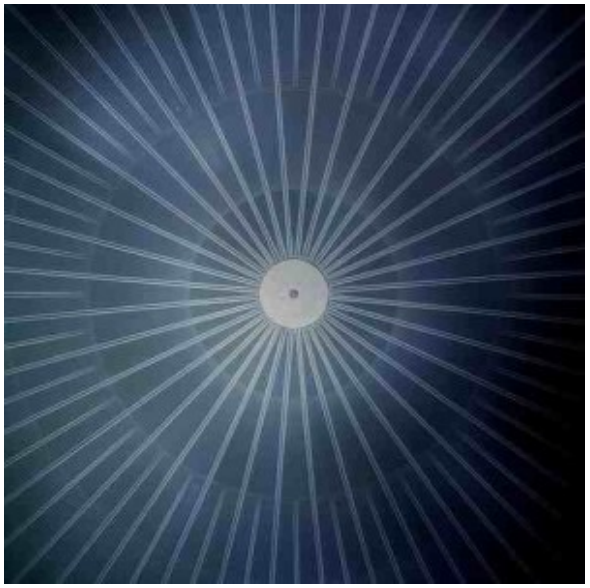


Más sobre patrones de diseño

Alejandra Garrido
Investigadora Independiente CONICET - Prof. Titular
Directora de la Maestría en Ingeniería de Software.
Vice-directora del LIFIA. UNLP
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?

Se denomina **patrón de conducta**, a una forma de conducta que hace las veces de modelo. Los patrones de conducta corresponden a normas específicas, que son guías que orientan la respuesta o acción ante situaciones o circunstancias específicas.[1]

¿Qué cosa es importante del diagrama de clases?

- Clases que componen el patrón (roles)
- Jerarquías
- Clases abstractas - Interfaces
- Métodos abstractos – Protocolo de interfaces
- Relaciones de conocimiento / composición

Nuevos patrones

(gracias a Gustavo Rossi por compartirme sus slides)

Ejemplo 1: Validación de Strings

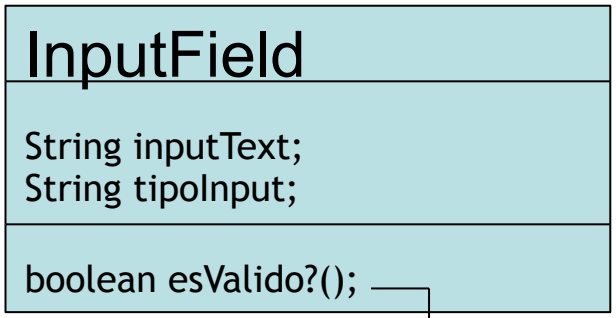
- Supongamos que estamos diseñando un formulario para ingresar datos

- 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?

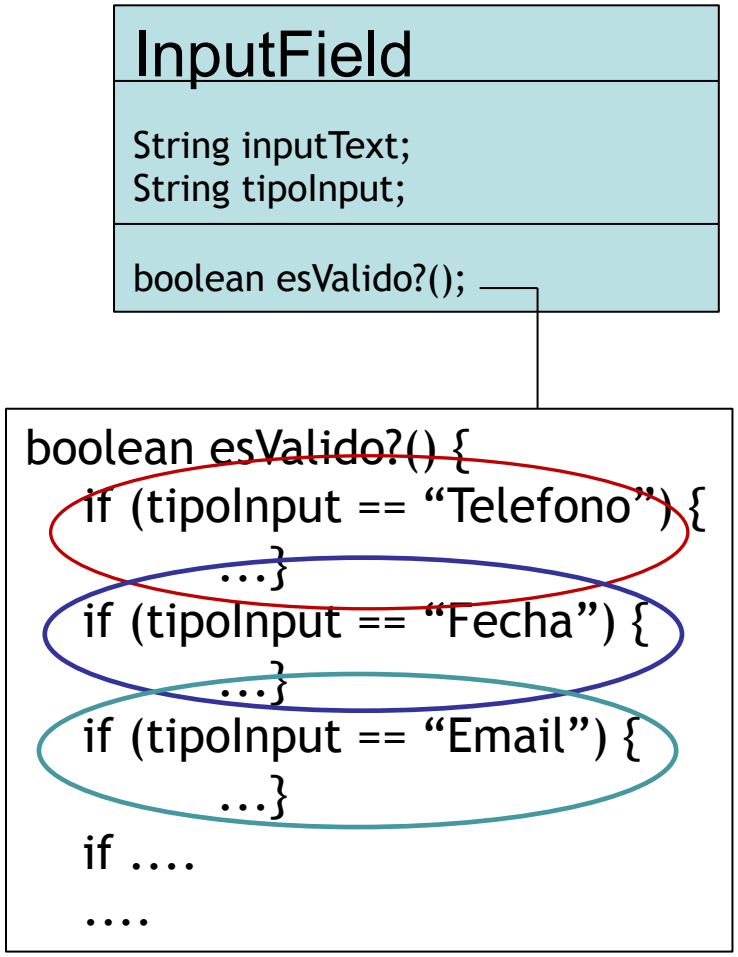
Cuál es una mejor solución?

