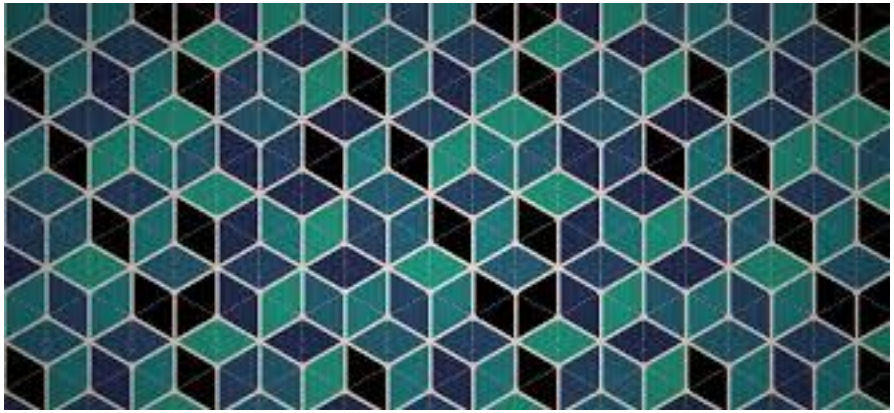
Orientación a Objetos 2



Patrones de diseño

Strategy, State

Gabriela Pérez
gperez@lifia.info.unlp.edu.ar

¿Qué cosa es importante estudiar y recordar?

- Intent / Propósito
- Applicability / (Uso el patrón cuando...)

- Estructura / solución:
  - clases que componen el patrón (roles),
  - cómo se relacionan (jerarquías, clases abstractas/interfaces, métodos abstractos – protocolo de interfaces, conocimiento/composición)
- Consecuencias
  - positivas y negativas
- Implementación
- Relación con otros patrones

Nuevo patrón de comportamiento

(caracteriza las formas en las que las clases y/o los objetos interactúan y distribuyen responsabilidades)

Ejemplo 1: App de compresión de archivos

Una posible solución

- La aplicación debe permitir a los usuarios comprimir y descomprimir archivos

- Por ahora se comprime ZIP, RAR
- Se planean agregar nuevas formas de compresión y permitir cambiarlas en tiempo de ejecución

Ejemplo 1: App de compresión de archivos

Una posible solución

- La aplicación debe permitir a los usuarios comprimir y descomprimir archivos

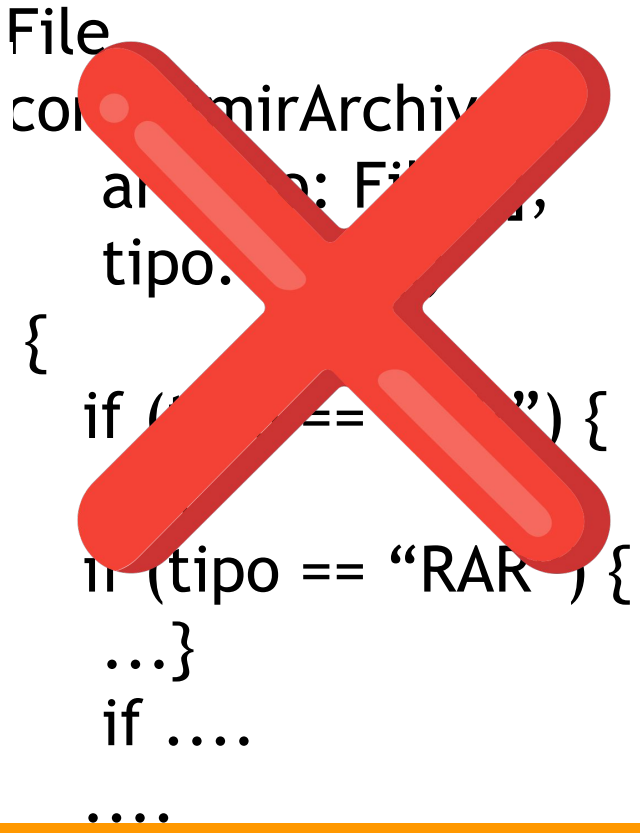

```
File
comprimirArchivos(
     archivo: File[*],
     tipo: String)
 {
    if (tipo == “ZIP”) {
     ...}
    if (tipo == “RAR”) {
     ...}
     if ....
    ....
```

- Por ahora se comprime ZIP, RAR
- Se planean agregar nuevas formas de compresión y permitir cambiarlas en tiempo de ejecución

Ejemplo 1: App de compresión de archivos

Una posible solución

- La aplicación debe permitir a los usuarios comprimir y descomprimir archivos



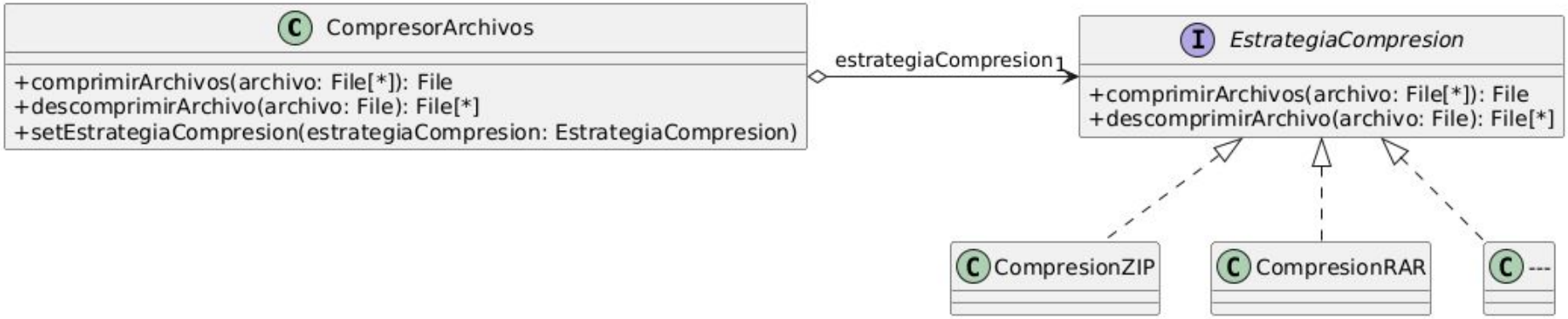
- Por ahora se comprime ZIP, RAR
- Se planean agregar nuevas formas de compresión y permitir cambiarlas en tiempo de ejecución

Una mejor solución

- Separar lo que varía
- "Objetificar" el algoritmo

- Por ahora se comprime ZIP, RAR
- Se planean agregar nuevas formas de compresión y permitir cambiarlas en tiempo de ejecución

Ejemplo 1: App de compresión de archivos



Ejemplo 2: Delivery de comida

- Elegir la comida
- Elegir dirección y forma de envío
- Elegir el método de pago

- Solo una forma de pago, por ejemplo con tarjeta de crédito

```
public boolean pagar(float monto, String nroTarj, String vto, String cvv) {
    CreditCard tarjeta = new CreditCard(nroTarj, vto, cvv);
    //Validar tarjeta
    if (tarjeta.validate(monto)) {
      tarjeta.charge(monto);
      this.registrarPago();
      return true; }
    else {
      System.out.println("No funcionó el pago con tarjeta");
      return false; }
}
```

Ejemplo 2: Delivery de comida

- Elegir la comida
- Elegir dirección y forma de envío
- Elegir el método de pago

- Solo una forma de pago, por ejemplo con tarjeta de crédito

```
public boolean pagar(float monto, String nroTarj, String vto, String cvv) {
    CreditCard tarjeta = new CreditCard(nroTarj, vto, cvv);
    //Validar tarjeta
    if (tarjeta.validate(monto)) {
       tarjeta.charge(monto);
       this.registrarPago();
       return true; }
    else {
       System.out.println("No funcionó el pago con tarjeta");
```

- Y si fueran apareciendo nuevos métodos de pago?

Ejemplo 2: Delivery de comida

Una posible solución

Cómo agregamos nuevos métodos de pago?

```
public boolean pagar(float monto, String metodoPago) {
    if (metodoPago == “CreditCard”) {
        //Obtener datos tarjeta ...
        CreditCard tarjeta = new CreditCard(nroTarj, vto, cvv);
        //Validar tarjeta
        if (tarjeta.validate(monto)) {
      tarjeta.charge(monto);
      this.registrarPago();
      return true; }
       else {
      System.out.println(“No funcionó el pago con tarjeta”);
      return false; }
    }
    else if (metodoPago == “MercadoPago”) {
...
}
```

- Y si fueran apareciendo nuevos métodos de pago?

