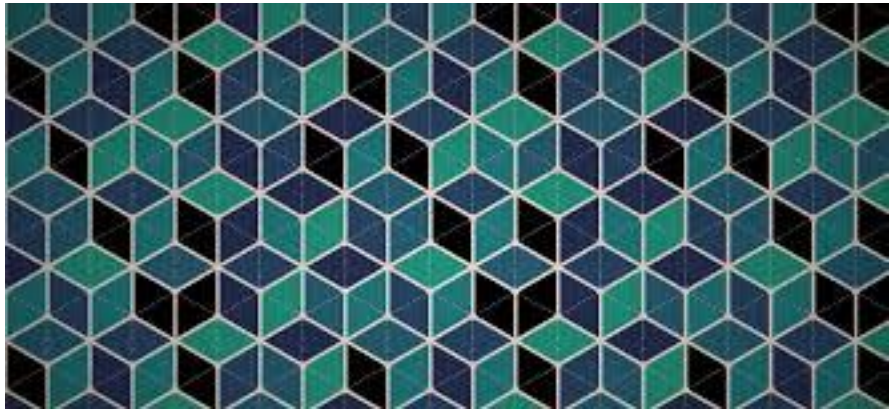


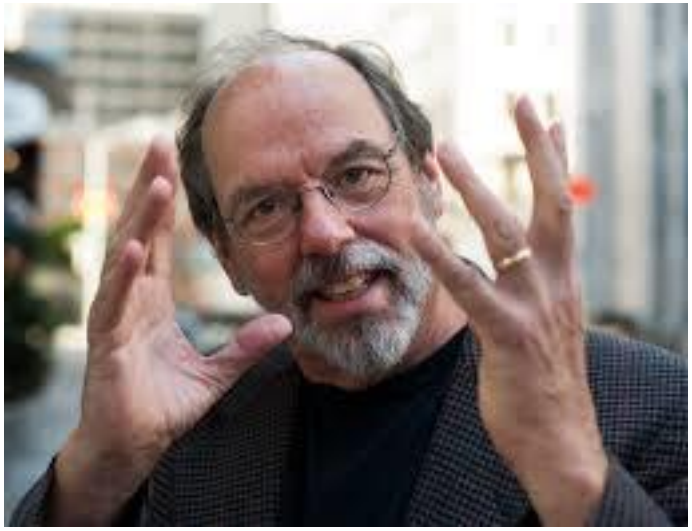
Patrones de diseño

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

Diseñar es difícil!

- Necesitamos encontrar
  - las abstracciones correctas,
  - la asignación correcta de responsabilidades,
  - las jerarquías adecuadas,
  - que nuestros diseños estén preparados para el cambio (que sean modulares, extensibles, comprensibles, reusables, etc.)

Los influencers del software en los ‘90

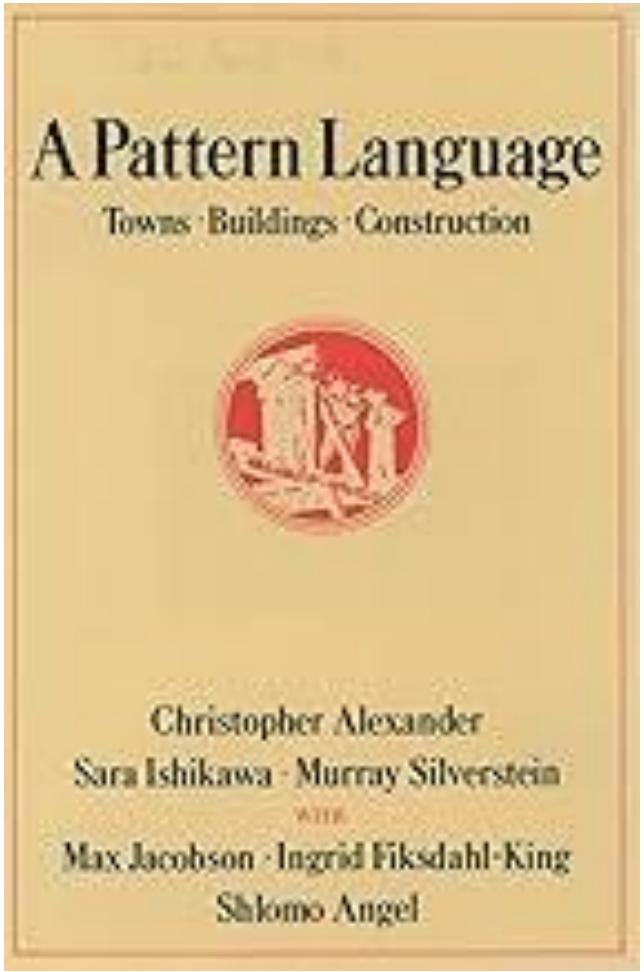


Ward Cunningham

Kent Beck

OOPSLA 87: “Using pattern languages for Object-Oriented programs”

