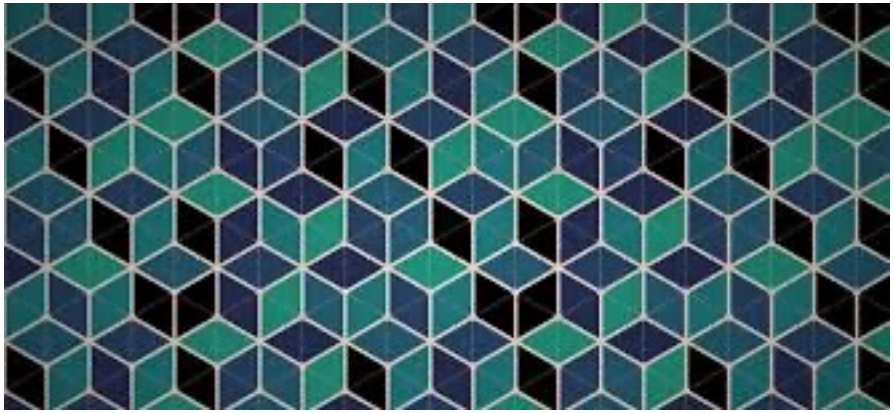


Patrones de diseño

Composite, Strategy, State

Dra. Alejandra Garrido
garrido@lifia.info.unlp.edu.ar
https://lifia.info.unlp.edu.ar/dra-alejandra-garrido/

Qué recuerdan sobre patrones?

- Qué es un patrón de diseño?
- Para qué sirve?
- Qué es importante recordar de un patrón?

¿Qué cosa es importante estudiar y recordar?

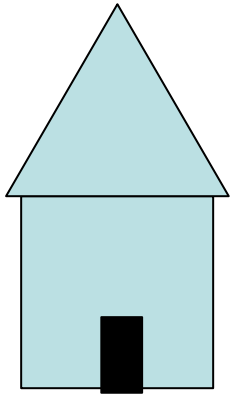
- Propósito
- Estructura:
  - clases que componen el patrón (roles),
  - cómo se relacionan (jerarquías, clases abstractas/interfaces, métodos abstractos – protocolo de interfaces, conocimiento/composición)
- Variantes de implementación
- Consecuencias positivas y negativas
- Relación con otros patrones

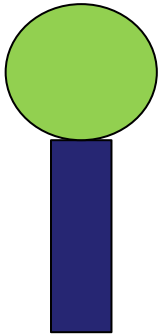
Nuevo patrón estructural

(se ocupa de la composición de clases y objetos)

Problema

- Supongamos que queremos manejar figuras compuestas y tratarlas como figuras simples (moverlas, rotarlas, etc).





Casa= Cuerpo + Techo
Cuerpo= Puerta + Pared

Arbol= Tronco + Copa

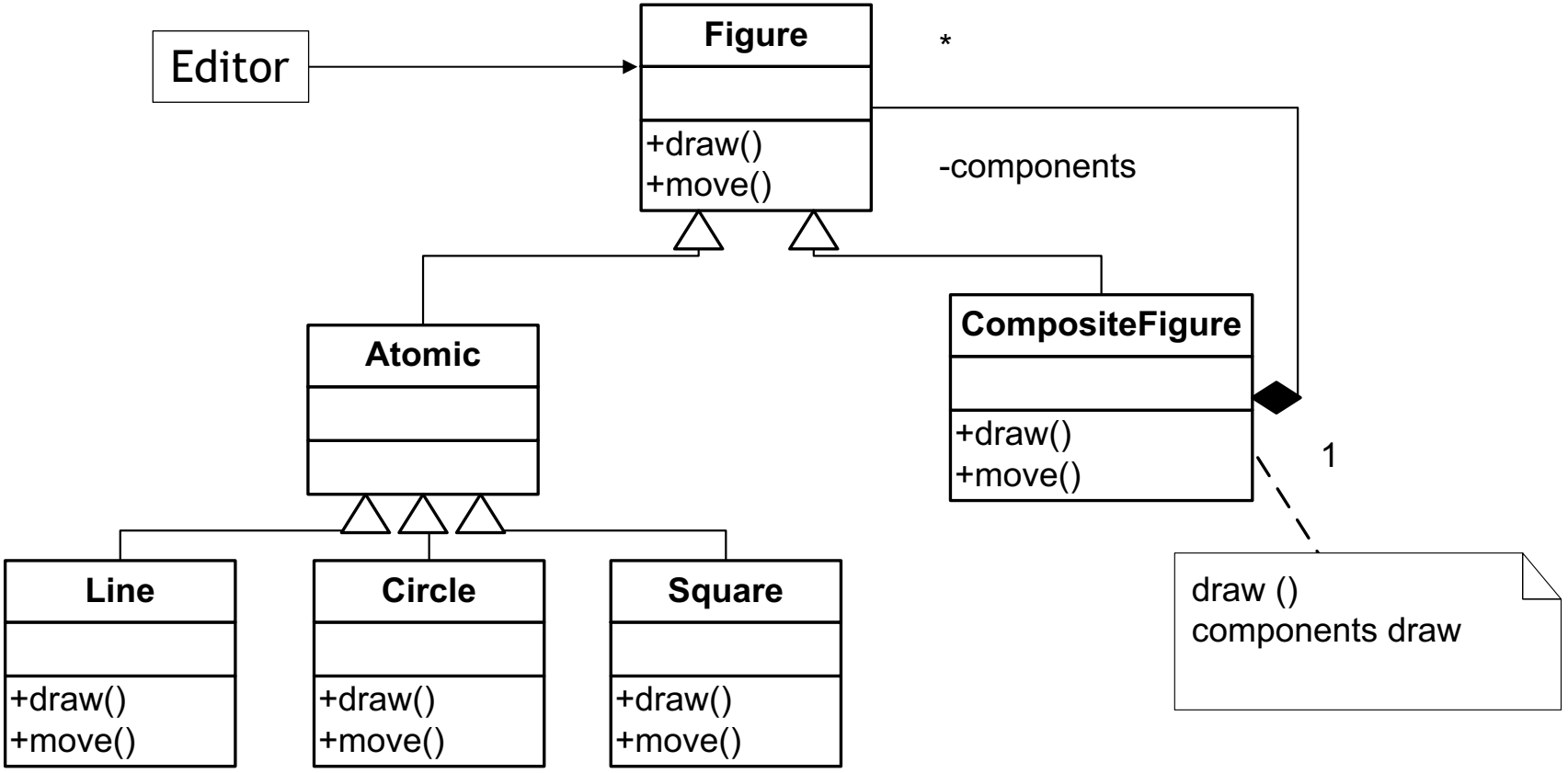
Propiedad= Casa + Arbol*

Soluciones?

- Podemos tener una lista de figuras y marcarlas como partes de otras….
- Cómo sería tal lista cuando hay composiciones muy “profundas”?
- Por ejemplo “Barrio”
- ¿Se les ocurre una estructura más adecuada que una lista secuencial?