Ejemplo 2: Delivery de comida

Una posible solución

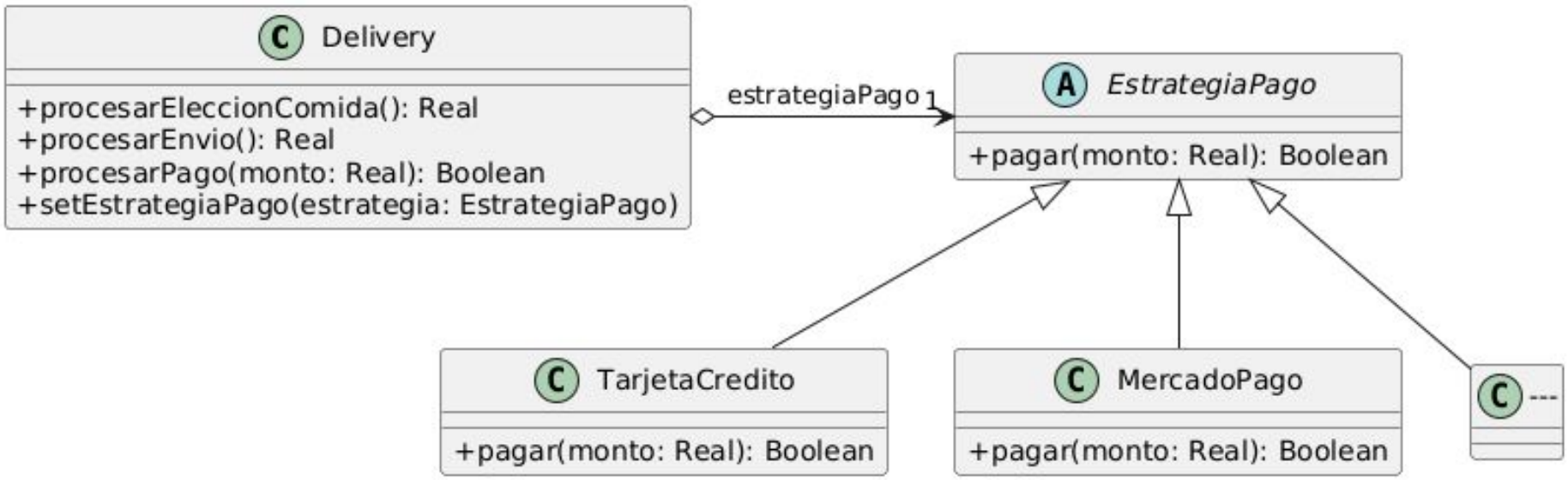
Cómo agregamos nuevos métodos de pago?

- IF!

```
public boolean pagar(float monto, String metodoPago) {
    if (metodoPago == "CreditCard") {
        //Datos tarjeta
        CreditCard tarjeta = new CreditCard(nroTarj, vto, cvv);
        //Validar
        if (tarjeta.validar()) {
            tarjeta.charge(monto);
            this.registrarPago();
            return true;
        } else {
            System.out.println("No fue posible pago con tarjeta");
            return false; }
    }
    else if (metodoPago == "MercadoPago") {
...
}
```

- Y si fueran apareciendo nuevos métodos de pago?

Ejemplo 2: Delivery de comida



¿Cuál es una mejor solución?

- Tengo un problema recurrente:
  - Cómo solucionamos este problema común de tener diferentes **algoritmos opcionales para realizar una misma tarea** (comprimir archivos, pagar, etc.)?

- El patrón Strategy propone
  - Encapsular cada algoritmo en un objeto y usarlos en forma intercambiable en tiempo de ejecución según se necesiten

-

Patrón Strategy

- **Intent**
- **Applicability**
- **Estructura**
- **Consecuencias**
- **Implementación**
- **Relación con otros patrones**

- Definir una familia de algoritmos, encapsular cada uno y hacerlos intercambiables.
- Permite que el **algoritmo varíe** independientemente de los clientes que lo usan.
- Permitir **cambiar (en forma dinámica)**, el algoritmo que se utiliza.
- Brindar flexibilidad para **agregar nuevos algoritmos** que lleven a cabo una función determinada.

Patrón Strategy

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

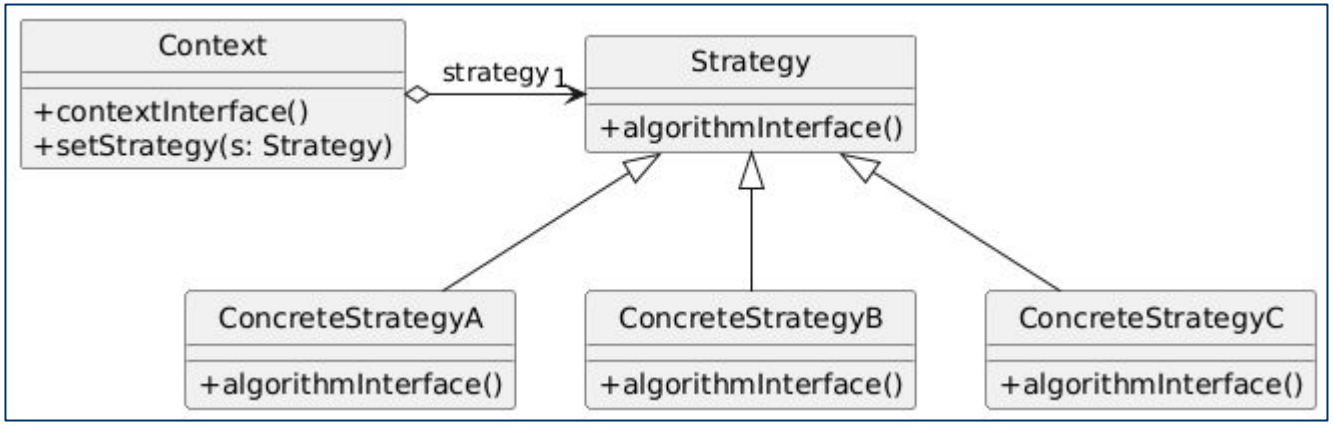
Uso el patrón Strategy cuando...

- Existen muchos algoritmos para llevar a cabo una tarea.
- No es deseable codificarlos todos en una clase y seleccionar cuál utilizar por medio de sentencias condicionales.
- Cada algoritmo utiliza información propia. Colocar esto en los clientes lleva a tener clases complejas y difíciles de mantener.
- Es necesario cambiar el algoritmo en forma dinámica, en tiempo de ejecución.

Patrón Strategy

- **Intent**
- **Applicability**
- **Estructura**
- **Consecuencias**
- **Implementación**
- **Relación con otros patrones**

- Definir una familia de algoritmos, encapsular cada uno en un objeto y hacerlos intercambiables.
- Son los <<clientes>> del contexto los que generalmente crean las estrategias.