- 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

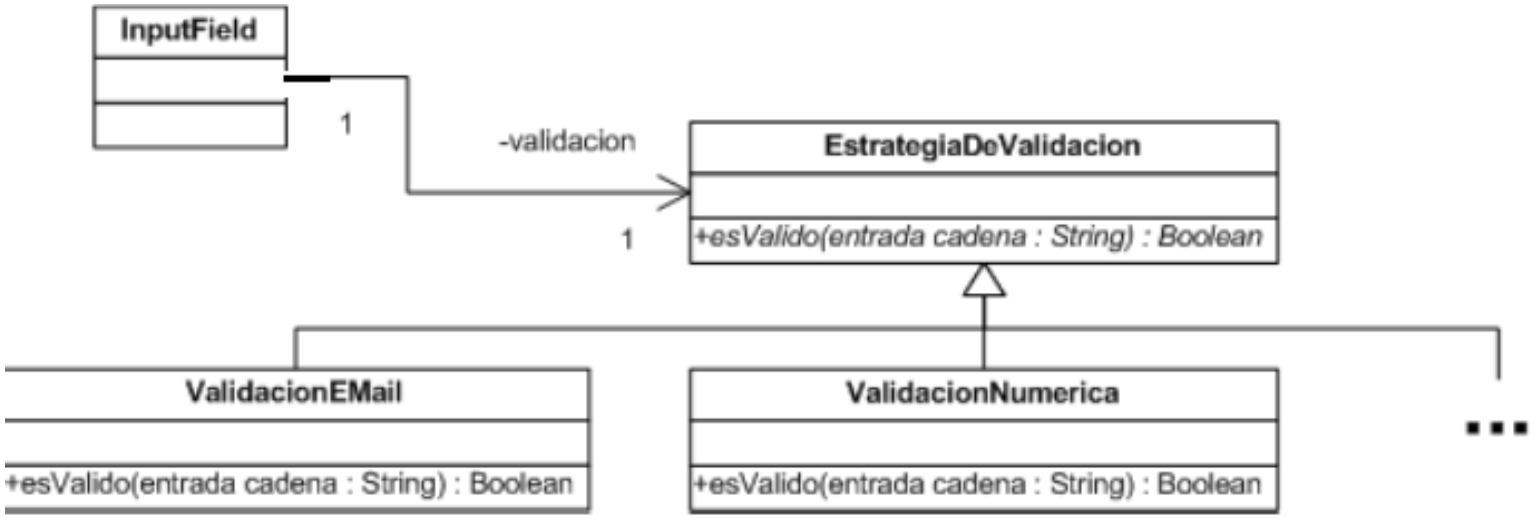
- Es lo que nos propone el patrón Strategy