Una solución más modular

- Tratarlas uniformemente



Como funciona?

- El editor solo maneja figuras
- Mantenemos la interfaz polimórfica
- Problemas? Como creamos figuras compuestas?

Una instancia del patrón Composite

Pattern Composite

- **Propósito**

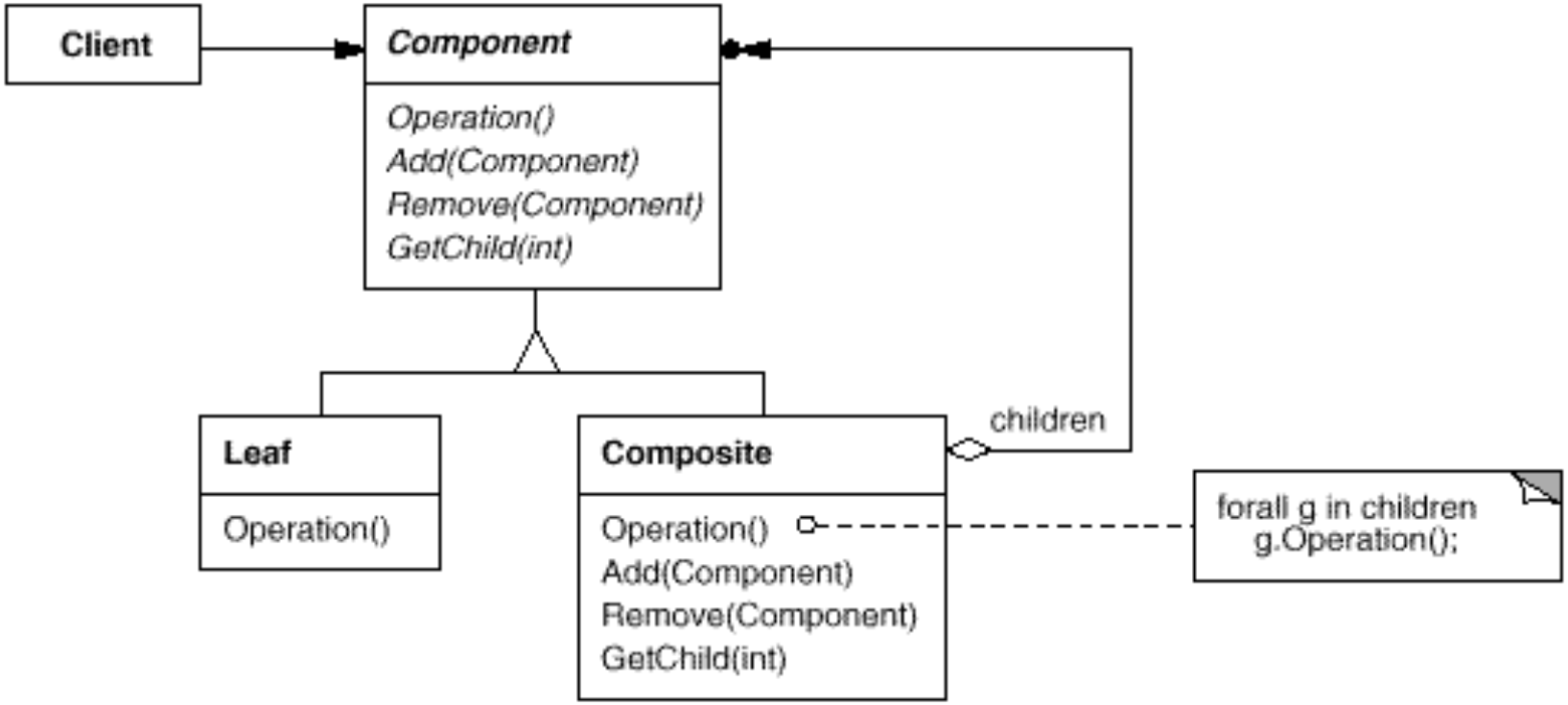
  Componer objetos en estructuras de arbol para representar jerarquias parte-todo. El Composite permite que los clientes traten a los objetos atómicos y a sus composiciones uniformemente

- **Aplicabilidad**

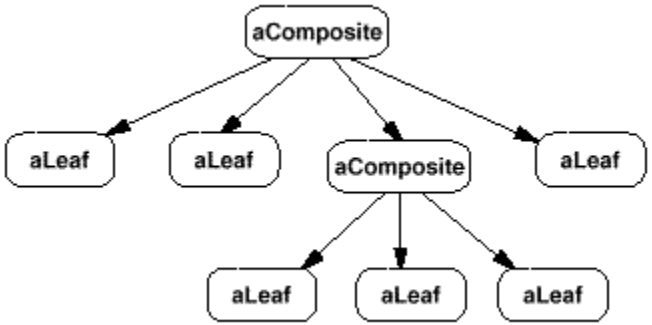
  Use el patrón Composite cuando

  - quiere representar jerarquías parte-todo de objetos.
  - quiere que los objetos “clientes” puedan ignorar las diferencias entre composiciones y objetos individuales. Los clientes trataran a los objetos atómicos y compuestos uniformemente.

Pattern Composite. Estructura



Una instancia típica se ve así



Composite

- **Participantes**
  - **Component** (Figure)
    - Declara la interfaz para los objetos de la composicion.
    - Implementa comportamientos default para la interfaz comun a todas las clases
    - Declara la interfaz para definir y acceder “hijos”.
    - (opcional) define una interfaz para para acceder el “padre” de un componente en la estructura recursiva y la implementa si es apropiado.

Composite

- **Leaf** (Rectangle, Line, Text, etc.)
  - Representa arboles “hojas” in la composicion. Las hojas no tienen “hijos”.
  - Define el comportamiento de objetos primitivos en la composición.
- **Composite** (CompositeFigure)
  - Define el comportamiento para componentes con “hijos”.
  - Contiene las referencias a los “hijos”.
  - Implementa operaciones para manejar “hijos”.

- **Consequencias**

+ Define jerarquías de clases consistentes de objetos primitivos y compuestos. Los objetos primitivos pueden componerse en objetos complejos, los que a su vez pueden componerse **recursivamente**.

+ Simplifica los objetos cliente. Los clientes usualmente no saben (y no deberian preocuparse) acerca de si están manejando un compuesto o un simple.

+ Hace más fácil el agregado de nuevos tipos de componentes porque los clientes no tienen que cambiar cuando aparecen nuevas clases componentes.

- No permite restringir las estructuras de composición (cuando ciertos compuestos pueden armarse solo con cierto tipo de atómicos)

Cuestiones de implementación

- Referencias explícitas al padre (cuidado en mantener consistencia)
- Maximizar el protocolo de la clase/interfaz Component
- **Transparencia vs. seguridad**
- Orden de los hijos
- Caching
- Borrado de componentes
- Diferentes estructuras para guardar componentes

Enunciados de ejemplo

El Banco de Alimentos de la Ciudad de La Plata entrega cajas con productos alimenticios a partir de donaciones que recibe de distintas empresas. Suponga que el Banco necesita armar las cajas que entrega con combinaciones de productos alimenticios que cumplan ciertos requerimientos nutricionales.