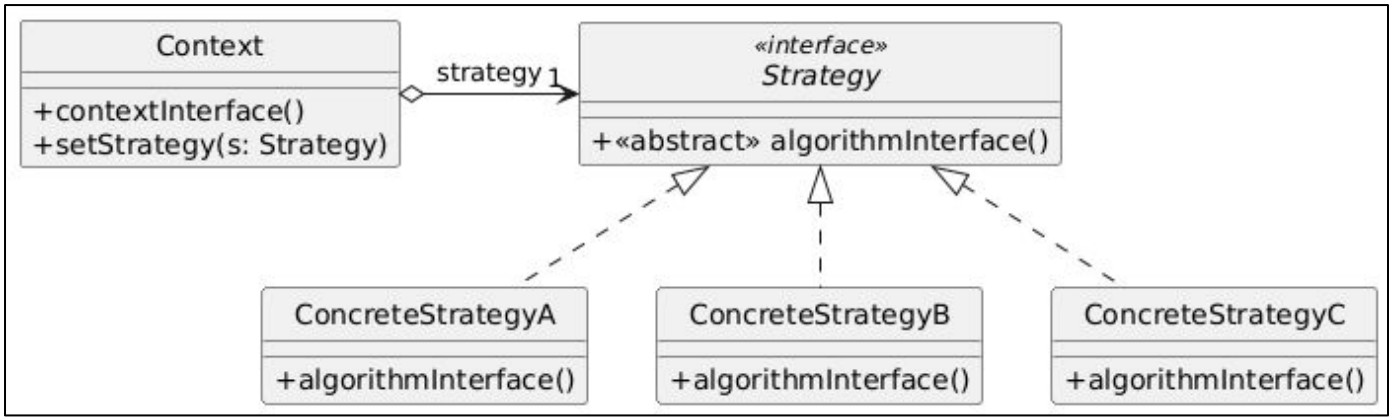


Patrón Strategy

Estructura con interfaces

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

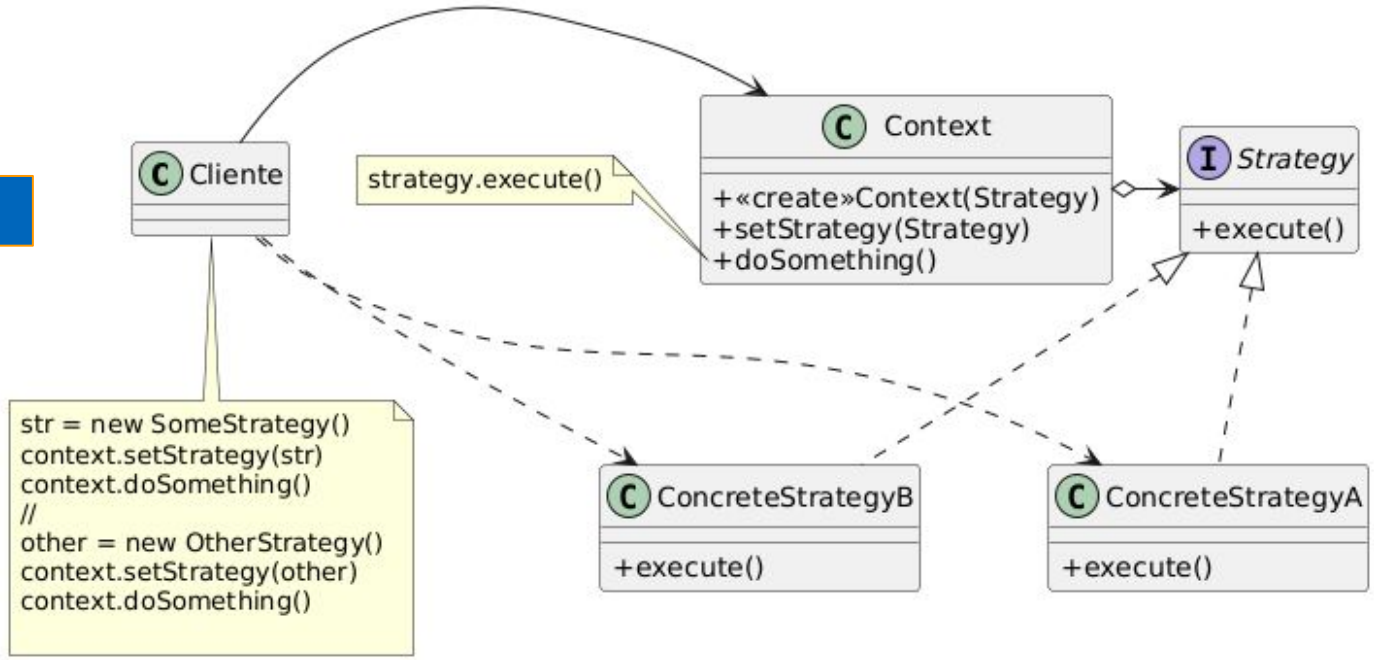
- Definir una familia de algoritmos, encapsular cada uno en un objeto y hacerlos intercambiables.
- Son los <<clientes>> del contexto los que generalmente crean las estrategias.



Patrón Strategy

Agregamos la clase <<cliente>>

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones



Patrón Strategy

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

Puntos a favor

- Mejor solución que subclasificar el contexto, cuando se necesita cambiar dinámicamente.
- Desacopla al contexto de los detalles de implementación de las estrategias.
- Se eliminan los condicionales.

Puntos en contra

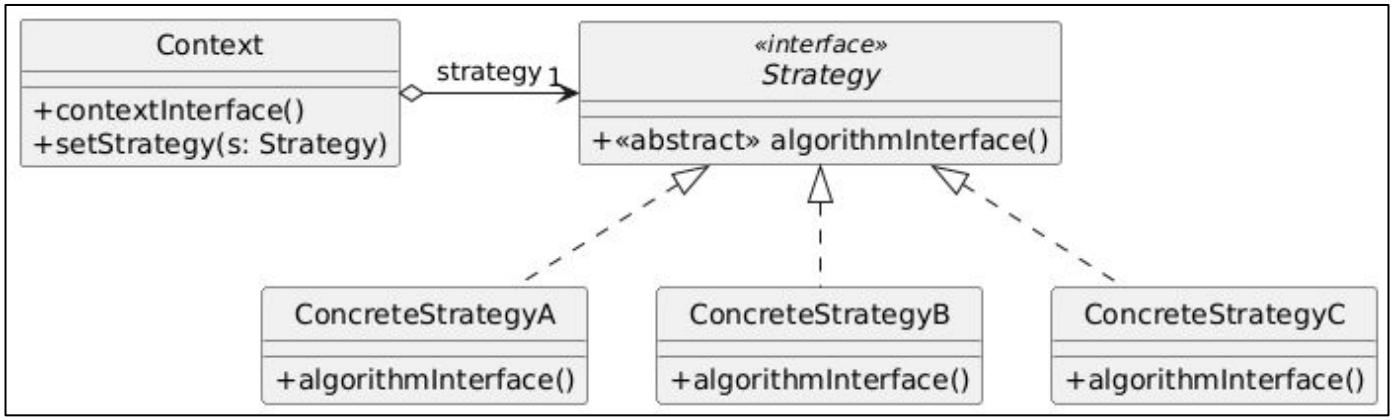
- La clase <<cliente>> debe conocer las diferentes estrategias para poder elegir.
- Overhead en la comunicación entre contexto y estrategias.

Patrón Strategy

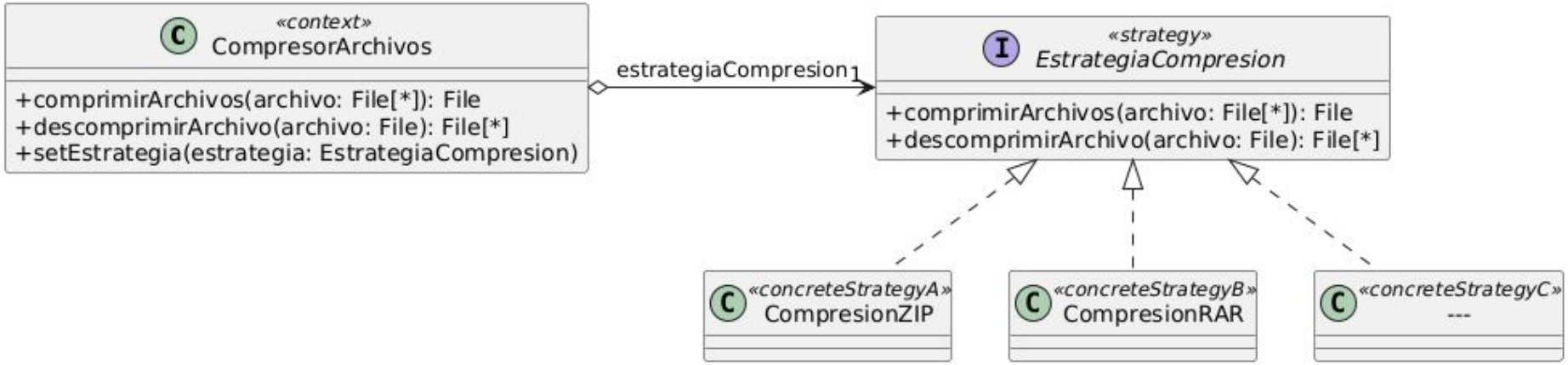
- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

- El contexto debe tener métodos en su protocolo que permitan cambiar la estrategia
- Parámetros entre el contexto y la estrategia
  - Hay que analizar qué datos se necesitan particularmente en cada caso

Ejemplo 1: App de compresión de archivos



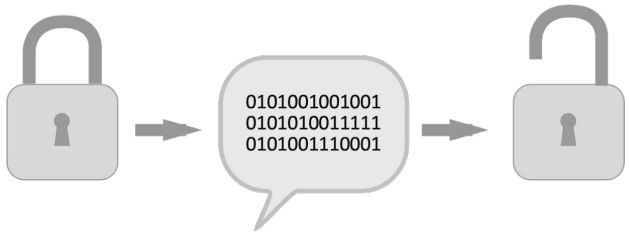
Aplicando Strategy



Relación entre Strategy y otros patrones

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

- Strategy y Adapter: ejemplo encriptación de mensajes



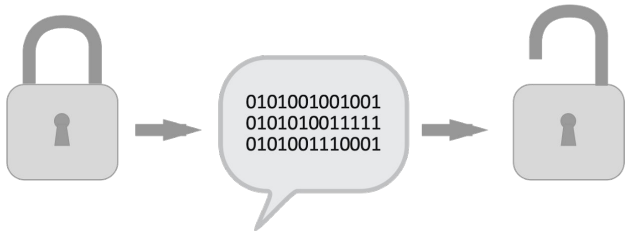
TEA
password
new: password encode: aString - setPassword: password

RC4
-
instance encrypt: aString using: key decript: encString using: key

Relación entre Strategy y otros patrones

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

- Strategy y Adapter: ejemplo encriptación de mensajes



TEA
password
new: password encode: aString - setPassword: password

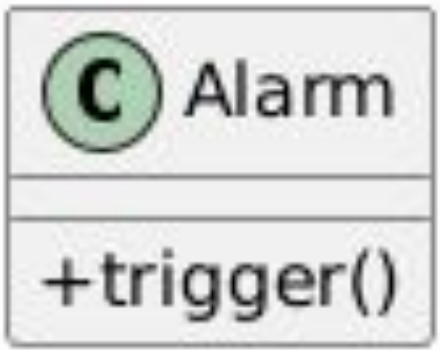
RC4
-
instance encrypt: aString using: key decript: encString using: key

- Strategy y Template Method: dónde puede aparecer un Template en el patrón Strategy?

