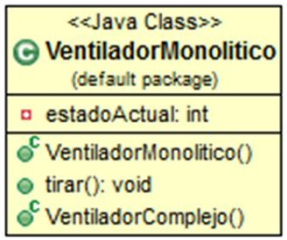
State – Ejemplo de máquina de estados

- Ventilador 3 velocidades y un solo control





Ventilador – versión monolítica



Cliente

```
VentiladorMonolitico v =
 new VentiladorMonolitico();

v.tirar(); // se enciende
v.tirar(); // acelera
v.tirar(); // acelera más
v.tirar(); // acelera mucho más
v.tirar(); // se apaga
...
```

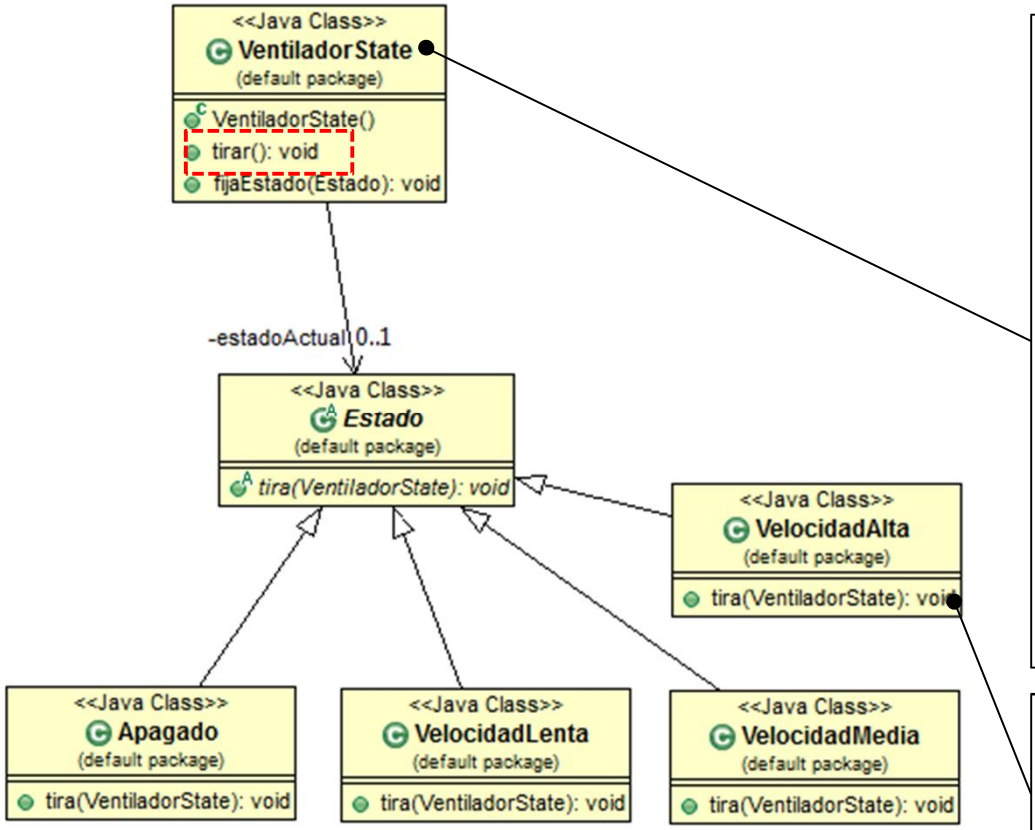
```
public class VentiladorMonolitico {

   private int estadoActual; // estado actual

   public VentiladorMonolitico()
   {
      estadoActual = 0;  // estado inicial
   }

   public void tirar() // transiciones
   {
      if (estadoActual == 0){
         estadoActual = 1;
         System.out.println("   pasando a velocidad lenta");
      }
     else if (estadoActual == 1){
         estadoActual = 2;
         System.out.println("   pasando a velocidad media");
      }
      else if (estadoActual == 2){
         estadoActual = 3;
         System.out.println("   pasando a velocidad alta");
      }
      else {      estadoActual = 0;
                    System.out.println("   apagando"); }
   }
}
```

Ventilador – versión State




```
public class VentiladorState {

 private Estado estadoActual;

 public VentiladorState() {
   fijaEstado(new Apagado());
 }
 public void tirar() {
   estadoActual.tira(this);
 }
 public void fijaEstado(Estado e){
   estadoActual = e;
  }
}
```

```
public void tira(VentiladorState v) {
  System.out.println("   apagando");
  v.fijaEstado(new Apagado());
}
```

- Cambiar de estado es reemplazar un objeto **Estado** por otro
- El programa cliente que use el Ventilador **no ha cambiado**
- Podemos **enriquecer la clase Estado** (ej: int velocidadActual)

State - Consecuencias

- Hace explícitas las transiciones entre estados
  - Cada estado es un objeto
  - Cada estado define sus transiciones a otros estados
  - **Robustez**: Los cambios son atómicos (indivisibles)
    - Ahora el Contexto sólo cambia una variable: su **Estado**
  - Los objetos Estado pueden compartirse
    - Cuando todos los estados **no** tienen variables internas, puede compartirse un mismo objeto estado por varios Contextos
  - Los objetos Estado pueden ser únicos
    - En lugar de generar uno nuevo cada vez se reutiliza el mismo objeto (Singleton)