Cumplir con los requerimientos nutricionales significa cumplir con todo lo siguiente:

- cierto valor energético mínimo
- cierto valor máximo de sodio
- cierto rango de valores mínimo y máximo de grasas saturadas
- cierto contenido vitamínico

Enunciados de ejemplo (2)

- A continuación se encuentra el código de 2 métodos de la clase BancoDeAlimentos. La misma representa a la organización que recibe donaciones y las distribuye a los comedores de la zona. Estos y otros métodos parecidos se ejecutan para calcular cuantas calorías, proteinas, hidratos, etc. se envían a un comedor en particular. El parámetro de la clase Producto puede ser en realidad un único producto, un pack o una caja que adentro puede contener variedad de cualquiera de las cosas anteriores, incluso otras cajas. Cabe aclarar que un pack es una bolsa que contiene un número determinado de productos del mismo tipo (ej. un pack de 30 latas de arvejas).

Atenti

- ¡¡Saber qué patrón aplicar es solo el 1er. paso!!
- Vamos a evaluar que sepan aplicarlo correctamente.

Nuevo patrón de comportamiento

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

Ejemplo 1: Validación de Strings

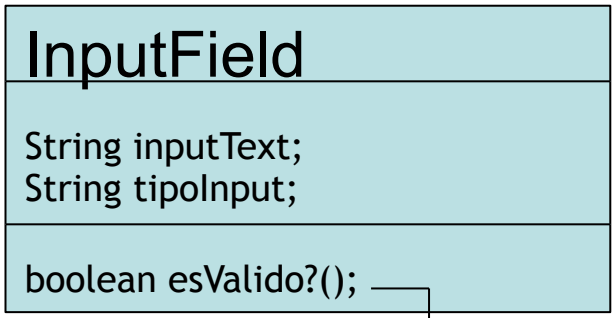
- Supongamos que estamos diseñando un formulario para ingresar datos

- Los campos de entrada de texto deben ser validados, pero la validación depende del dominio del texto ingresado
  - Teléfono
  - Fecha de nacimiento
  - DNI
  - E-mail
  - Dirección
  - ...

- No podemos preveer de antemano todas las posibles formas de validación.

Una posible solución

- En la clase donde esta el String, por ejemplo InputField, realizar el chequeo ….




```
boolean esValido?() {
    if (tipoInput == “Telefono”) {
            …}
    if (tipoInput == “Fecha”) {
            …}
    if (tipoInput == “Email”) {
            …}
    if ….
    ….
```

Problemas?

Ejemplo 2: Delivery de comida

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

- Pensemos en el pago. Si hubiera una sola forma de pago, por ejemplo con tarjeta de crédito, sería fácil.

```
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);
            return true; }
    else {
            System.out.println("No funcionó el pago con tarjeta");
            return false; }
}
```

Cómo agregamos nuevos métodos de pago?