Nuevo patrón de comportamiento

Ejemplo 1: Sistema de Alarmas

- Supongamos una clase Alarm con un comportamiento "trigger()" que reacciona a mensajes enviados por sensores

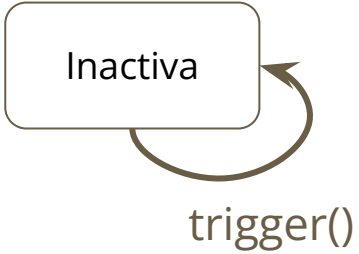


Ejemplo 1: Sistema de Alarmas

- La alarma puede estar en diferentes estados y en función de eso reacciona:

Si esta inactive	No toma en cuenta ningún aviso de los sensores
Si está active	tiene que reaccionar de acuerdo a su comportamiento como Alarma
Si está sleeping	se activa
Otras combinaciones	

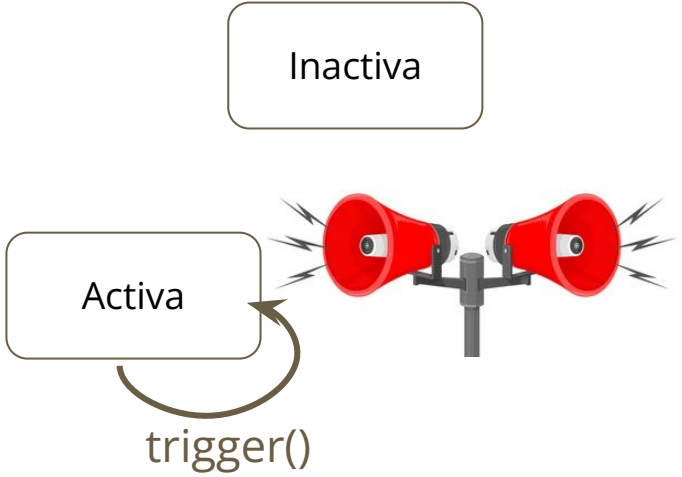
Ejemplo 1: Sistema de Alarmas



- La alarma puede estar en diferentes estados y en función de eso reacciona:

Si esta inactive	No toma en cuenta ningún aviso de los sensores
Si está active	tiene que reaccionar de acuerdo a su comportamiento como Alarma
Si está sleeping	se activa
Otras combinaciones	

Ejemplo 1: Sistema de Alarmas

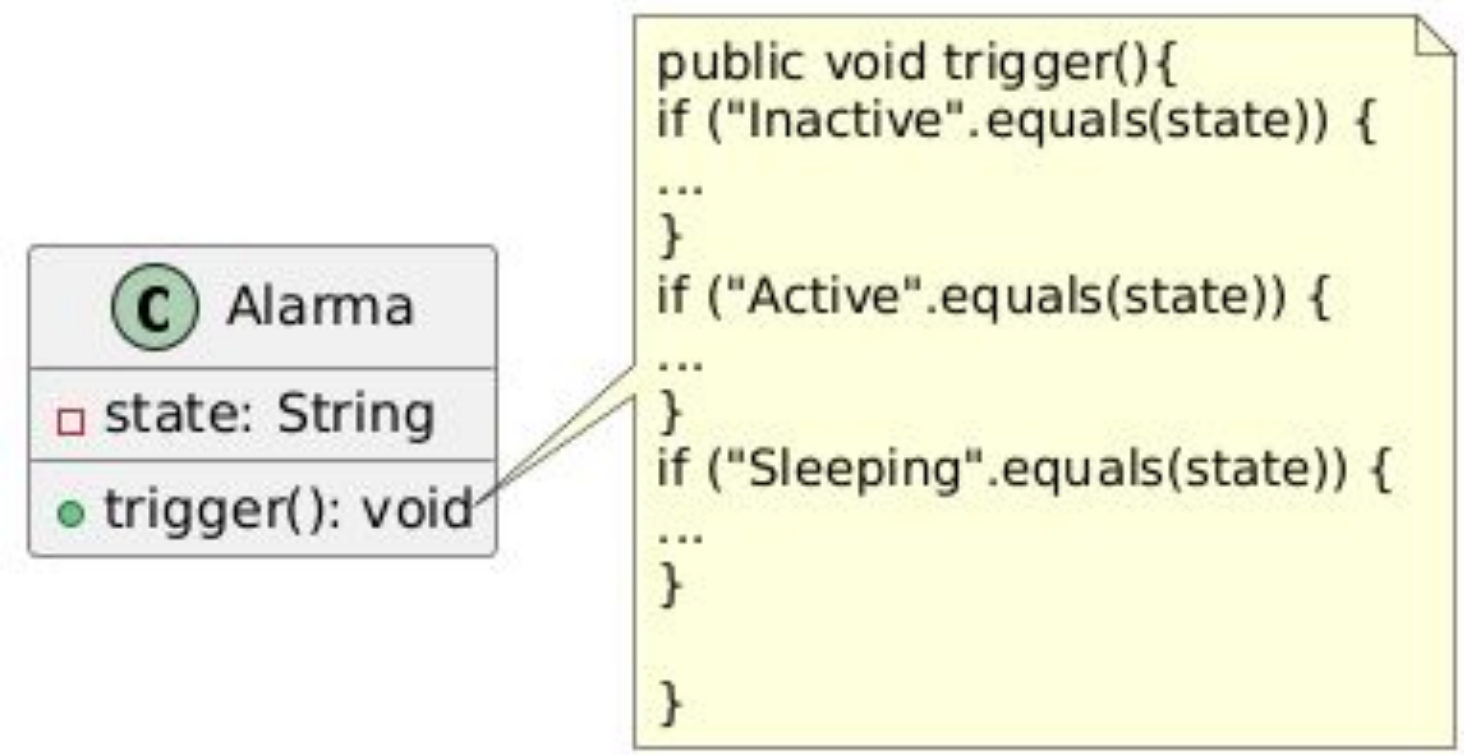


- La alarma puede estar en diferentes estados y en función de eso reacciona:

Si esta inactive	No toma en cuenta ningún aviso de los sensores
Si está active	tiene que reaccionar de acuerdo a su comportamiento como Alarma
Si está sleeping	se activa
Otras combinaciones	

Ejemplo 1: Sistema de Alarmas
¿Cómo resolvemos el problema?

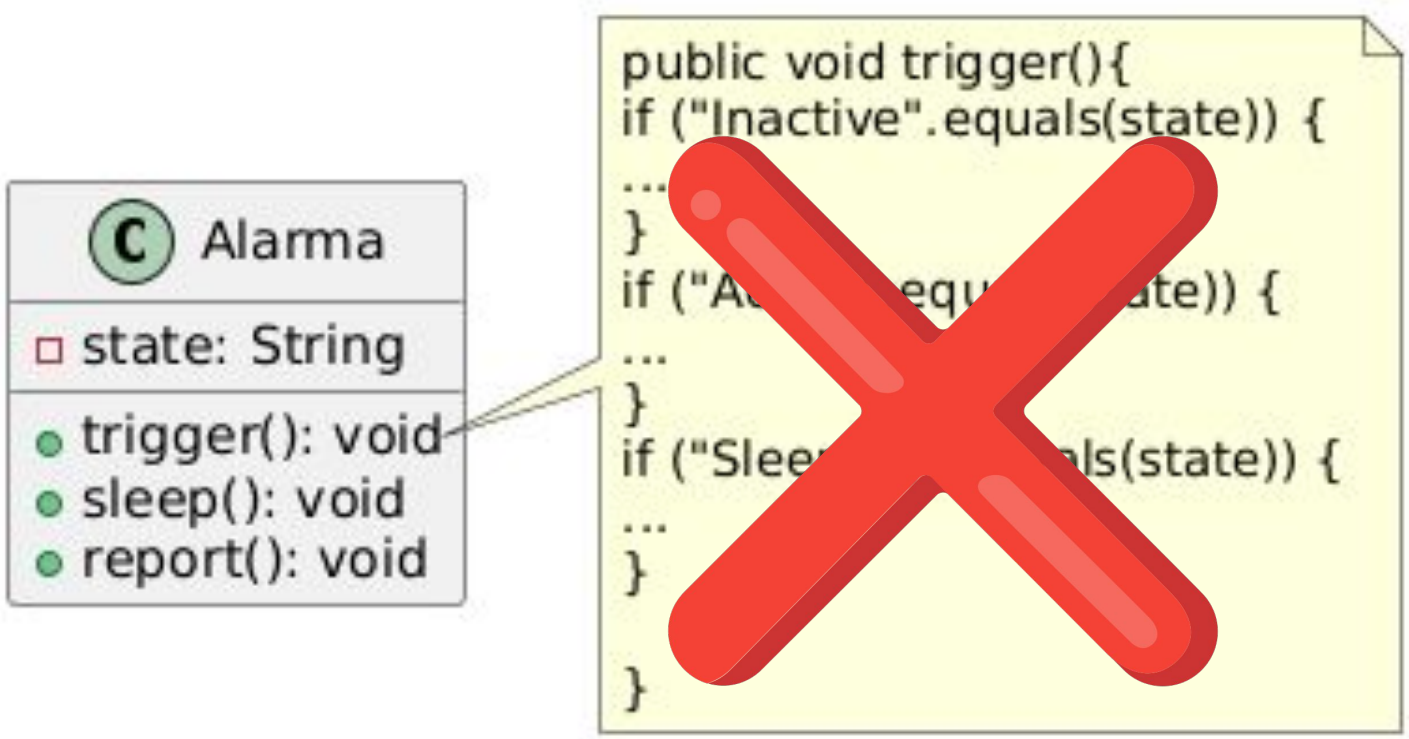
- Solución “ingenua”:



Problemas con esta solución?