Ejemplo 1: Validación de Strings

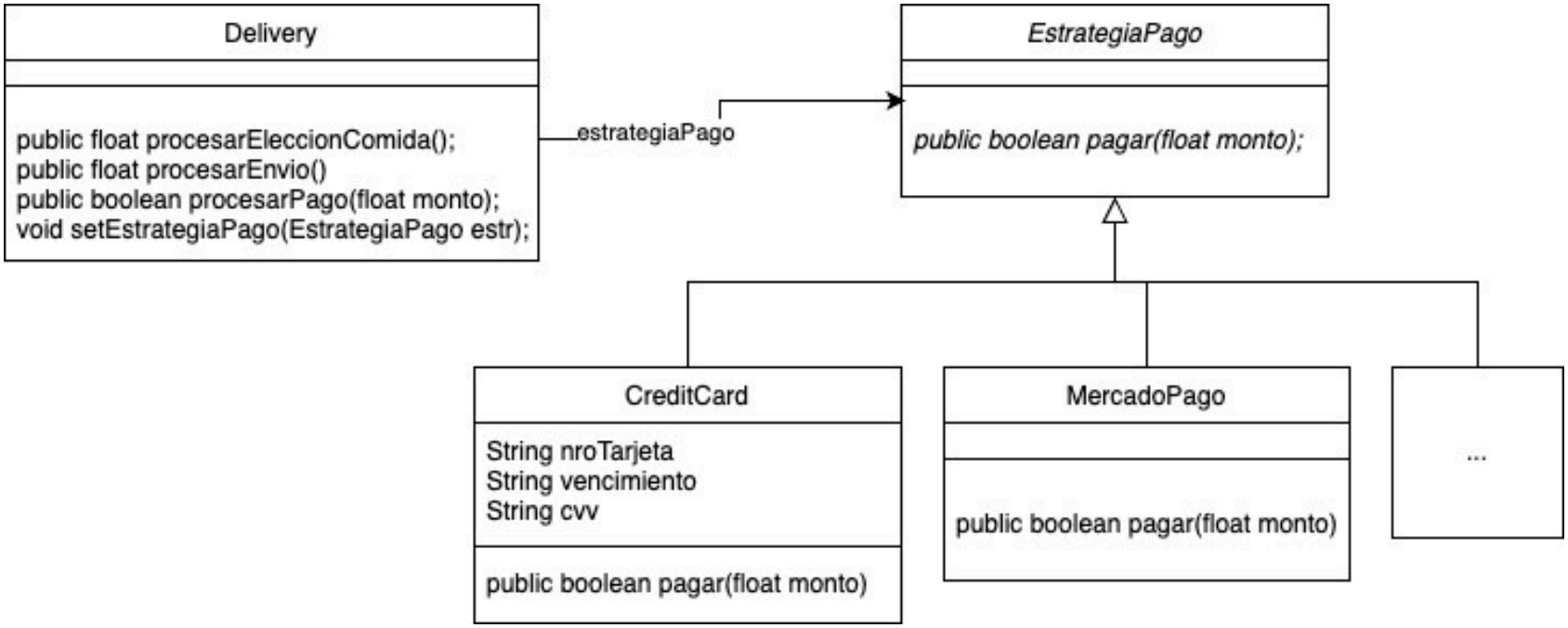


Aplicando Strategy

- 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

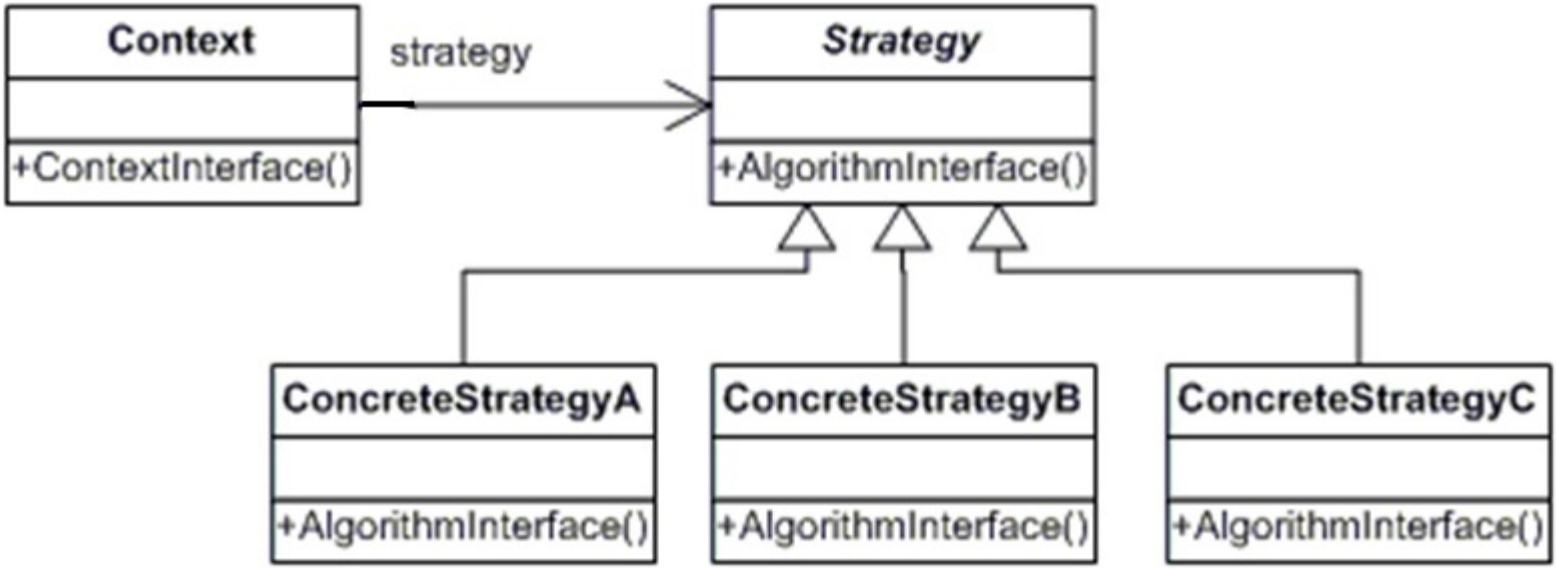
- **Intent:**
  - Desacoplar un algoritmo del objeto que lo utiliza.
  - Permitir cambiar el algoritmo que un objeto utiliza en forma dinámica.
  - Brindar flexibilidad para agregar nuevos algoritmos que lleven a cabo una función determinada.