- IF!

```
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);
            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?

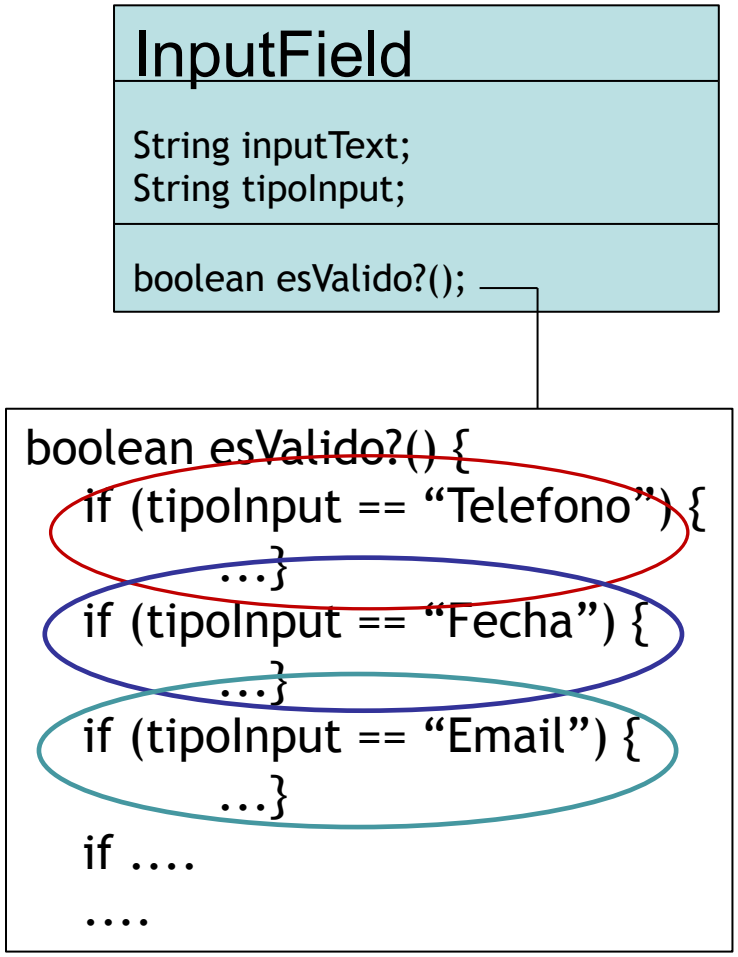
Cuál es una mejor solución?

- Cómo solucionamos este problema común de tener diferentes algoritmos opcionales para realizar una misma tarea (validar un string, pagar, etc.)?

- Encapsular cada algoritmo en un objeto y usarlos en forma intercambiada según se necesiten

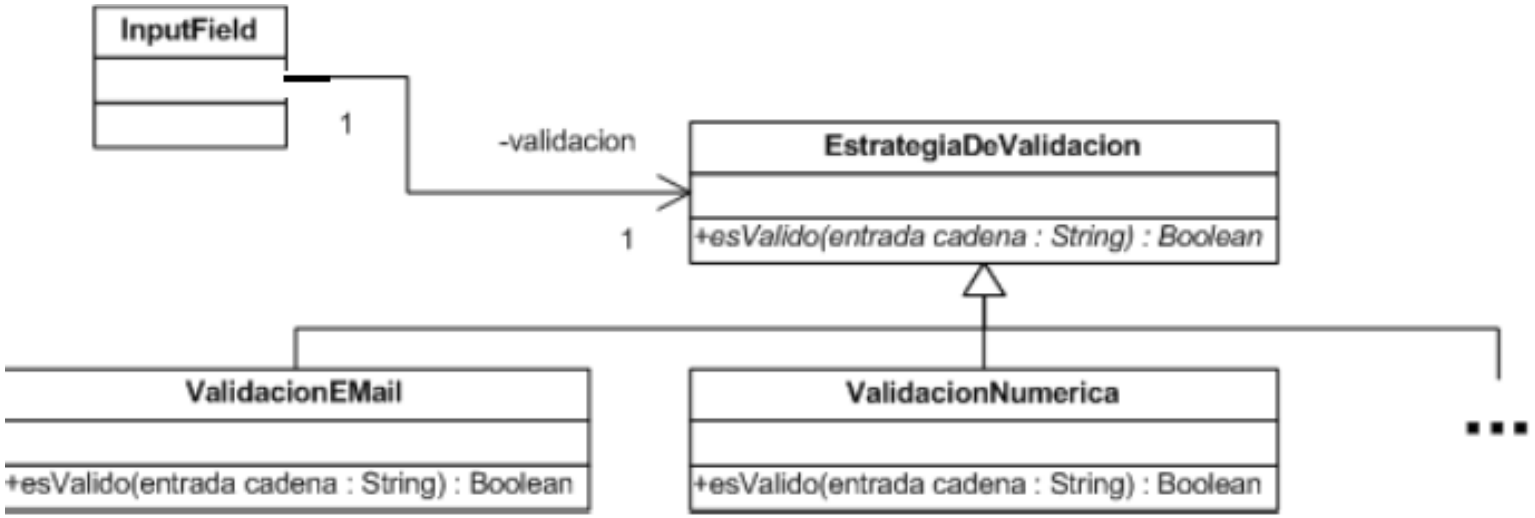
- Es lo que nos propone el patrón Strategy

Ejemplo 1: Validación de Strings

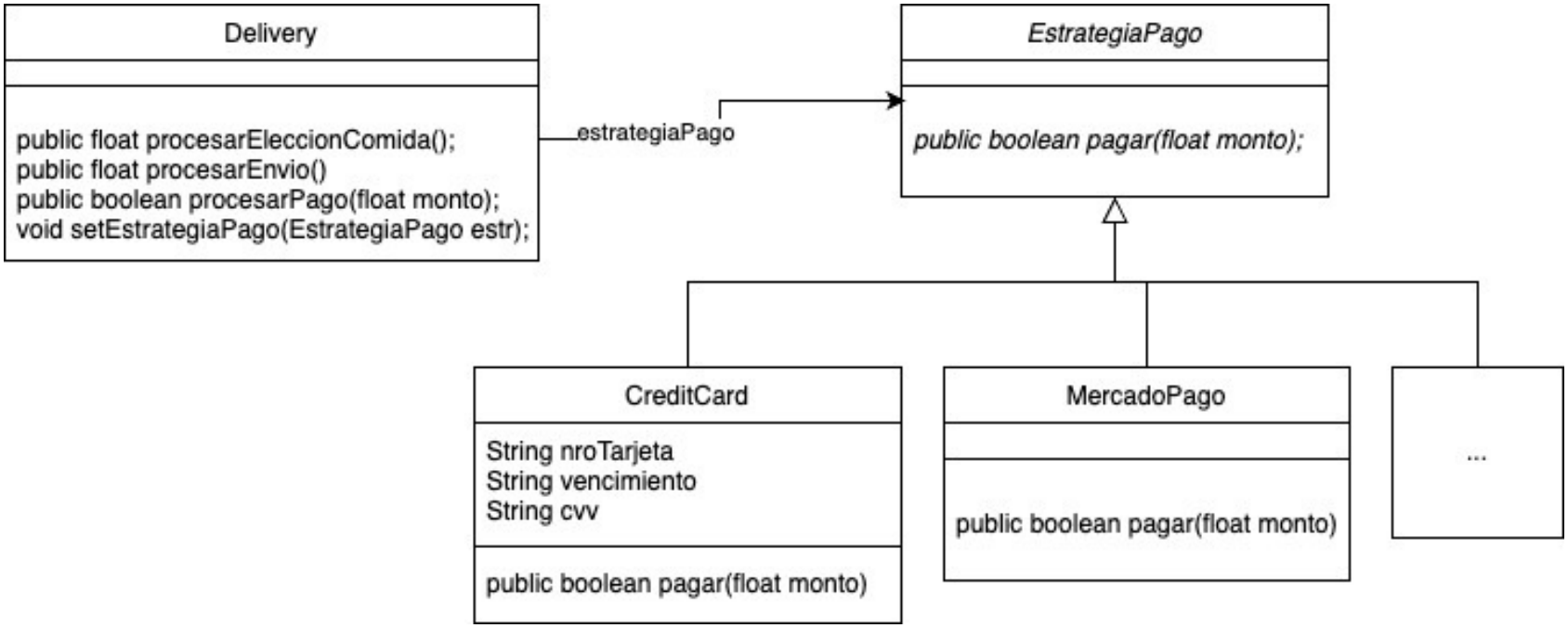


Aplicando Strategy

- Próposito de Strategy: definir una familia de algoritmos, encapsular cada uno y hacerlos intercambiables. El Strategy permite que el algoritmo varíe independientemente de los clientes que lo usan.



Ejemplo 2: Delivery de comida



Patrón Strategy

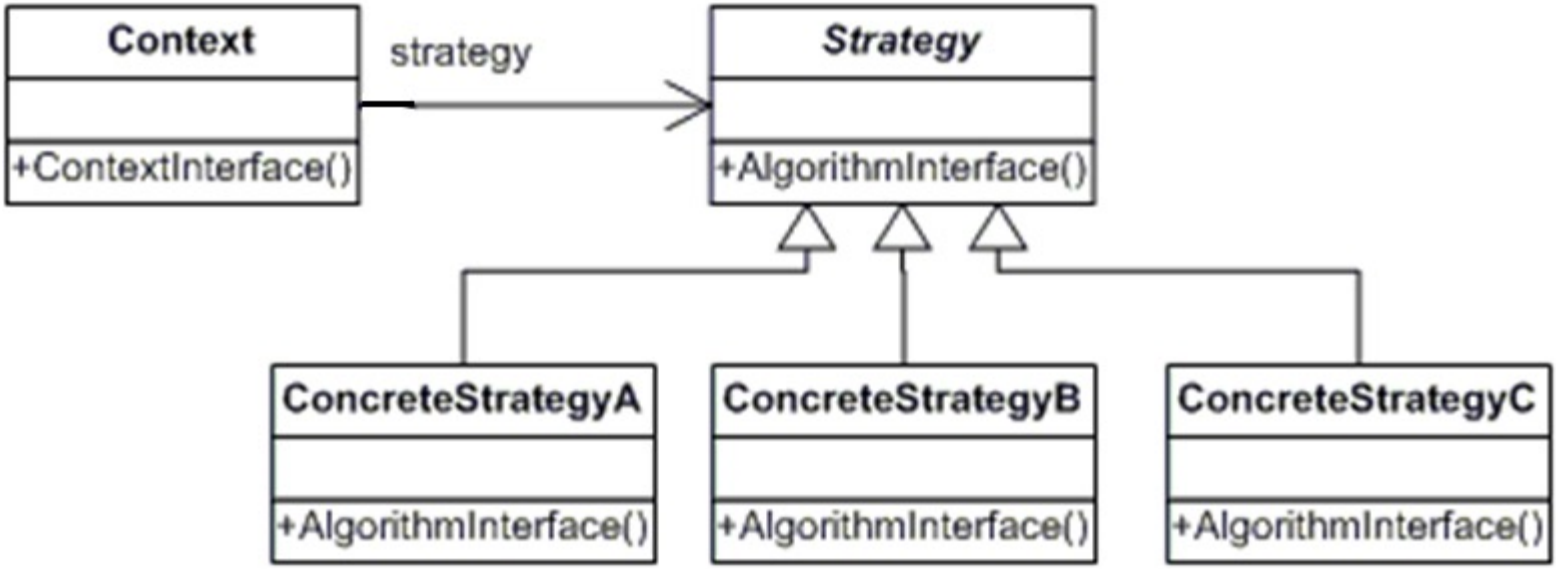
- **Intent:**
  - Desacoplar un algoritmo del objeto que lo utiliza.
  - Permitir cambiar el algoritmo que un objeto utiliza en forma dinámica.