El origen del concepto de patrones



- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

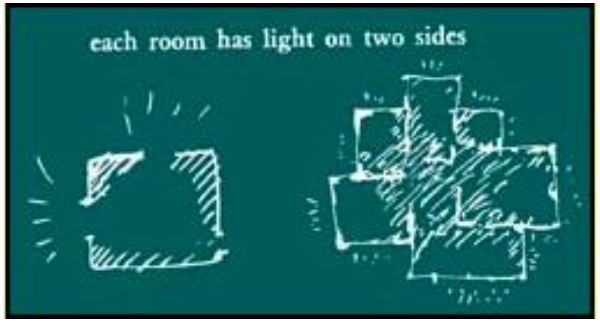
(Alexander et al. 1977)

Light on two sides of every room

- Problema: cómo diseñar las envolventes de cada habitación
- Cuando tienen la opción, las personas evitan las habitaciones con luz de un solo lado, y prefieren las habitaciones con luz en ambos lados

With light on two sides . . . . and without

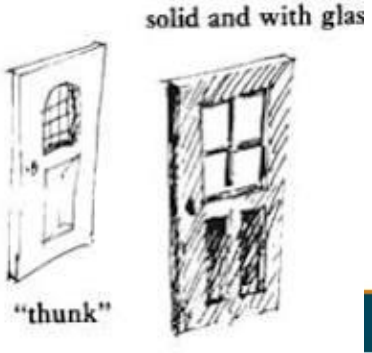
- *Por lo tanto:*
- Ubica cada habitación de manera que tenga espacio exterior en al menos dos lados y luego coloca ventanas en estas paredes exteriores para que la luz natural entre en cada habitación desde más de una dirección.



Otro patrón de Alexander: “Solid Door With Glass”

- . . this pattern finishes the doors defined by **Corner Doors (196)** and **Low Doorway (224)**. It also helps to finish **Tapestry of Light and Dark (135)** and **Interior Windows(194)**, since it requires glazing in the doors, and can help to create daylight in the darker parts of indoor places.

- **An opaque door makes sense in a vast house or palace, where every room is large enough to be a world unto itself; but in a small building, with small rooms, the opaque door is only very rarely useful.**
- What is needed is a kind of door which gives some sense of visual connection together with the possibility of acoustic isolation... Glazed doors have been traditional in certain periods - they are beautiful, and enlarge the sense of connection and make the life in the house one, but still leave people the possibility of privacy they need. A glazed door allows for a more graceful entrance ...
- Therefore:
- **As often as possible build doors with glazing in them, so, that the upper half at least, allows you to see through them. At the same time, build the doors solid enough, so that they give acoustic isolation and make a comfortable "thunk" when they are closed.**



De la arquitectura al software...

- ¿Cómo traducir este lenguaje de patrones de arquitectura de espacios al diseño orientado a objetos?

- ¿Qué es importante?
- Un patrón es un par problema-solución
- Los patrones tratan con problemas recurrentes y buenas soluciones (probadas) a esos problemas
- La solución es suficientemente genérica para poder aplicarse de diferentes maneras

- Veamos un problema en particular, buscaremos una solución, y luego vamos a generalizar al patrón

Ejemplo en IoT: sensores y actuadores

- Los actuadores se “suscriben” para recibir notificaciones a distintos eventos o cambios en los sensores. Supongamos un solo sensor que registra un valor y varios actuadores

```
public class Sensor {
    private List<Actuador> suscriptores = new ArrayList<>();
    private float valor;
    public void setValor(float unValor) {
            valor = unValor;
            this.changed();
    }
    protected void changed() {
            suscriptores.stream().forEach(sus -> sus.update(this));
    }
    public void agregarSuscriptor(Actuador actuador) { ....}
    // eliminarSuscriptor(), getValor(),....}
```