Patrón Strategy

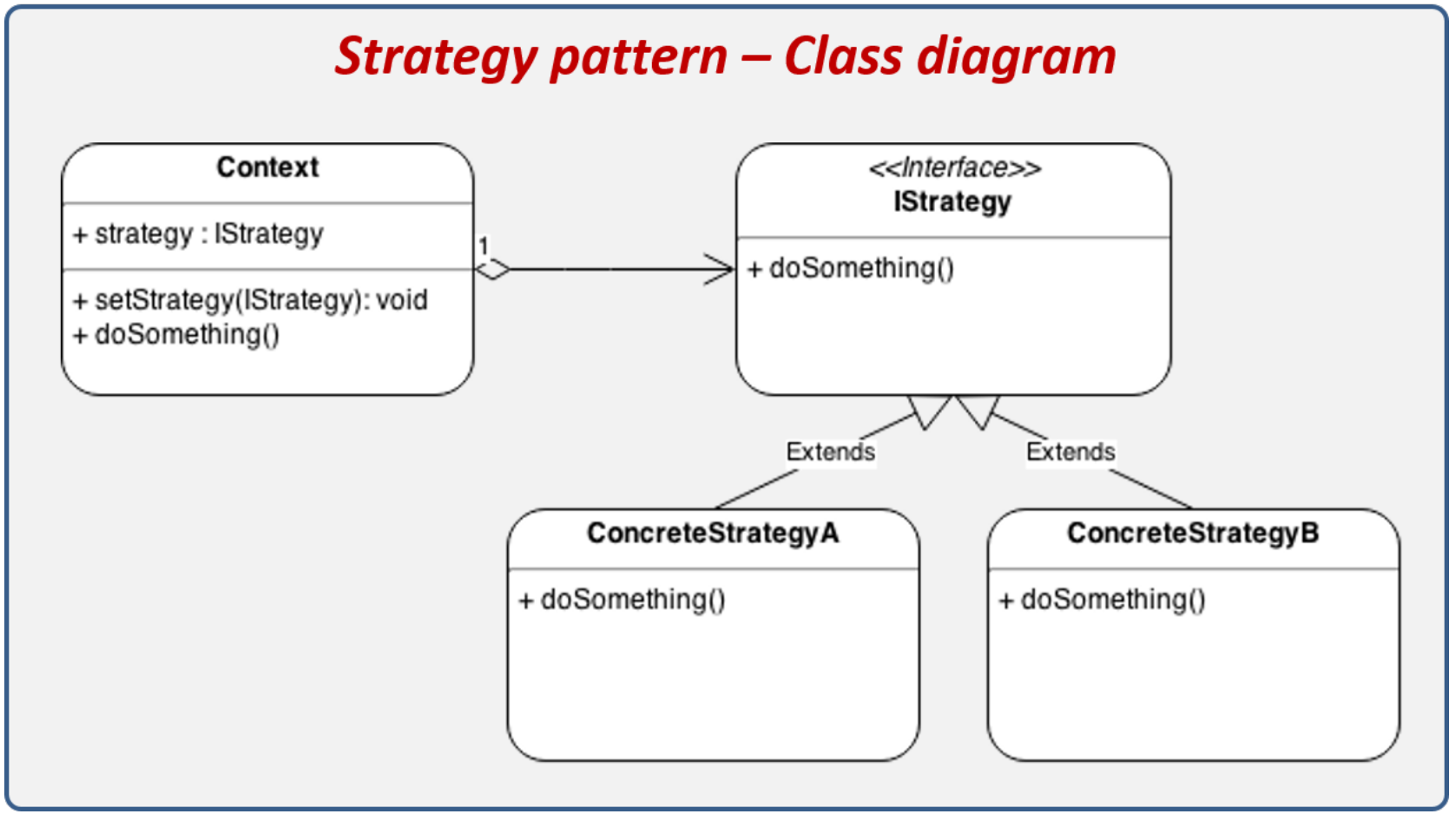
- **Applicability:**
  - Existen muchos algoritmos para llevar a cabo una tarea.
  - No es deseable codificarlos todos en una clase y seleccionar cual utilizar por medio de sentencias condicionales.
  - Cada algoritmo utiliza información propia. Colocar esto en los clientes lleva a tener clases complejas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