Ejemplo 1: Sistema de Alarmas
¿Cómo resolvemos el problema?

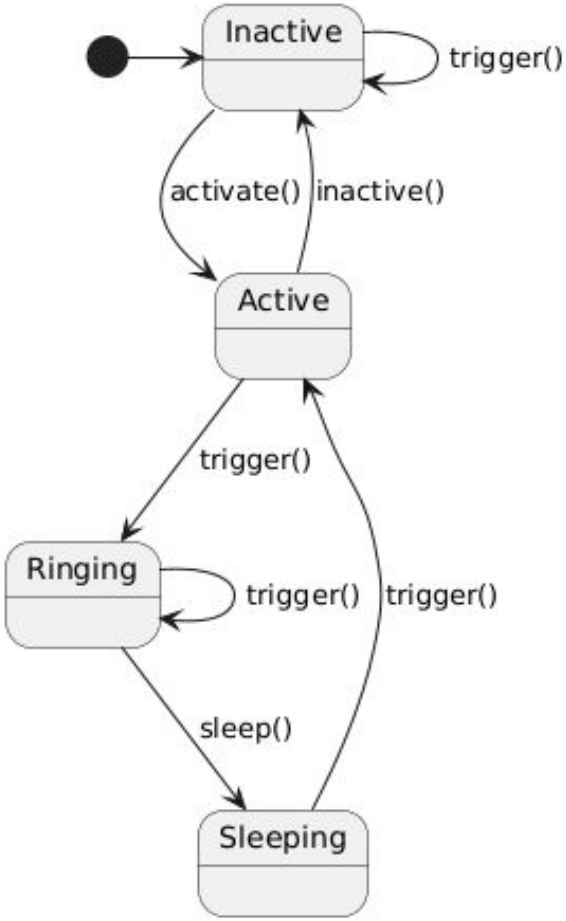
- Solución “ingenua”:

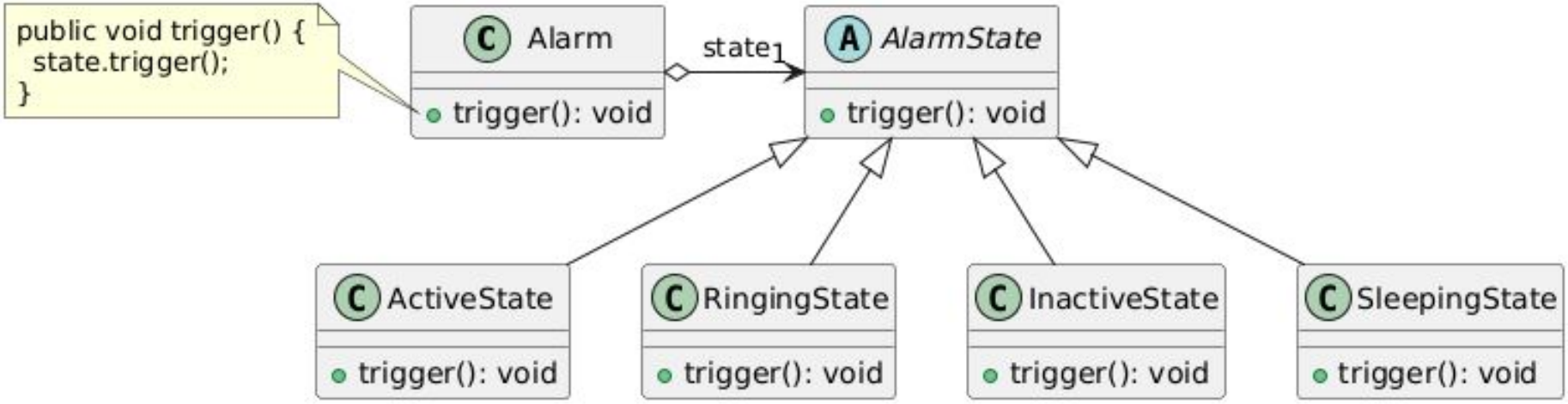


Problemas con esta solución?

Ejemplo 1: Sistema de Alarmas
Una solución mejor

- "Objetificar" el estado





Ejemplo 2: Calculadora

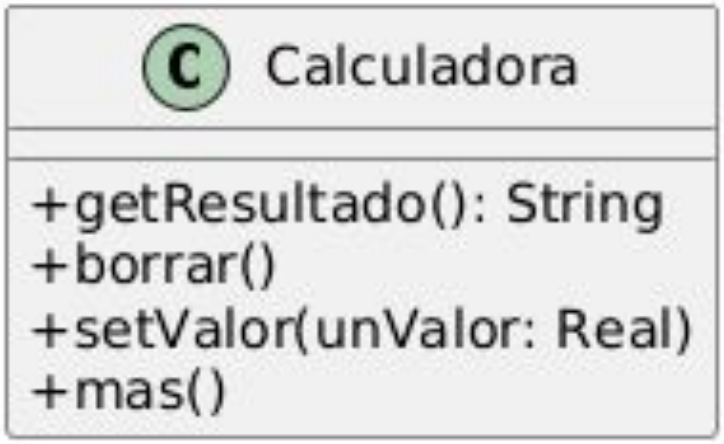
- Se desea diseñar e implementar una calculadora que realice operaciones matemáticas básicas, similar a las calculadoras tradicionales. Esta calculadora permitirá al usuario ingresar valores numéricos y realizar operaciones de suma, resta, multiplicación y división. Además, contará con la posibilidad de borrar la entrada actual y reiniciar los cálculos.
- La calculadora debe responder a los siguientes mensajes

Ejemplo 2: Calculadora

```
 /** Devuelve el resultado actual de la operación realizada.
   * Si no se ha realizado ninguna operación, devuelve el valor acumulad
   * Si la calculadora se encuentra en error, devuelve "error"
  */
 public String getResultado() {...}

 /** Pone en cero el valor acumulado y reinicia la calculadora*/
 public void borrar() {...}

 /** Asigna un valor para operar.
  * si hay una operación en curso, el valor será utilizado en la operaci
 public void setValor(double unValor) {...}

  /**Indica que la calculadora debe esperar un nuevo valor.
 * Si a continuación se le envía el mensaje setValor(), la calculadora s
recibido como parámetro, al valor actual y guardará el resultado */
 public void mas() {...}
```

Ejemplo 2: Calculadora

ValorAcumulado = 0



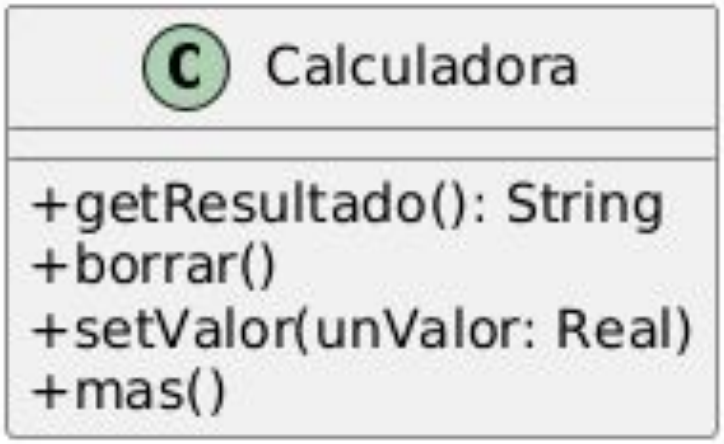
Inicio

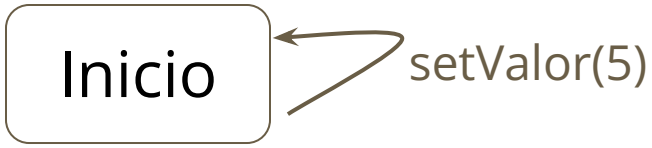
La calculadora muestra "0" y espera el primer número.

```
Calculadora calc = new Calculadora();
calc.setValor(5); // Establece el valor inicial
calc.mas(); // Prepara para sumar
calc.setValor(3); // Suma 3 al valor acumulado
System.out.println(calc.resultado()); // Imprimirá "8.0"
calculadora.por();
calculadora.setValor(2);
assertEquals(calculadora.resultado(), "16.0");
```