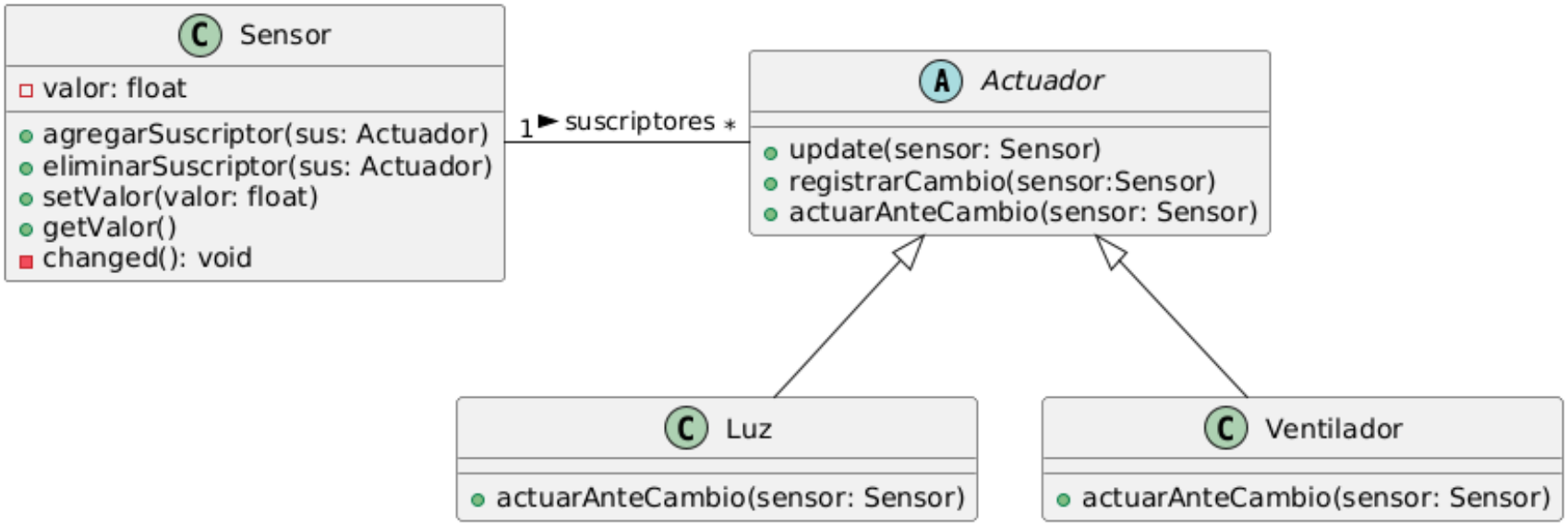
Ejemplo: actuadores

```
abstract class Actuador {
    public void update(Sensor sensor) {
            this.registrarCambio(sensor);
            this.actuarAnteCambio(sensor);
}

class Ventilador extends Actuador {
    public void actuarAnteCambio(Sensor sensor) {
        if (sensor.getValor() > 18.5) {
            this.encenderVentilador();
            System.out.println(“¡Ventilador encendido!");
        } else {
            this.apagarVentilador();
              System.out.println("Ventilador apagado.");
        }
    }}
```

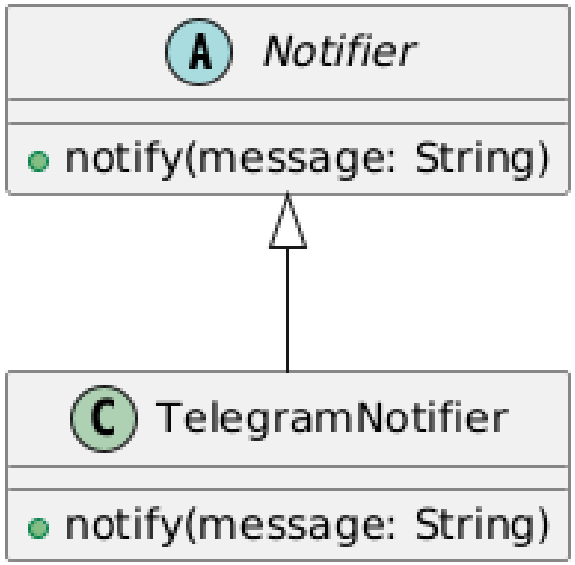
Ejemplo

- Diagrama de clases



Problema

- Queremos agregar un nuevo actuador que ante un cambio en el sensor envía un mensaje por Telegram avisando del nuevo valor
- Para esto ya existe una clase TelegramNotifier de una librería que queremos reutilizar. La clase no puede cambiarse (ni su código, ni la jerarquía a la que pertenece, ni las interfaces que implementa)