  - Brindar flexibilidad para agregar nuevos algoritmos que lleven a cabo una función determinada.

Patrón Strategy

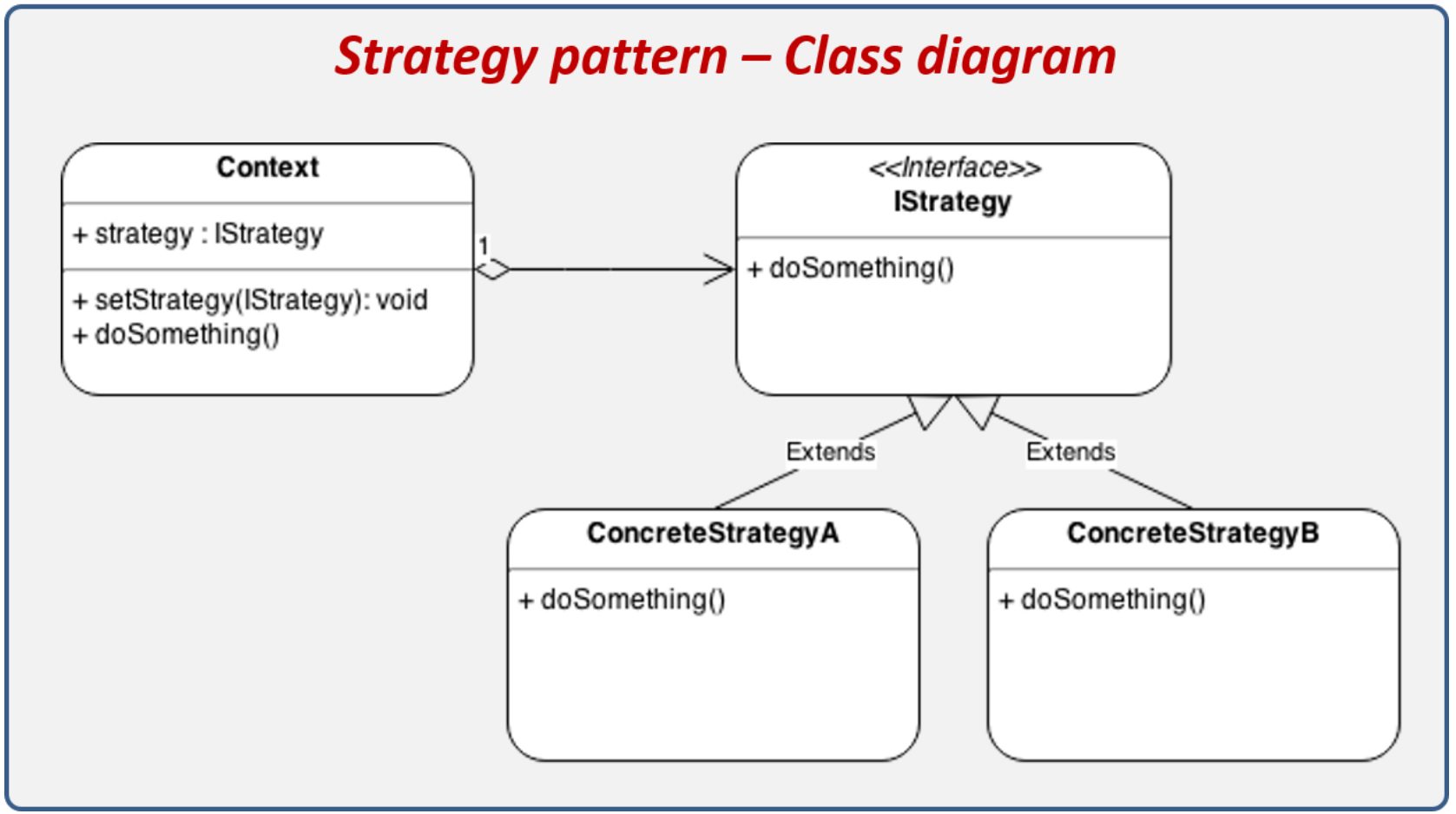
- **Applicability:**
  - Existen muchos algoritmos para llevar a cabo una tarea.
  - No es deseable codificarlos todos en una clase y seleccionar cual utilizar por medio de sentencias condicionales.
  - Cada algoritmo utiliza información propia. Colocar esto en los clientes lleva a tener clases complejas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

Patrón Strategy

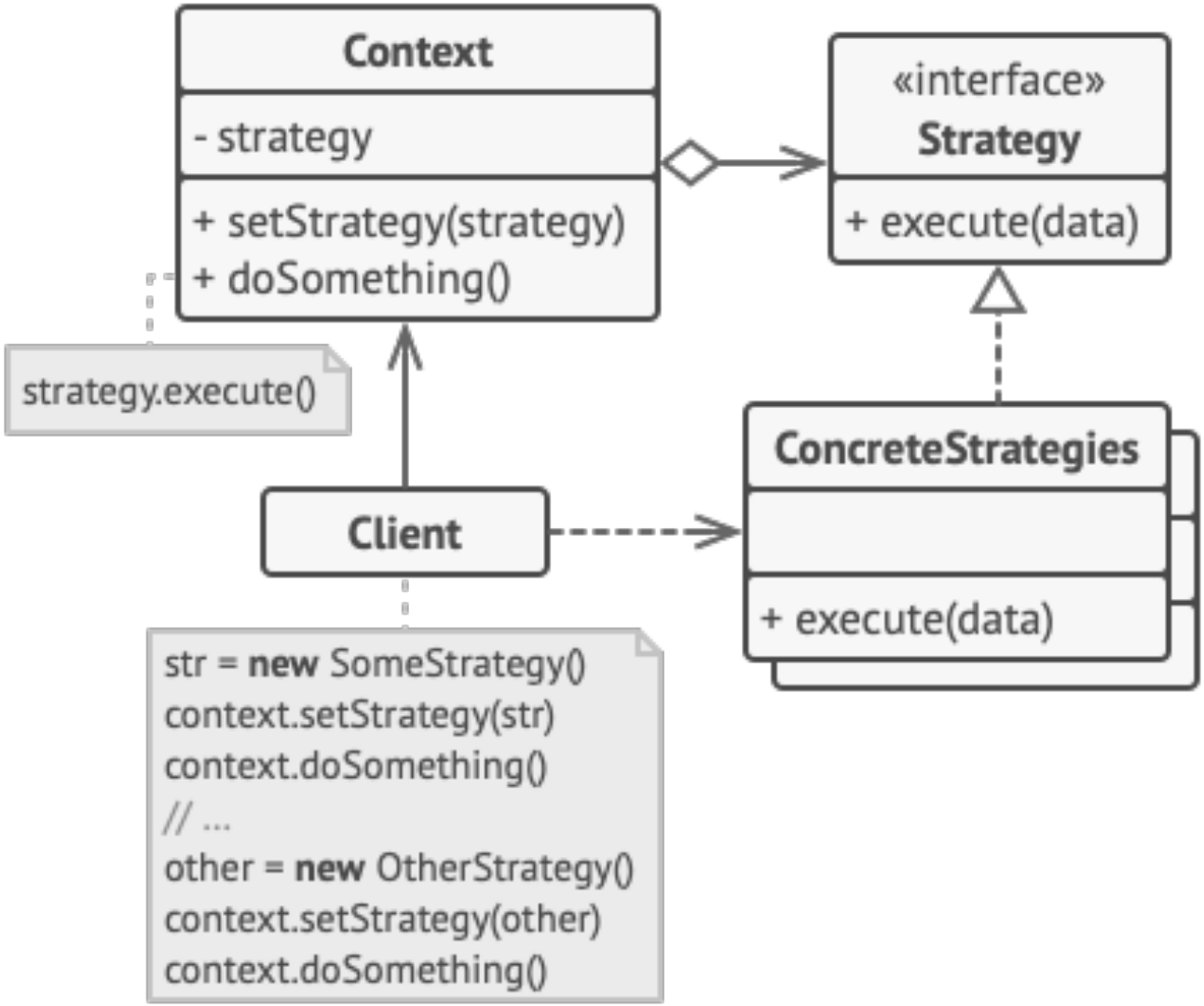
- **Solución-Estructura:**
  - Definir una familia de algoritmos, encapsular cada uno en un objeto y hacerlos intercambiables.
  - Son los clientes del contexto los que generalmente crean las estrategias.



Estructura del patrón Strategy con interfaces



Agregamos la clase Client



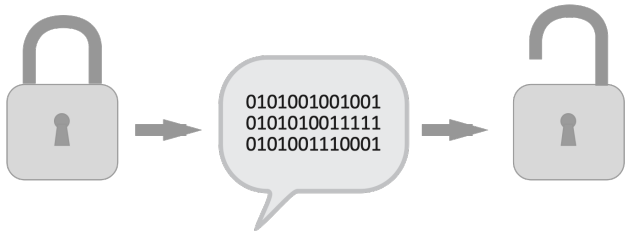
Patrón Strategy

- **Consecuencias:**
  - + Alternativa a subclasificar el contexto, para permitir que se pueda cambiar dinámicamente.
  - + Desacopla al contexto de los detalles de implementación de las estrategias.