Ejemplo 2: Calculadora

ValorAcumulado = 5

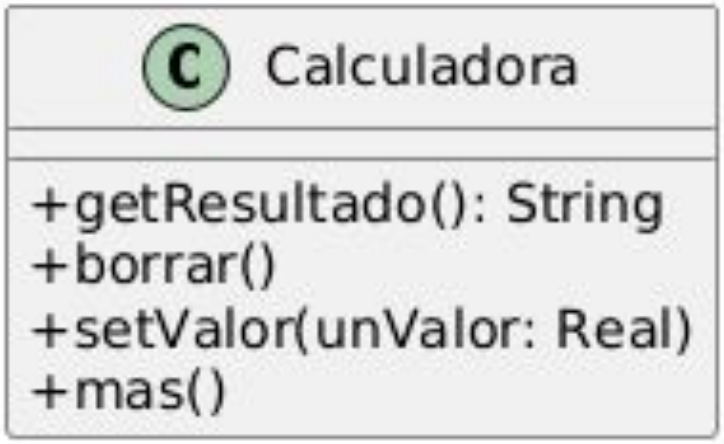


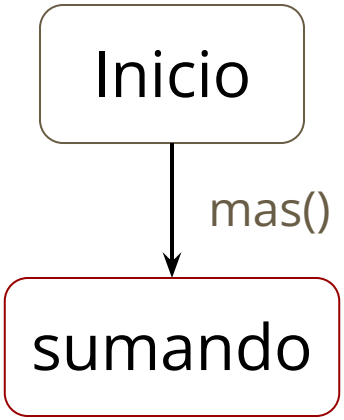



```
Calculadora calc = new Calculadora();
calc.setValor(5); // Establece el valor inicial
calc.mas(); // Prepara para sumar
calc.setValor(3); // Suma 3 al valor acumulado
System.out.println(calc.resultado()); // Imprimirá "8.0"
calculadora.por();
calculadora.setValor(2);
assertEquals(calculadora.resultado(), "16.0");
```

Ejemplo 2: Calculadora

ValorAcumulado = 5

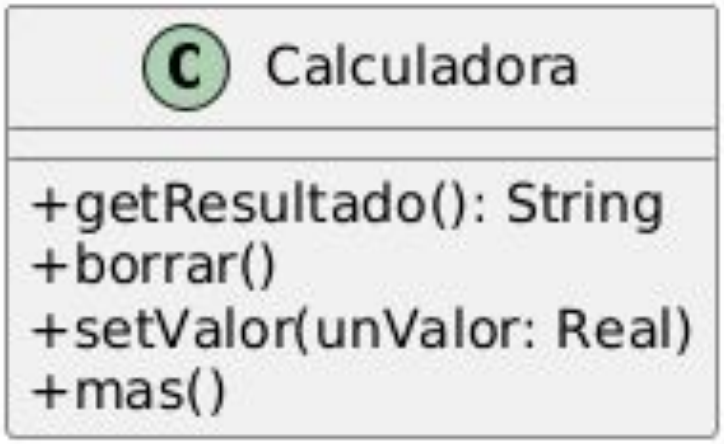


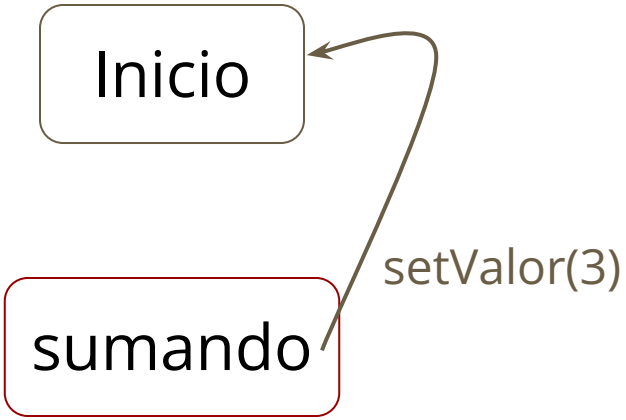



```
Calculadora calc = new Calculadora();
calc.setValor(5); // Establece el valor inicial
calc.mas(); // Prepara para sumar
calc.setValor(3); // Suma 3 al valor acumulado
System.out.println(calc.resultado()); // Imprimirá "8.0"
calculadora.por();
calculadora.setValor(2);
assertEquals(calculadora.resultado(), "16.0");
```

Ejemplo 2: Calculadora

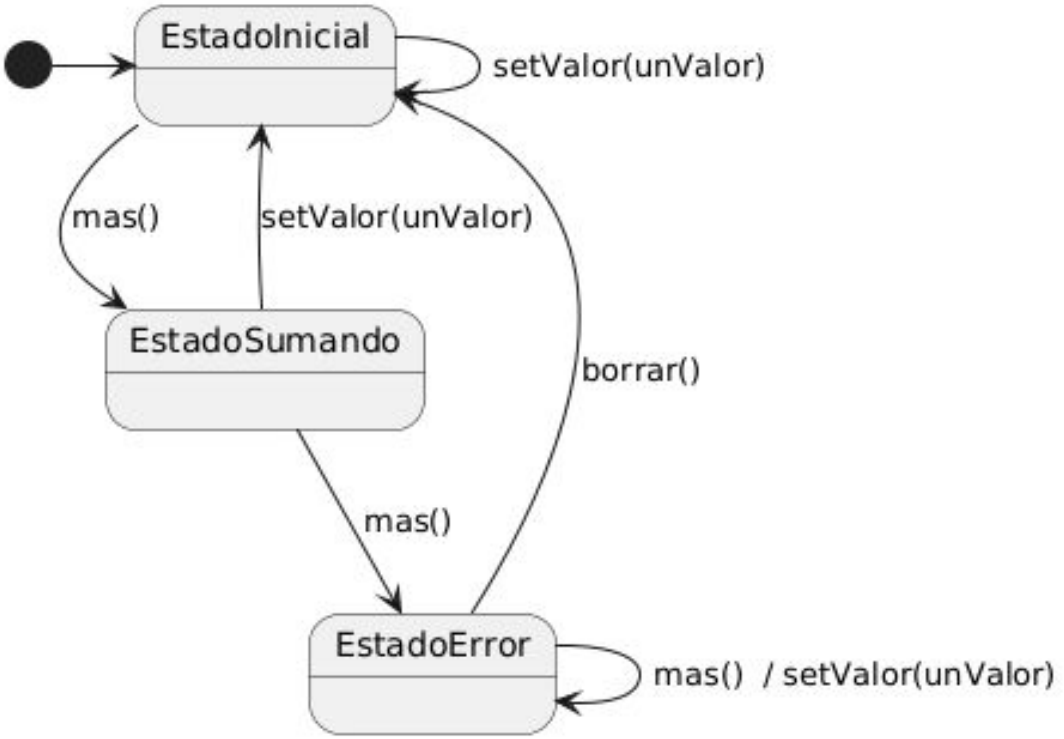
ValorAcumulado = 8

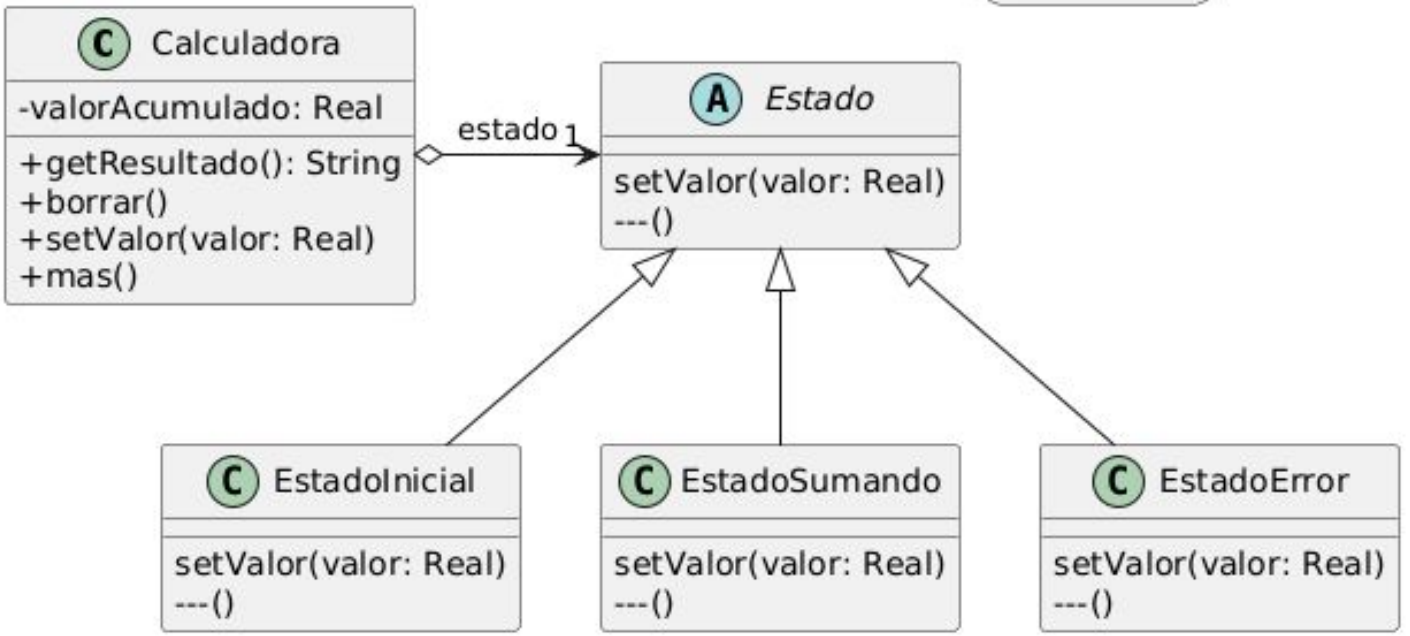





```
Calculadora calc = new Calculadora();
calc.setValor(5); // Establece el valor inicial
calc.mas(); // Prepara para sumar
calc.setValor(3); // Suma 3 al valor acumulado
System.out.println(calc.resultado()); // Imprimirá "8.0"
calculadora.por();
calculadora.setValor(2);
assertEquals(calculadora.resultado(), "16.0");
```

Ejemplo 2: Calculadora





Patrón State

- **Intent**
- **Applicability**
- **Estructura**
- **Consecuencias**
- **Implementación**
- **Relación con otros patrones**

- Modificar el comportamiento de un objeto cuando su estado interno se modifica.
- Externamente parecería que la clase del objeto ha cambiado.