Patrón Strategy

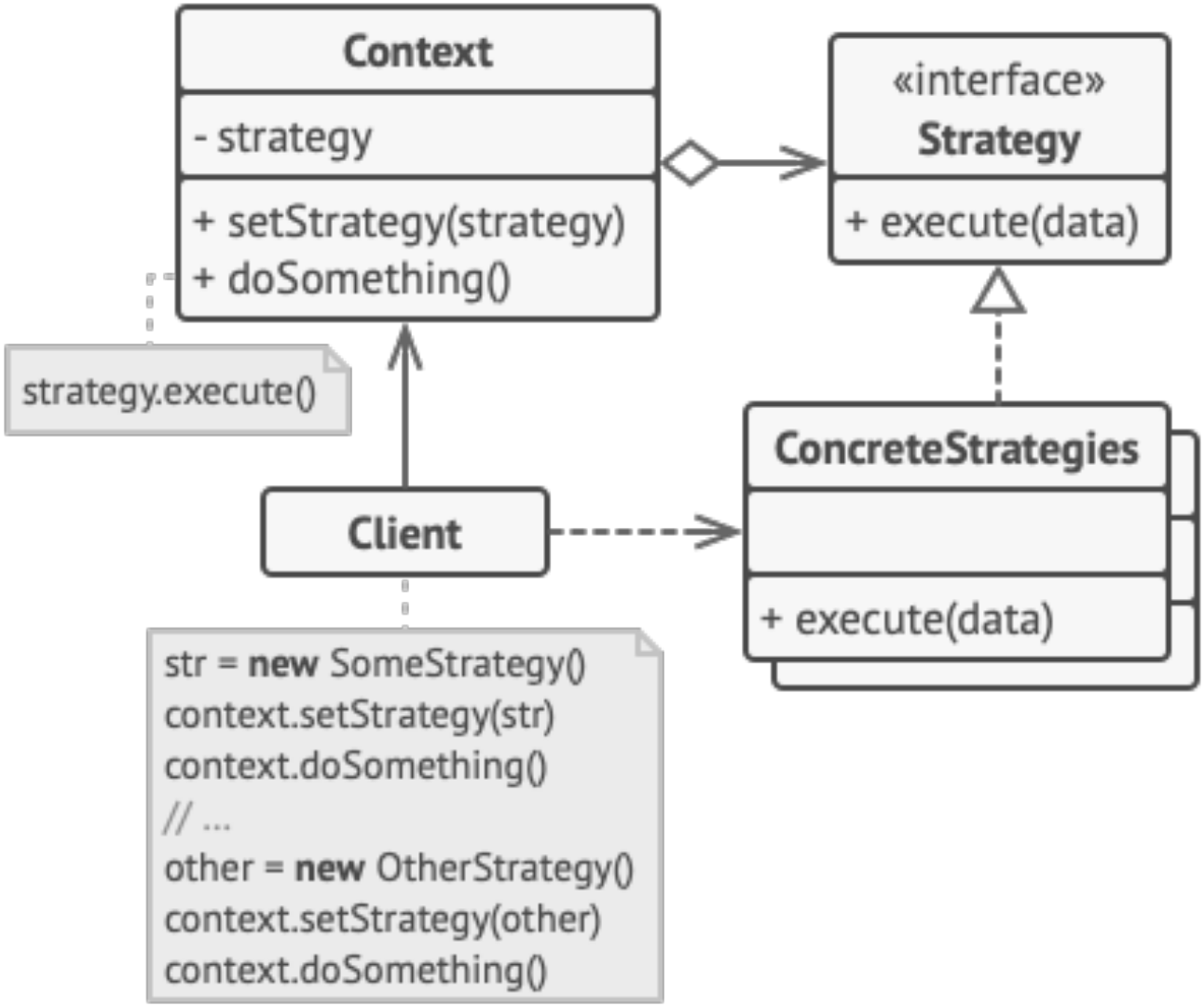
- **Solución-Estructura:**
  - Definir una familia de algoritmos, encapsular cada uno en un objeto y hacerlos intercambiables.
  - Son los clientes del contexto los que generalmente crean las estrategias.



Estructura del patrón Strategy con interfaces



Agregamos la clase Client



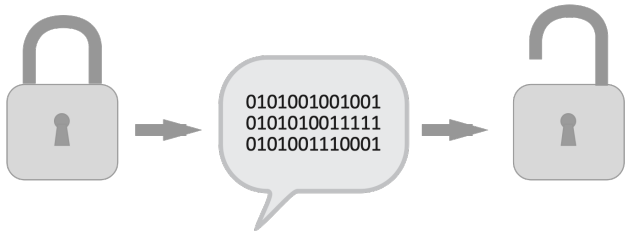
Patrón Strategy

- **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

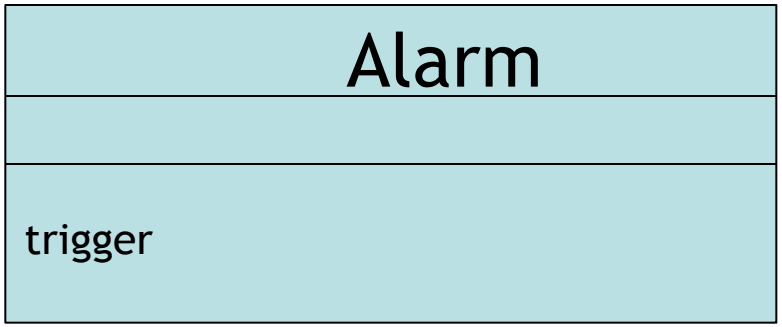
RC4
-
instance encrypt: aString using: key decript: encString using: key