  - + Se eliminan los condicionales.
  - – Overhead en la comunicación entre contexto y estrategias.
  - – Los clientes deben conocer las diferentes estrategias para poder elegir.

- **Implementación:**
  - El contexto debe tener en su protocolo métodos que permitan cambiar la estrategia
  - Parámetros entre el contexto y la estrategia

Relación entre Strategy y otros patrones

- Strategy y Adapter: ejemplo encriptación de mensajes



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

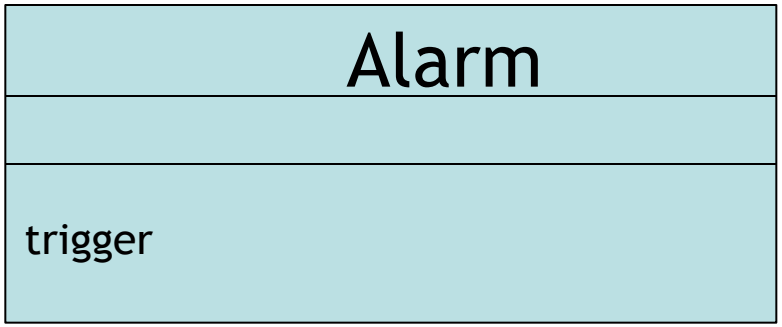
RC4
-
instance encrypt: aString using: key decript: encString using: key

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

Nuevo patrón de comportamiento

Sistema de Alarmas

- Supongamos una clase Alarma con un comportamiento “trigger” que reacciona a mensajes enviados por sensores



Pero….

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

✓Si esta inactive no toma en cuenta ningun aviso de los sensores

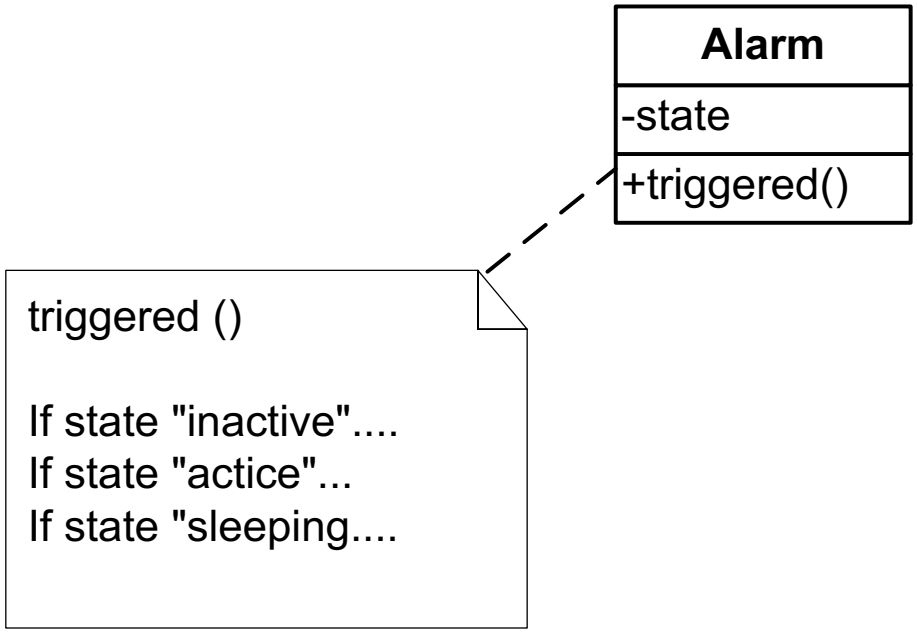
✓Si esta active tiene que reaccionar de acuerdo a su comportamiento como Alarma