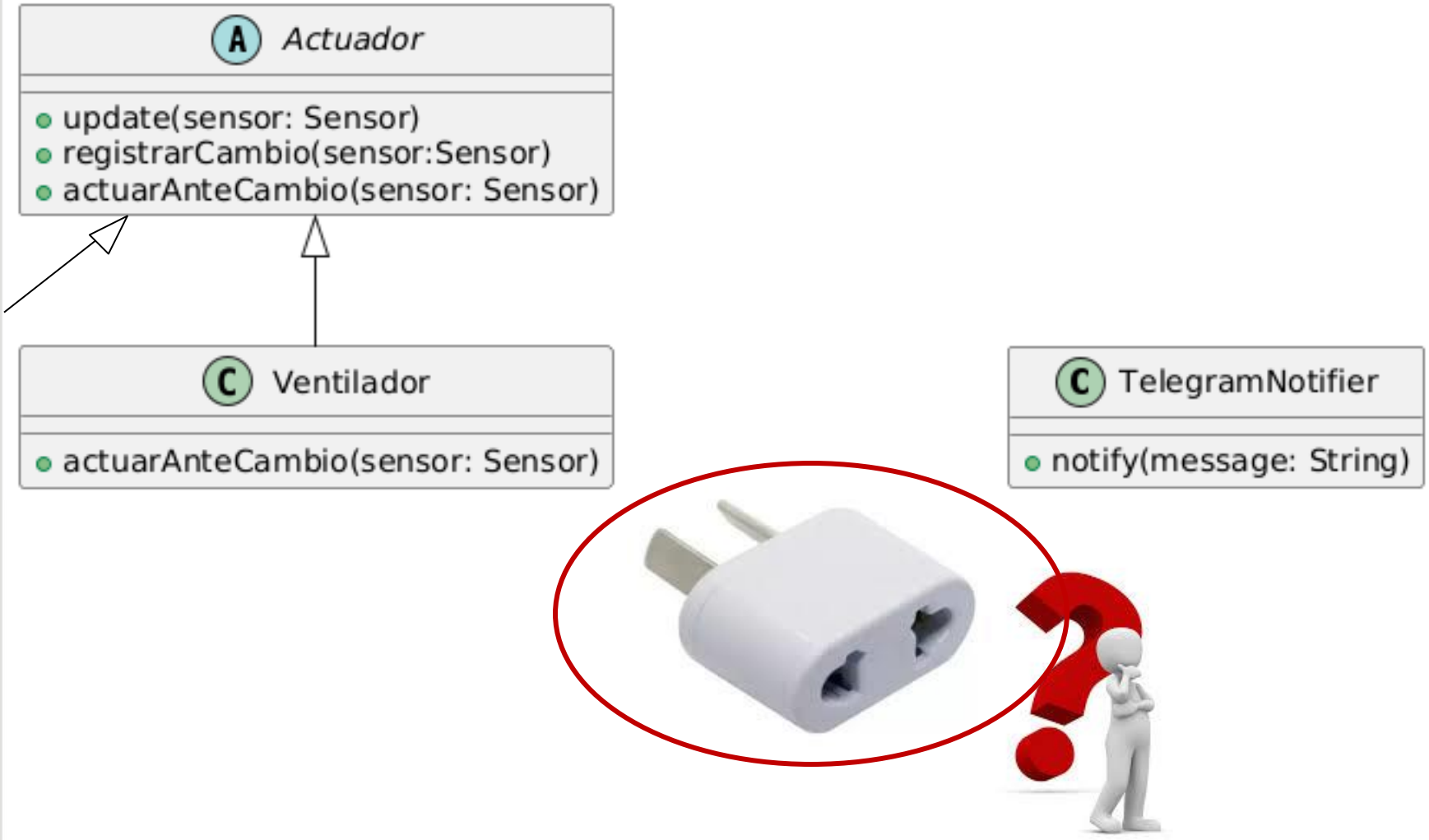


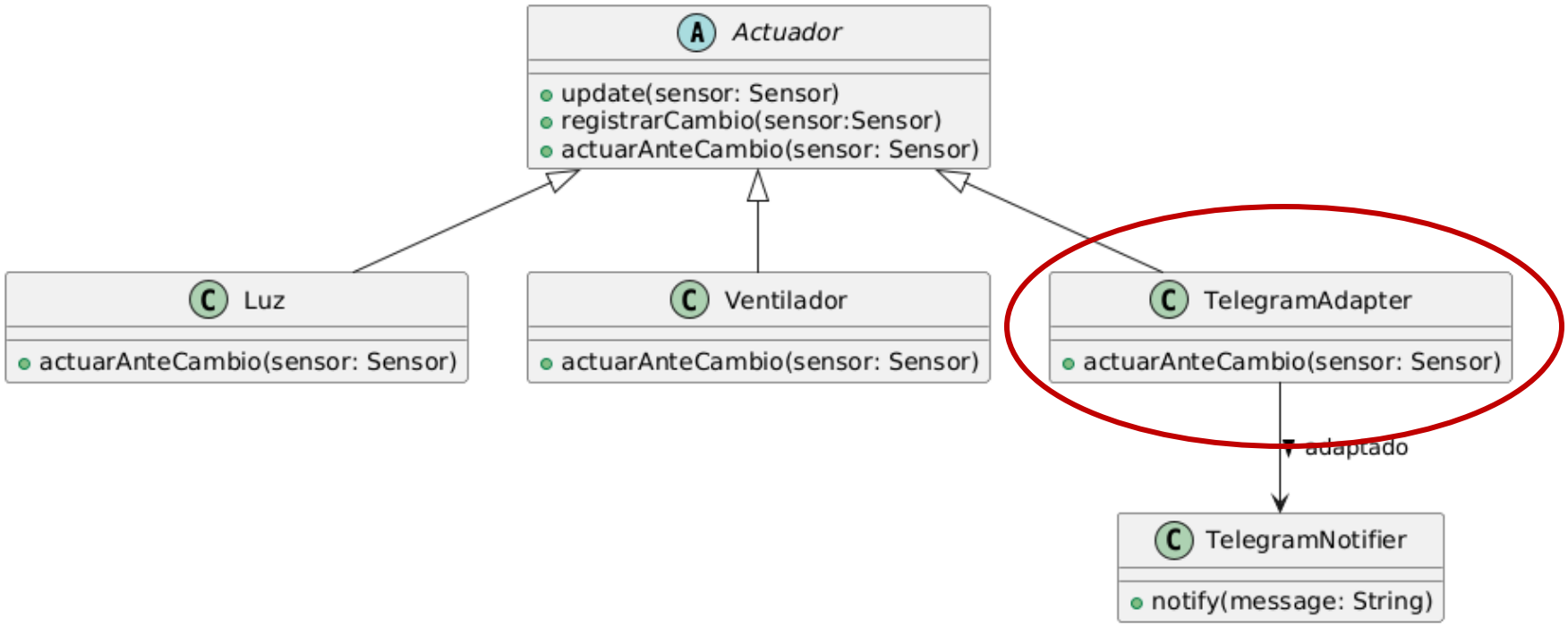
- ¿Por qué es un problema?
- Queremos que pueda suscribirse a los cambios en Sensor, pero Sensor solo permite objectos Actuador como suscriptores
- Sensor envía el mensaje update(sensor) a ellos

- ¿Solución más simple que funciona?