- 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

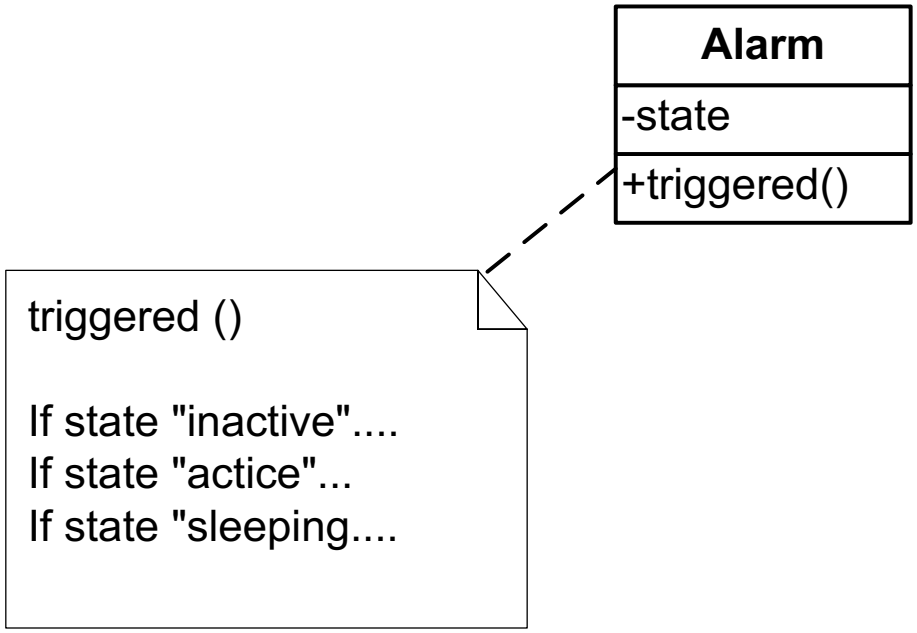
✓Si esta active tiene que reaccionar de acuerdo a su comportamiento como Alarma

✓Si esta “sleeping” se activa…..

✓Otras combinaciones….

Como resolvemos el problema?

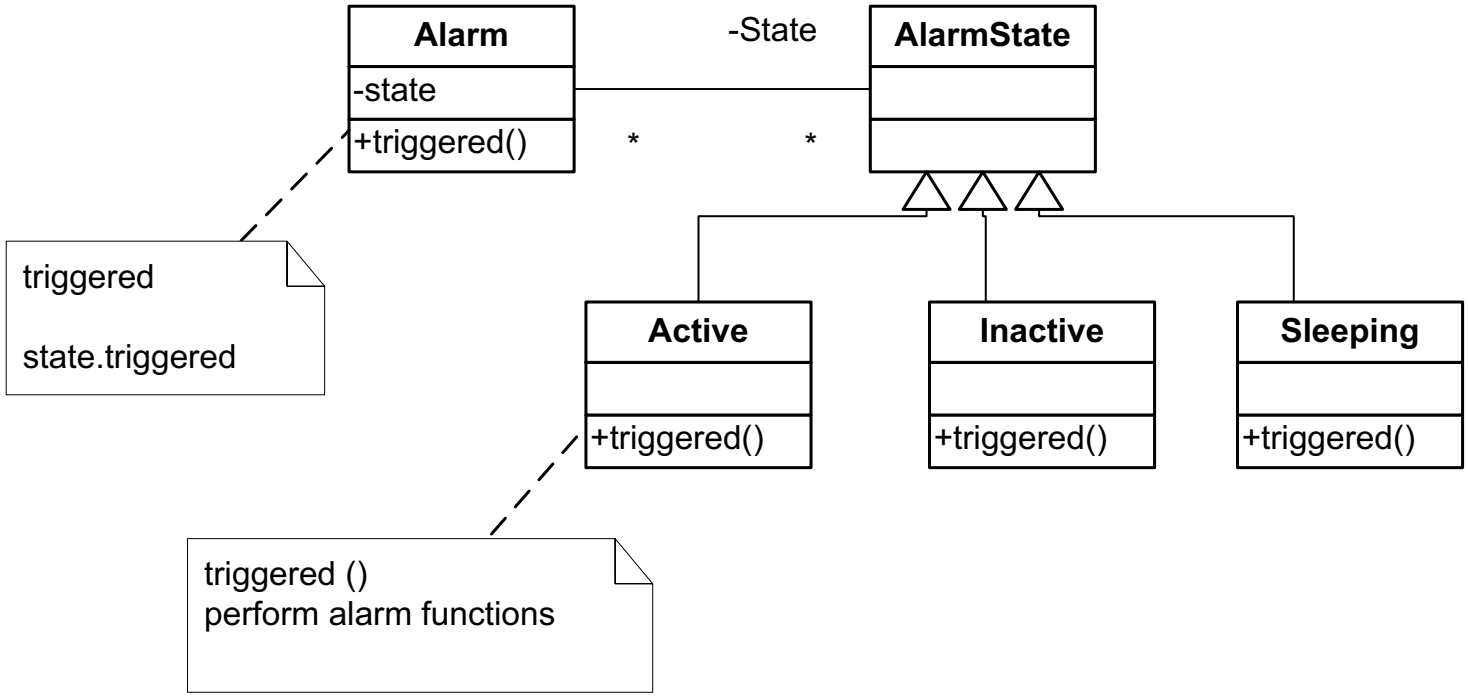
- Solucion “ingenua”:



Problemas con esta solucion?