✓Si esta “sleeping” se activa…..

✓Otras combinaciones….

Como resolvemos el problema?

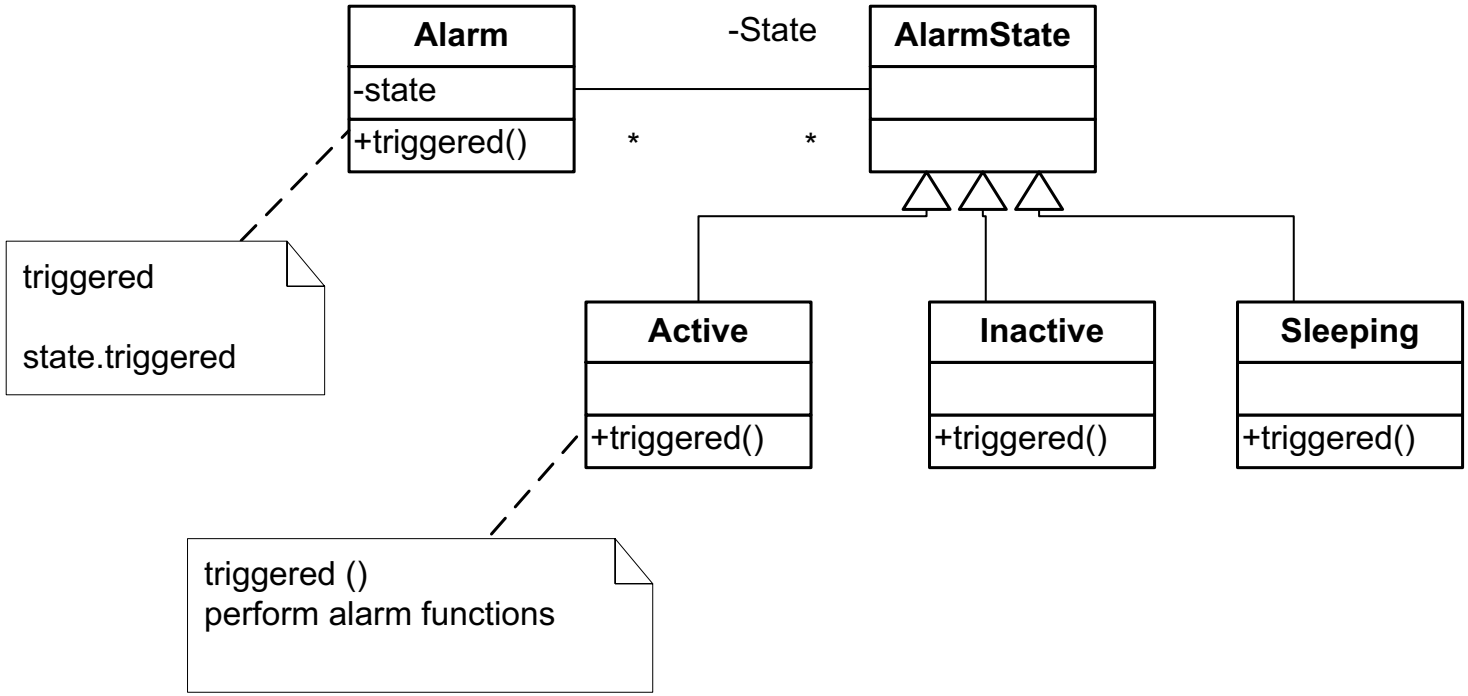
- Solucion “ingenua”:



Problemas con esta solucion?

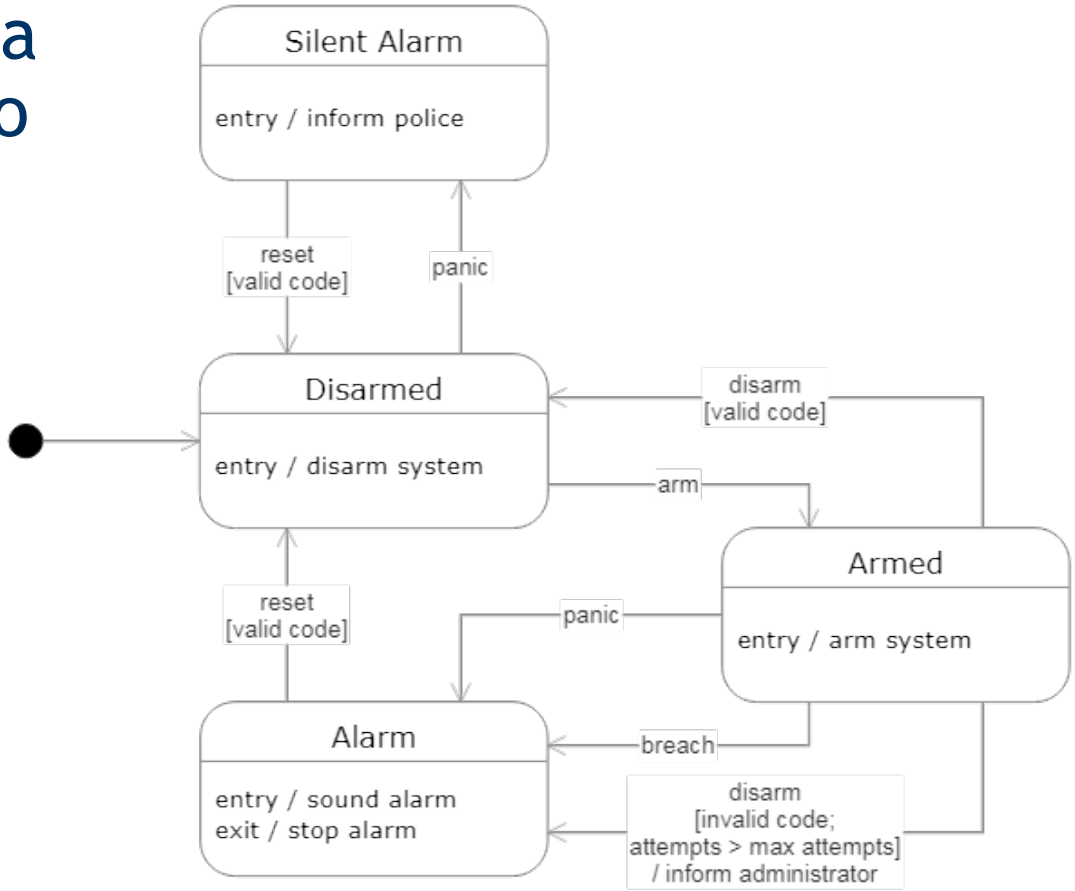
Una solucion mejor

- "Objetificar" el estado



Patrón State. Propósito

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



Patron State

- **Aplicabilidad:**

Usamos el patron State cuando:

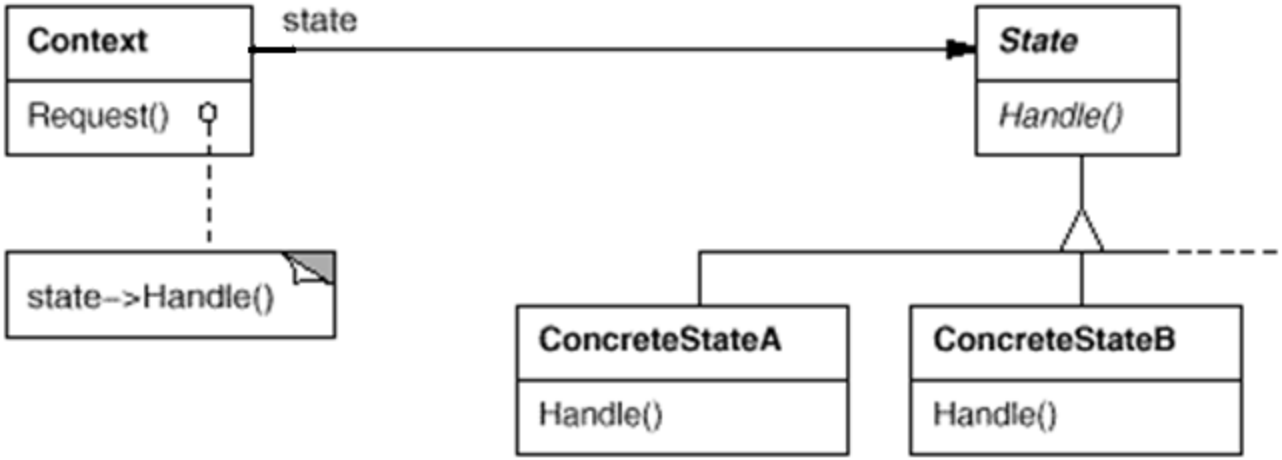
- Elcomportamiento de un objeto depende del estado en el que se encuentre.
- Los metodos tienen sentencias condicionales complejas que dependen del estado.
  Este estado se representa usualmente por constantes enumerativas y en muchas operaciones aparece el mismo condicional.
  El patron State reemplaza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

Patrón State

- **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).

Patron State

- Estructura



Patron State

- **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,Sleeping)
    - Cada subclase implementa el comportamiento respecto al estado especifico.