- ¿Qué otros problemas nos trae esa solución simple?

Pensamos en el mundo fuera del software


Actuador
update(sensor: Sensor)
registrarCambio(sensor:Sensor)
actuarAnteCambio(sensor: Sensor)
Ventilador
actuarAnteCambio(sensor: Sensor)
TelegramNotifier
notify(message: String)


Actuador
update(sensor: Sensor)
registrarCambio(sensor:Sensor)
actuarAnteCambio(sensor: Sensor)
Luz
actuarAnteCambio(sensor: Sensor)
Ventilador
actuarAnteCambio(sensor: Sensor)
TelegramAdapter
actuarAnteCambio(sensor: Sensor)
adaptado
TelegramNotifier
notify(message: String)

Patrón Adapter

- **Intención:**

  "Convertir" la interfaz de una clase en otra que el cliente espera.

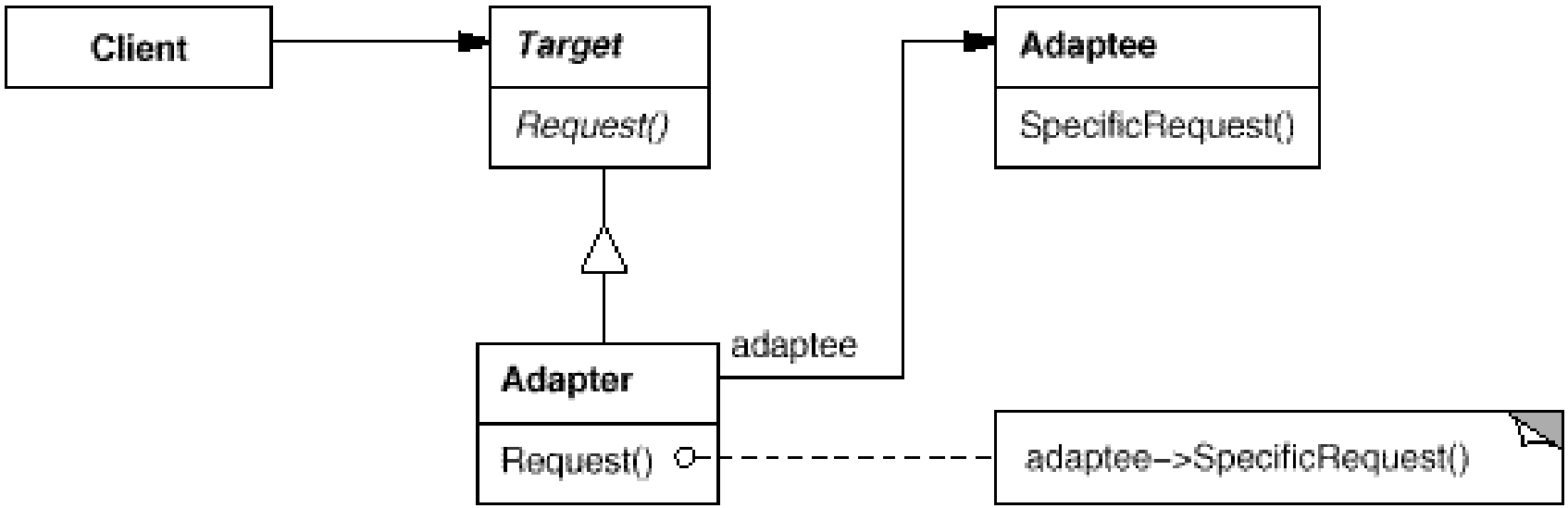
  Adapter permite que ciertas clases con interfaces incompatibles puedan trabajar en conjunto

- **Aplicabilidad:**

  Use Adapter cuando Ud. quiere usar una clase existente y su interfaz no es compatible con lo que precisa

Patrón Adapter

- **Estructura**



- Cómo se interpreta este diagrama?
- Cómo usamos esta información? Es suficiente?
- Qué otra información necesitamos?