Patrón State

- **Intent**
- **Applicability**
- **Estructura**
- **Consecuencias**
- **Implementación**
- **Relación con otros patrones**

Uso el patrón State cuando...

- El comportamiento de un objeto depende del estado en el que se encuentre.
- Los métodos tienen sentencias condicionales complejas que dependen del estado. Este estado se representa usualmente por constantes enumerativas y en muchas operaciones aparece el mismo condicional.
  El patrón State reemplaza el condicional por clases (es un uso inteligente del polimorfismo)

Patrón State

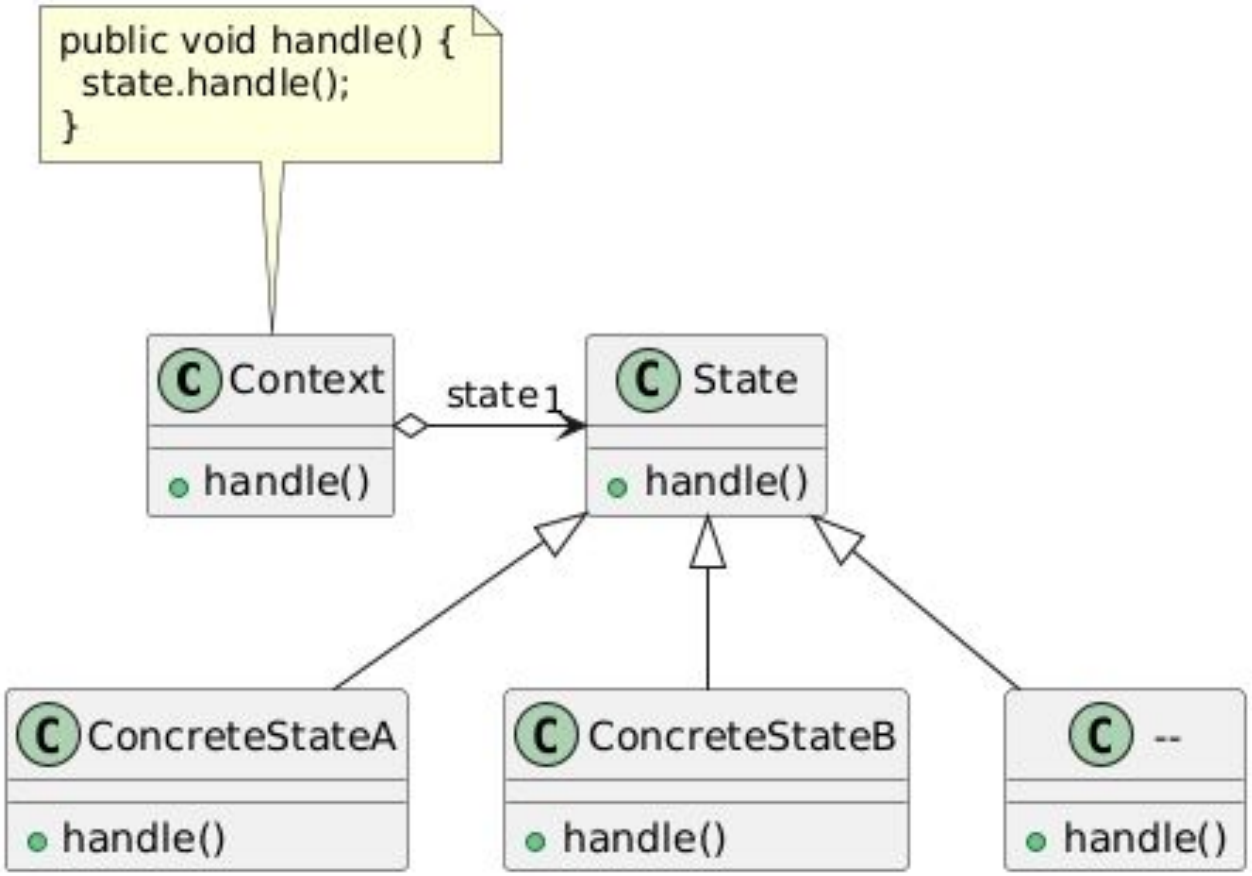
- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

Detalles:

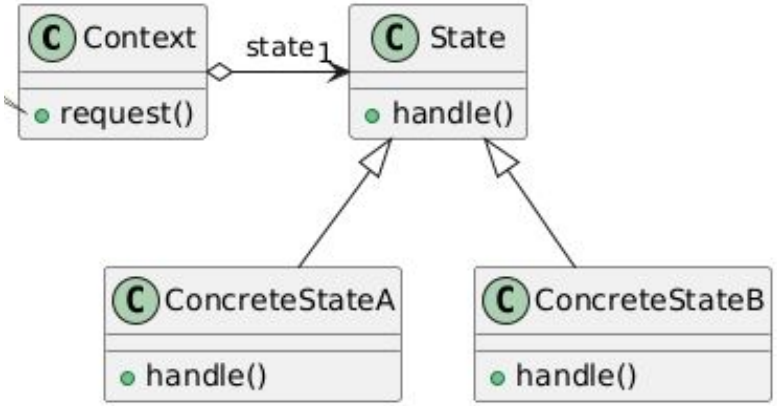
- Desacoplar el estado interno del objeto en una jerarquía de clases.
- Cada clase de la jerarquía representa un estado concreto en el que puede estar el objeto.
- Todos los mensajes del objeto que dependan de su estado interno son delegados a las clases concretas de la jerarquía (polimorfismo).

Patrón State

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones



Patrón State



- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

- **Participantes**
  - **Context (Alarm)**
    - Define la interfaz que conocen los clientes.
    - Mantiene una instancia de alguna clase de ConcreteState que define el estado corriente
  - **State (AlarmState)**
    - Define la interfaz para encapsular el comportamiento de los estados de Context
  - **ConcreteState subclases (Active, Inactive, Ringing, Sleeping)**
    - Cada subclase implementa el comportamiento respecto al estado específico.

Patrón State

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

- **Puntos a favor**
  - Localiza el comportamiento relacionado con cada estado.
  - Las transiciones entre estados son explícitas.
  - En el caso que los estados no tengan variables de instancia pueden ser compartidos.

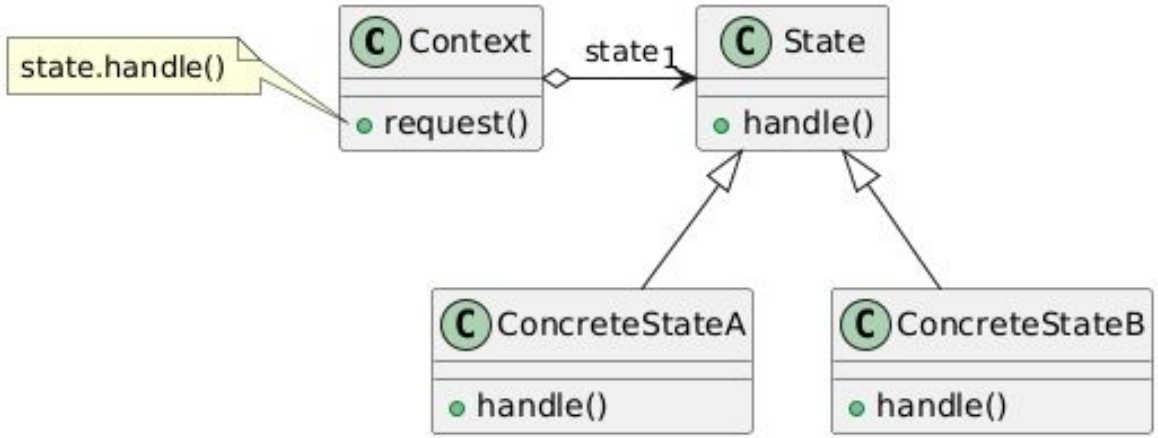
- **Puntos en contra**
  - En general hay bastante acoplamiento entre las subclases de State porque la transición de estados se hace entre ellas, por lo que deben conocerse entre sí