Temas interesantes

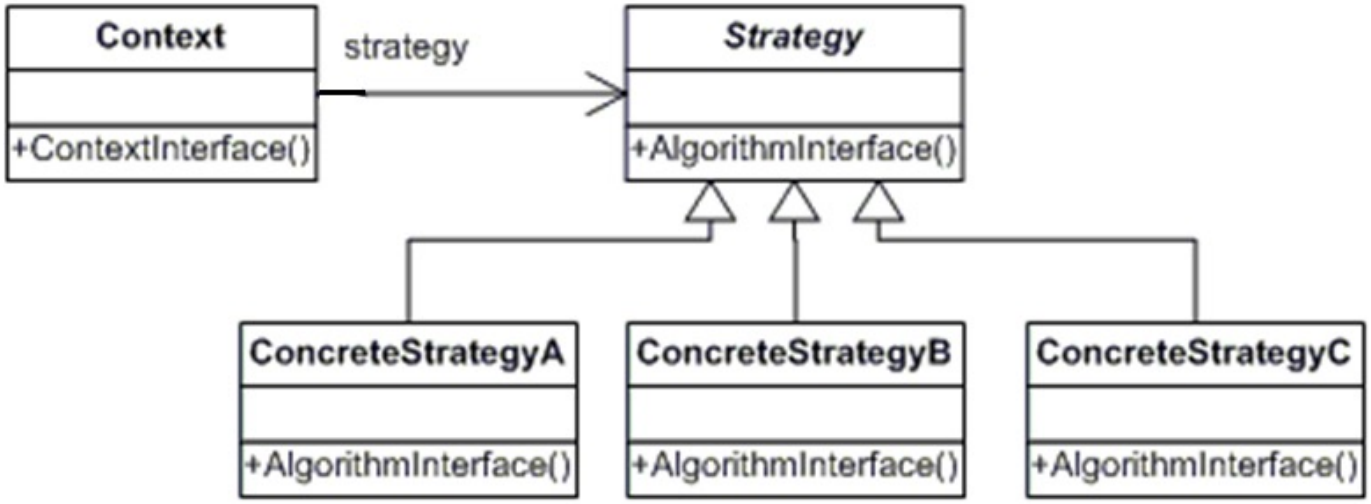
- Los estados son internos al contexto
- Ejecucion de los comportamientos de la alarma. Donde estan ubicados?
- Cómo cambiamos de estado?
- Muchos objetos Alarma, comparten la jerarquia de estados?

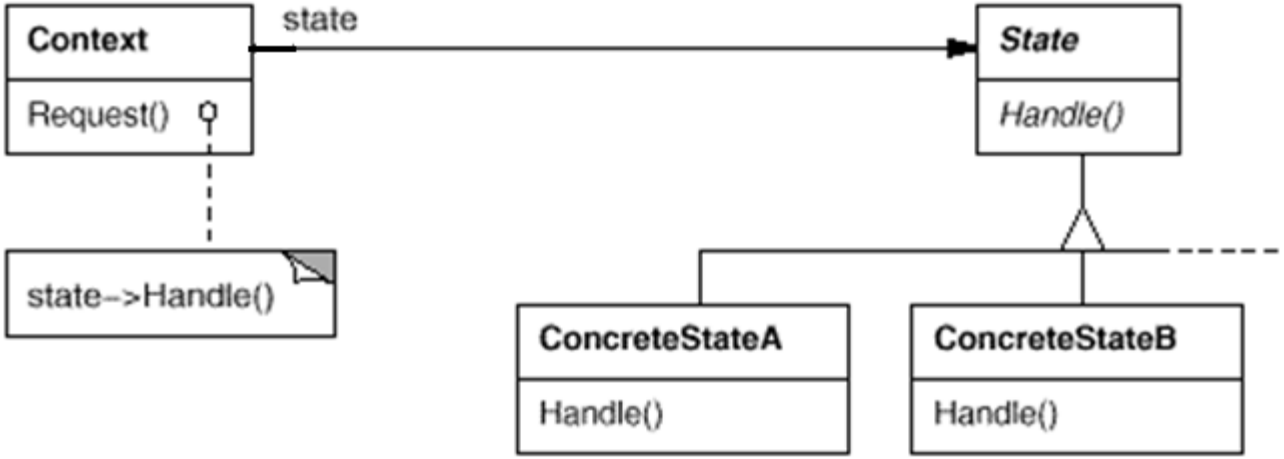
Patrón State

- **Consecuencias:**
  PROS
  - 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.

- CONS
  - 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

State o Strategy?





State o Strategy?

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

- 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 vs. Strategy

- En Strategy, las diferentes estrategias son conocidas desde afuera del contexto, por las clases clientes del contexto.
- En State, los diferentes estados son internos al contexto, no los eligen las clases clientes sino que la transición se realiza entre los estados mismos.
- Los diagramas se ven muy parecidos, pero 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 si.

≠Los strategies concretos no.