Volviendo al patrón Adapter

- **Participantes**

Target

Define la interfaz especìfica que usa el cliente

Client

Colabora con objetos que satisfacen la interfaz de Target

Adaptee

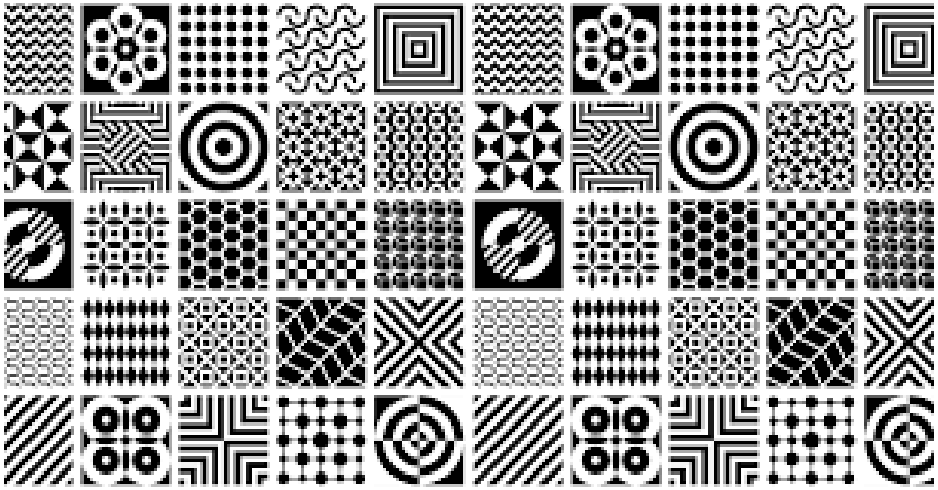
Define una interfaz que precisa ser adaptada

Adapter

Adapta la interfaz del Adaptee a la interfaz de Target

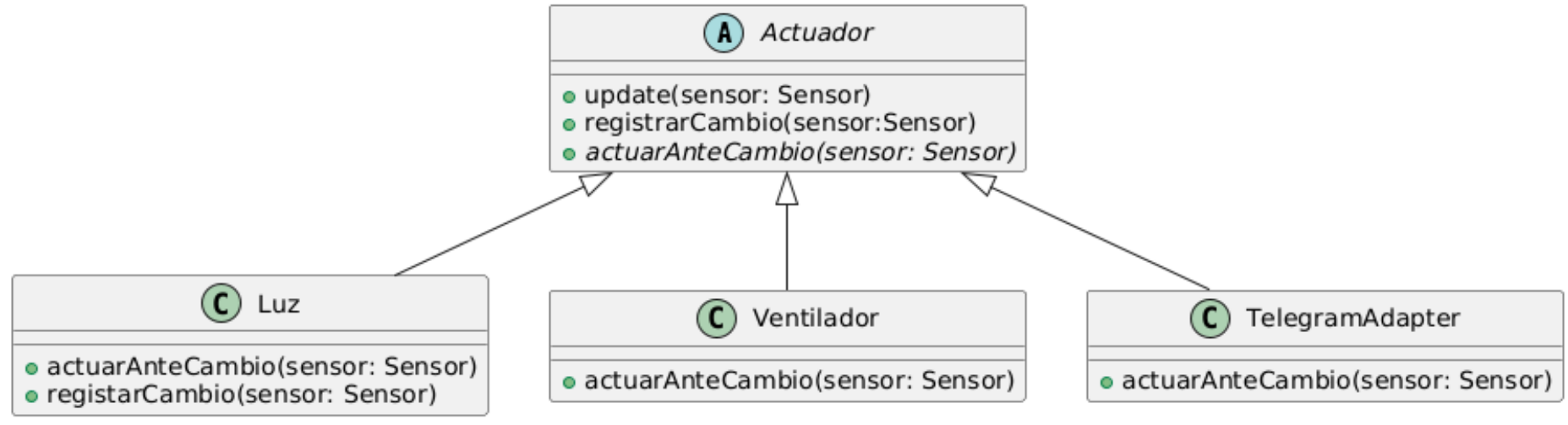
Elementos del Patrón Adapter

- Colaboraciones
  - Los objetos Client llaman a las operaciones en la instancia del Adapter
  - A su vez, el Adapter llama a las operaciones definidas en el Adaptee
- Consecuencias
  - Una misma clase Adapter puede usarse para muchos Adaptees (el Adaptee y todas sus subclases)
  - El Adapter puede agregar funcionalidad a los adaptados
  - Se generan más objetos intermediarios
- Implementación:
  - pluggable adapters, parameterized adapters



Nuevo patrón

Volvamos al ejemplo anterior: actuadores




```
abstract class Actuador {
    public void update(Sensor sensor) {
            this.registrarCambio(sensor);
            this.actuarAnteCambio(sensor);
}
class Ventilador extends Actuador {
    public void actuarAnteCambio(Sensor sensor) {
        if (sensor.getValor() > 18.5) { ....    }}
```

- Problema
  - Necesitamos que cada vez que un actuador es notificado haga 2 cosas: registrar el cambio (por ejemplo en un log de cambios) y realizar la acción, y en ese orden
  - Si se agregan nuevas subclases, deben realizar lo mismo
- Solución naive?
- Consecuencias de la solución naive?