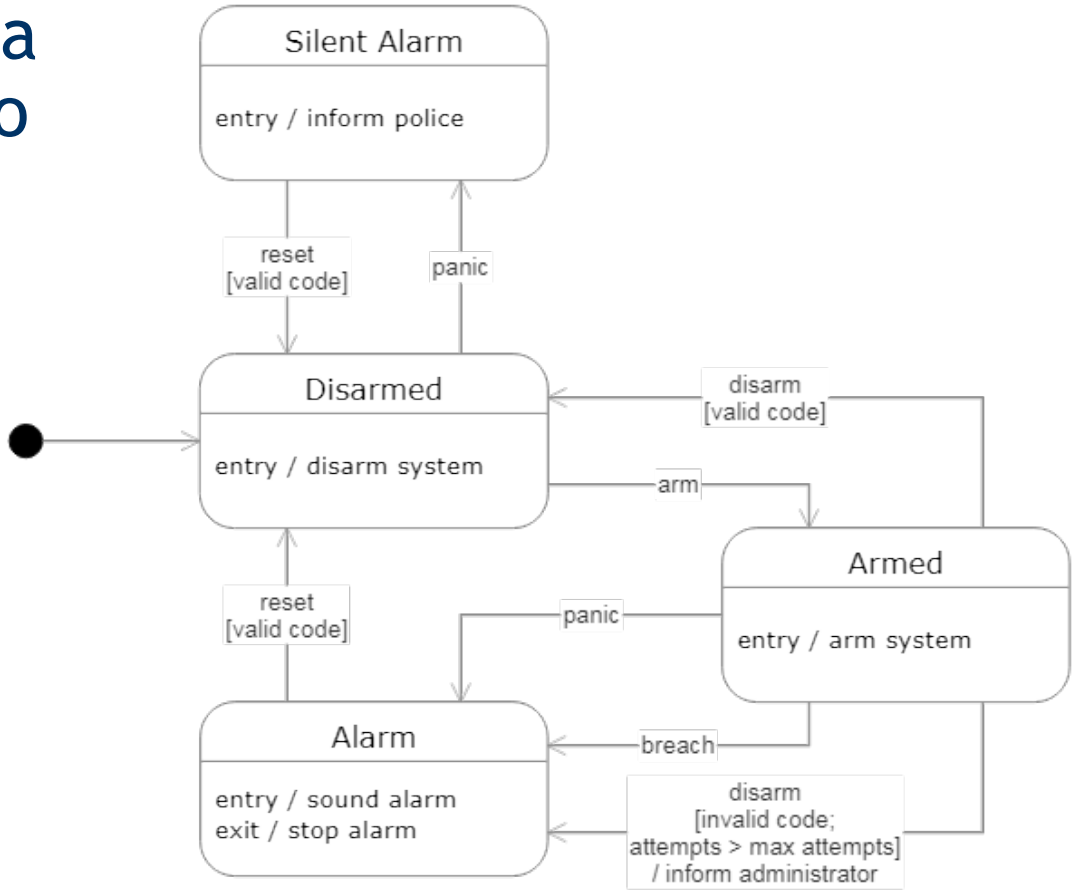
Una solucion mejor

- "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:

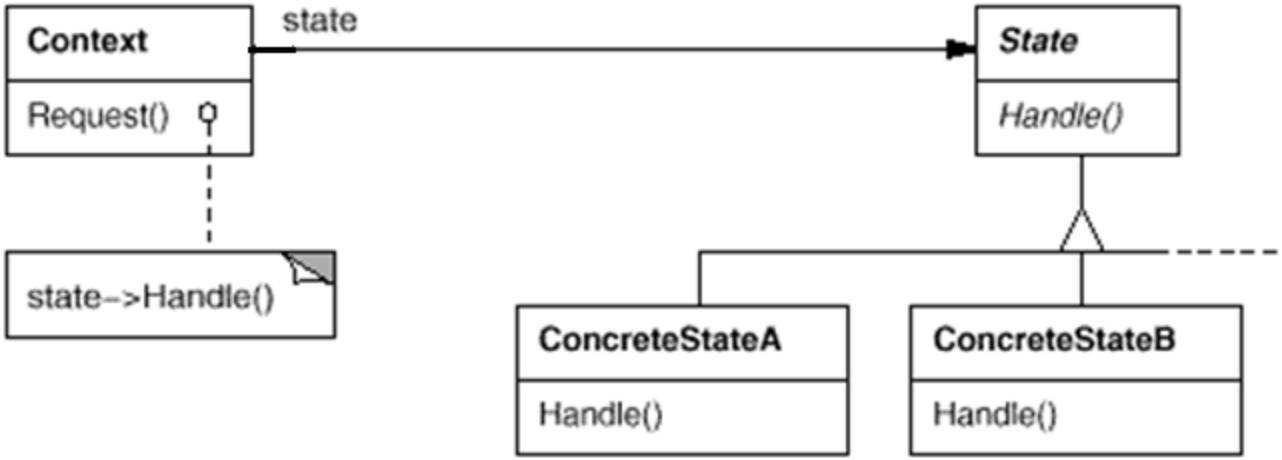
- Elcomportamiento de un objeto depende del estado en el que se encuentre.
- Los metodos tienen sentencias condicionales complejas que dependen del estado.
  Este estado se representa usualmente por constantes enumerativas y en muchas operaciones aparece el mismo condicional.
  El patron State reemplaza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