Patrón State

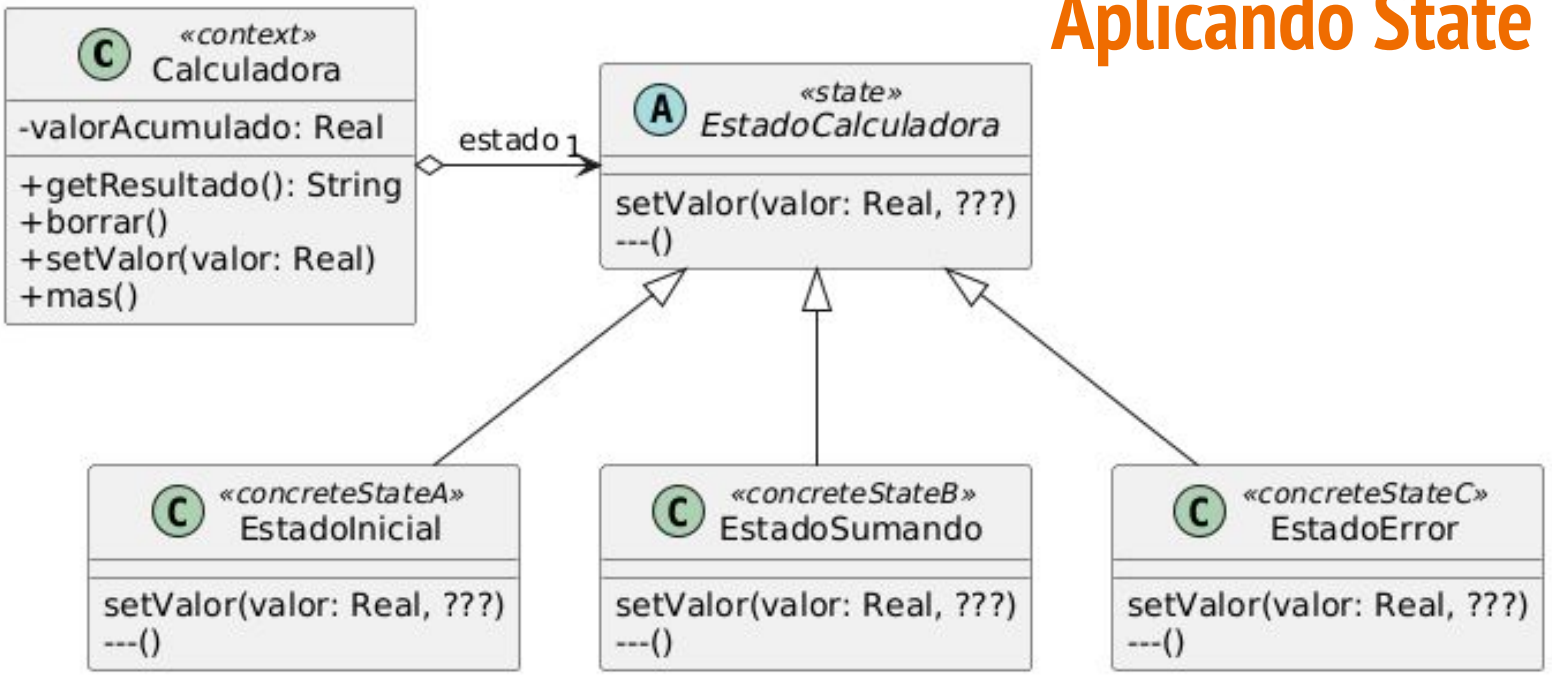
Intent
Applicability
Estructura
Consecuencias
Implementación
Relación con otros patrones

- **Temas Interesantes**
  - Los estados son internos al contexto
  - Ejecución de los **comportamientos de la alarma.** Dónde están ubicados?
  - Cómo cambiamos de estado?
  - Muchos objetos Alarma, comparten la jerarquía de estados?

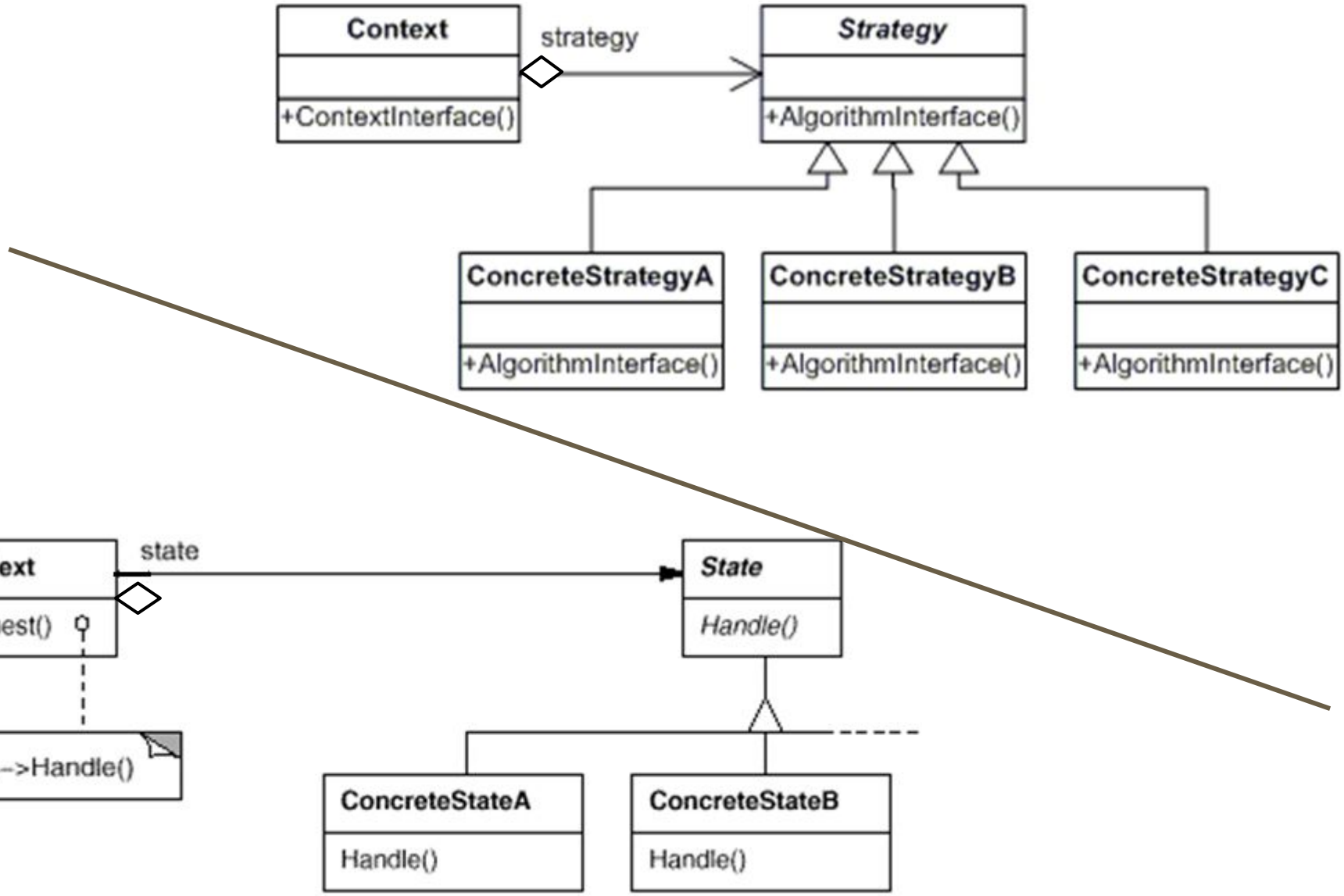
Ejemplo 2: Calculadora



Aplicando State

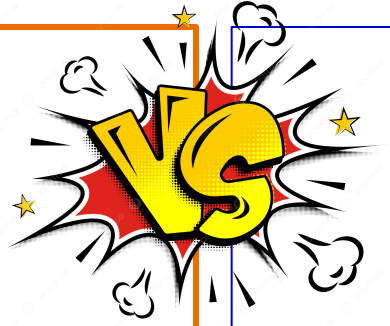


¿State o Strategy?



Los diagramas se ven muy parecidos, pero

State



Strategy

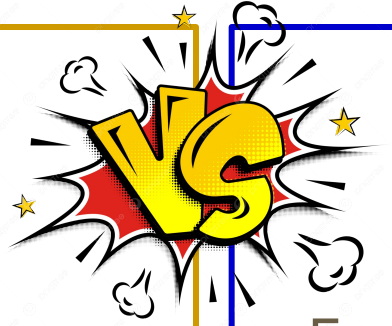
El comportamiento de un objeto depende del estado en el que se encuentre

El patrón State es útil para una clase que debe realizar transiciones entre estados fácilmente.

Necesito uno de diferentes **algoritmos opcionales para realizar una misma tarea**

El patrón Strategy es útil para permitir que una clase delegue la ejecución de un algoritmo a una instancia de una familia de estrategias

State



Strategy

- En State, los diferentes estados:

  - son **internos al contexto**,

  - no los eligen las clases clientes

  - la transición se realiza entre los estados mismos

- En Strategy, l**as diferentes estrategias:**

  - son conocidas desde afuera del contexto, por las clases clientes del contexto.

- el Contexto del Strategy **debe contener un mensaje público para cambiar el ConcreteStrategy**.

State vs. Strategy

Resumen

- El **estado es privado del objeto**, ningún otro objeto sabe de él. vs.

≠ El Strategy suele setearse por el cliente, que debe conocer las posibles estrategias concretas.

- Cada State **puede definir muchos mensajes.** vs.

≠ Un Strategy suele tener un único mensaje público.

- Los states concretos **se conocen** entre sí. Saben a cual estado se debe pasar en respuesta a algún mensaje.

≠ Los strategies concretos no.

Preguntas ¿?