Patrón Template Method

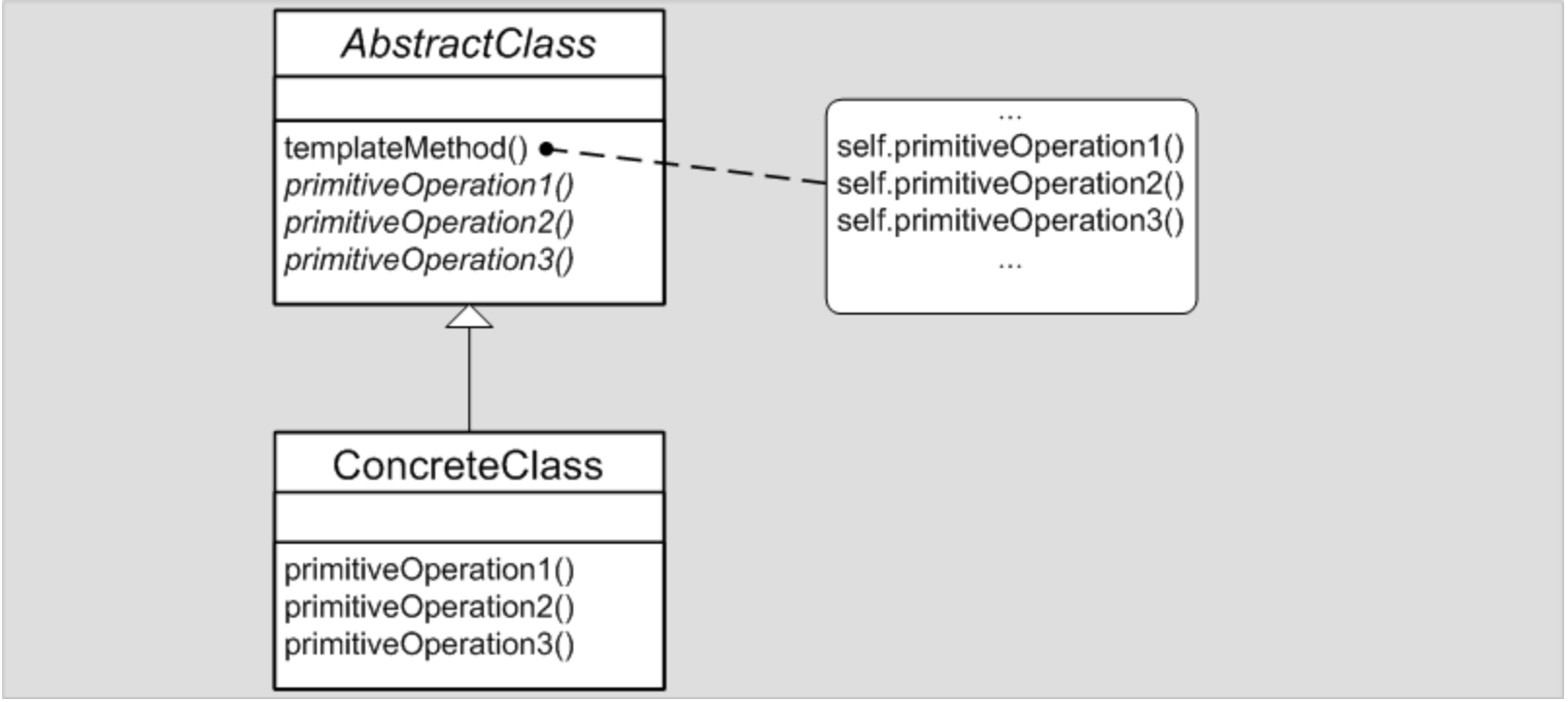
- **Intención:**

  Definir el esqueleto de un algoritmo en un método, difiriendo algunos pasos a las subclases.
  Template Method permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar la estructura del algoritmo.

- **Aplicabilidad:** usar Template Method
  - para implementar las partes invariantes de un algoritmo una vez y dejar que las subclases implementen los aspectos que varian;
  - para evitar duplicación de código entre subclases;
  - para controlar las extensiones que pueden hacer las subclases

Patrón Template Method

- **Estructura**



Patrón Template Method

- **Participantes**

AbstractClass

- Implementa un método que contiene el esqueleto de un algoritmo (el template method). Ese método llama a operaciones primitivas así como operaciones definidas en AbstractClass

- Declara operaciones primitivas abstractas que las subclases concretas deben definir para implementar los pasos de un algoritmo

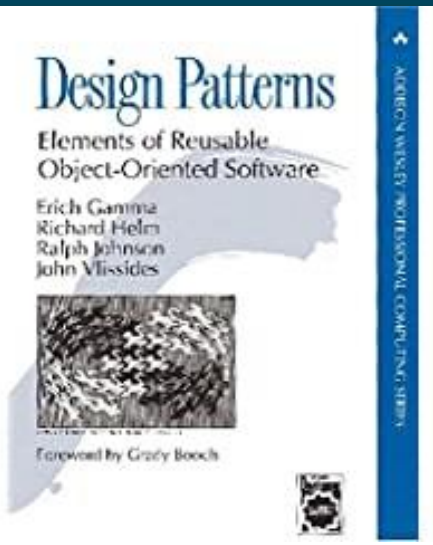
ConcreteClass

- Implementa operaciones primitivas que llevan a cabo los pasos específicos del algoritmo

Elementos del Patrón Template Method

- Colaboraciones
  - ConcreteClass confía en que la superclase implemente las partes invariantes del algoritmo
- Consecuencias
  - Técnica fundamental de reuso de código
  - Lleva a tener **inversión de control** (la superclase llama a las operaciones definidas en las subclases)
  - El template method llama a dos tipos de operaciones:
    - operaciones primitivas (abstractas en AbstractClass y que las subclases *tienen* que definir)
    - operaciones concretas definidas en AbstractClass y que las subclases *pueden* redefinir si hace falta (**hook methods**)
- Implementación:
  - minimizar la cantidad de operaciones primitivas que las subclases deben redefinir