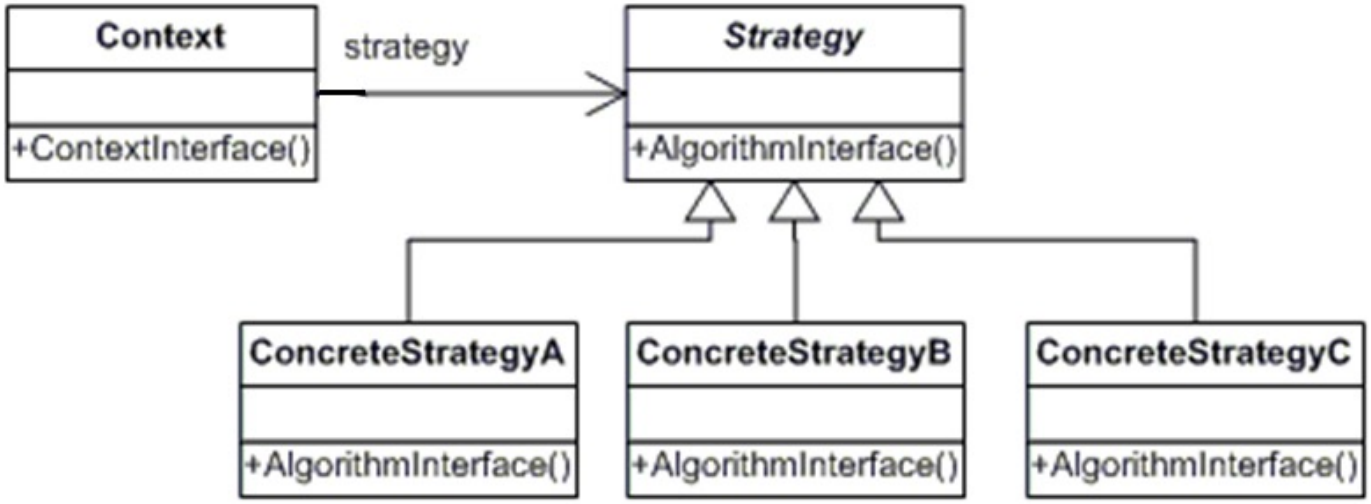
- Los estados son internos al contexto
- Ejecucion de los comportamientos de la alarma. Donde estan ubicados?
- Cómo cambiamos de estado?
- Muchos objetos Alarma, comparten la jerarquia de estados?

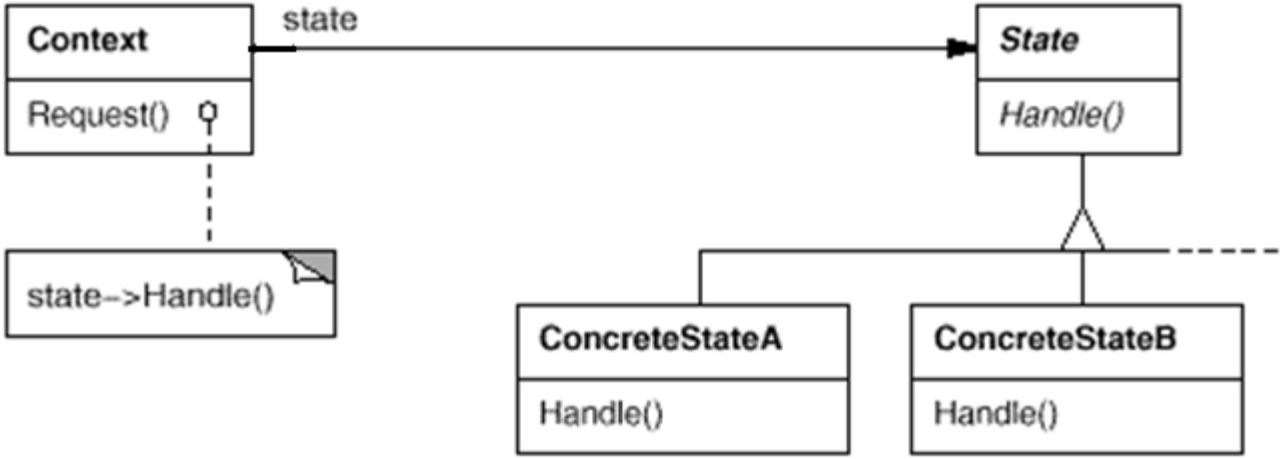
Patrón State

- **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.