GoF: el catálogo de patrones de diseño OO



Gamma, Helms, Johnson, Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software (GoF: Gang of Four)

- Cada patrón del catálogo describe una solución simple y elegante a un problema específico en el diseño de software OO
- Los patrones capturan soluciones que han sido desarrolladas y evolucionaron en el tiempo, después del trabajo de desarrolladores en rediseñar, fallar y reflexionar hasta lograr un diseño flexible y reusable
- Cada patrón nombra, abstrae e identifica los aspectos claves de un diseño que resuelve un problema particular: clases que participan, sus roles, relaciones, distribución de responsabilidades, consecuencias, pros y contras

Partes de la descripción de un patrón (GoF)

- Nombre (y otros nombres por los que puede conocerse)
- Intención
- Motivación
- Aplicabilidad
- Estructura
- Participantes
- Colaboraciones
- Consecuencias
- Implementación
- Código (C++)
- Usos conocidos
- Patrones relacionados

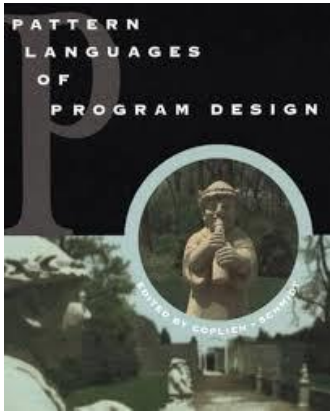
¿Qué cosa es importante estudiar y recordar?

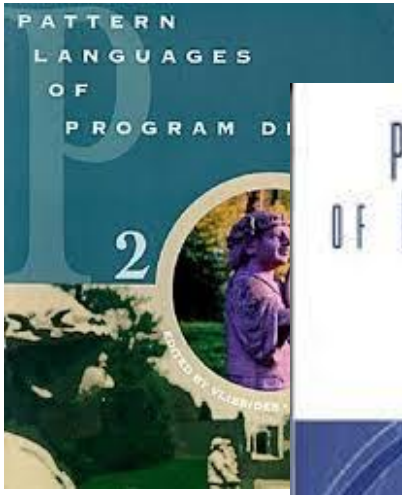
- Propósito
- Estructura:
  - clases que componen el patrón (roles),
  - cómo se relacionan (jerarquías, clases abstractas/interfaces, métodos abstractos – protocolo de interfaces, conocimiento/composición)
- Variantes de implementación
- Consecuencias positivas y negativas
- Relación con otros patrones

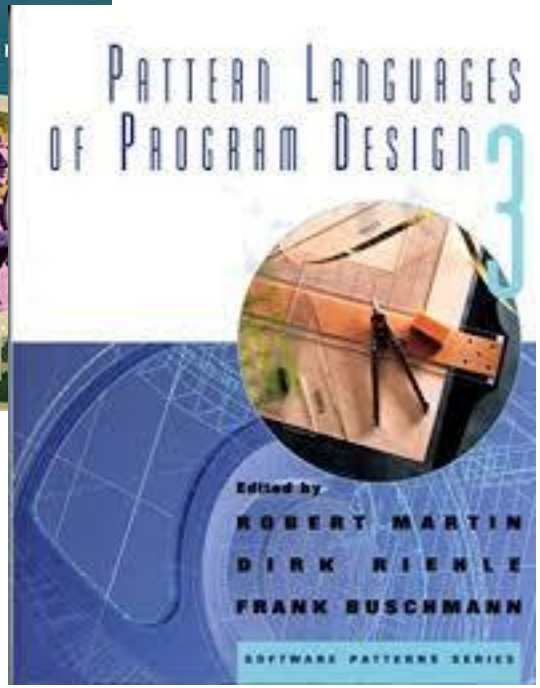
PLoP: Pattern Languages of Programs Conference

- Donde se han presentado patrones para problemas de distinta índole y se discuten en grupo entre pares
- PLoP, EuroPLoP, AsianPLoP, KoalaPLoP, SugarLoaf PLoP, etc.
- Auspiciadas por Hillside Group (hillside.net)

- SugarLoaf PLoP 2016. Tango Edition https://hillside.net/sugarloafplop/2016/







- Pueden encontrar patrones con otro formato
- Por ejemplo los patrones de Scrum siguen el formato de Alexander (https://scrumbook.org/)

Recursos

- Ward Cunningham. The Starting Point of Software Patterns: https://www.youtube.com/watch?v=_V0kVOLOCrY
- Scrum Patterns: https://scrumbook.org/
- Pattern Languages of Program Design (Vol. 1, 2, 3, 4, 5). Addison-Wesley Professional.
- 32th PLoP Conference: https://plopcon.org/plop2025/
- The Hillside Group: https://hillside.net/
- Christopher Alexander - keynote en OOPSLA'96 https://www.youtube.com/watch?v=98LdFA-_zfA