

SEMINARIO

2

Principios SOLID

Principios de diseño

Diseño del Software

Grado en Ingeniería Informática del Software

Curso 2017-2018

Principios SOLID

SRP	Single Responsibility Principle <i>Principio de responsabilidad única</i>
OCP	Open-Closed Principle <i>Principio de abierto-cerrado</i>
LSP	Liskov Substitution Principle <i>Principio de sustitución de Liskov</i>
DIP	Dependency Inversion Principle <i>Principio de inversión de dependencias</i>
ISP	Interface Segregation Principle <i>Principio de segregación de interfaces</i>

Principio de responsabilidad única



*Una clase debería tener un
único motivo para cambiar*

Principio de responsabilidad única (SRP)

Employee

```
+calculatePay(): double  
+calculateTaxes(): double  
+writeToDisk()  
+readFromDisk()  
+createXML(): String  
+parseXML(String xml)  
+printEmployeeReport(PrintWriter)  
+printPayrollReport(PrintWriter)  
+printTaxReport(PrintWriter)
```

¿Qué podemos decir de esta clase?

¿Qué podemos decir de Empleado?

○ Que hace demasiadas cosas

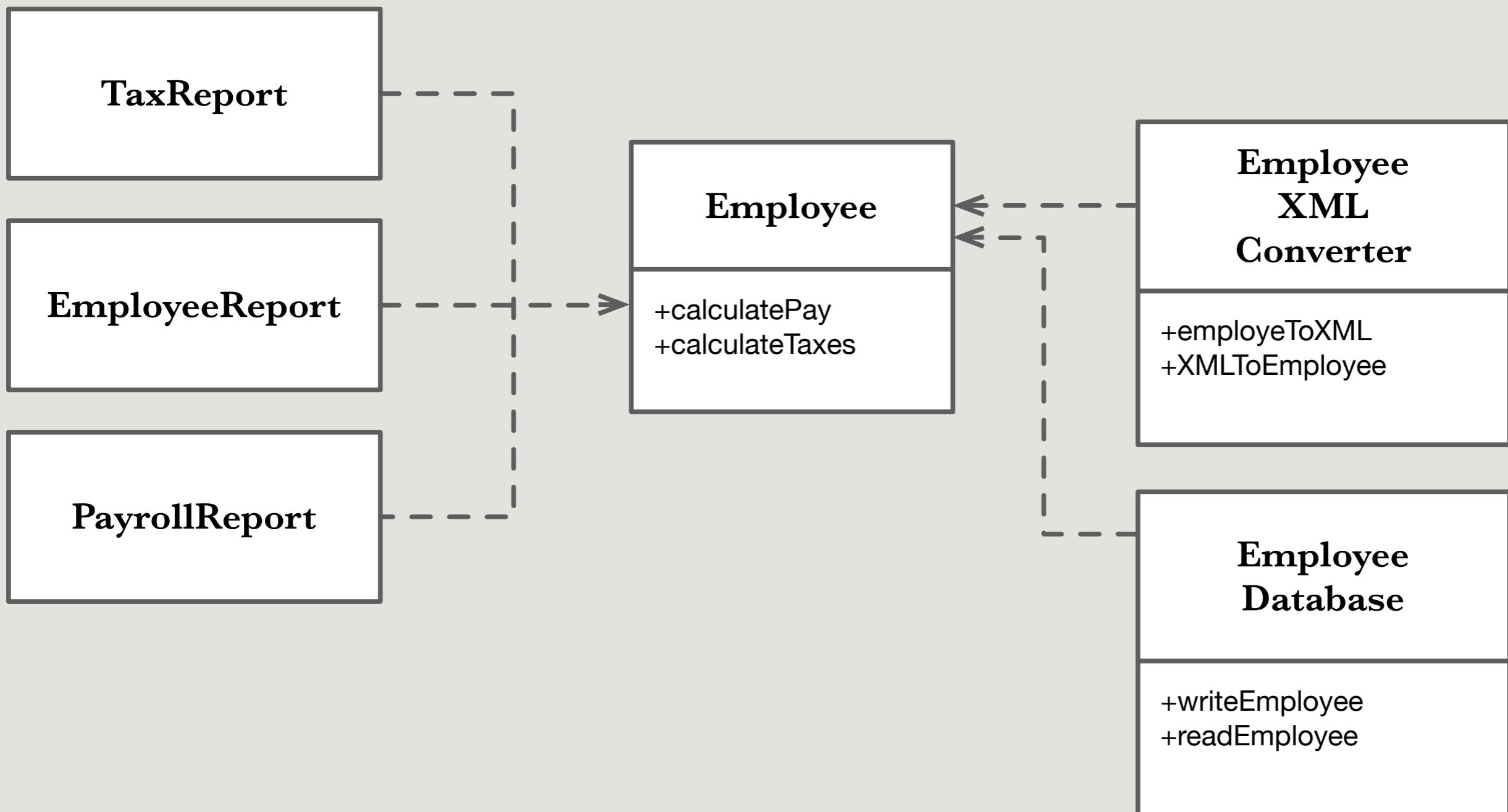
- Sabe cómo calcular el salario e impuestos del trabajador
- Cómo guardarse a sí mismo y crearse a partir de una base de datos
- Lo mismo pero escribiendo y leyendo de un fichero XML
- Sabe cómo generar e imprimir varios informes

Principio de responsabilidad única (SRP)

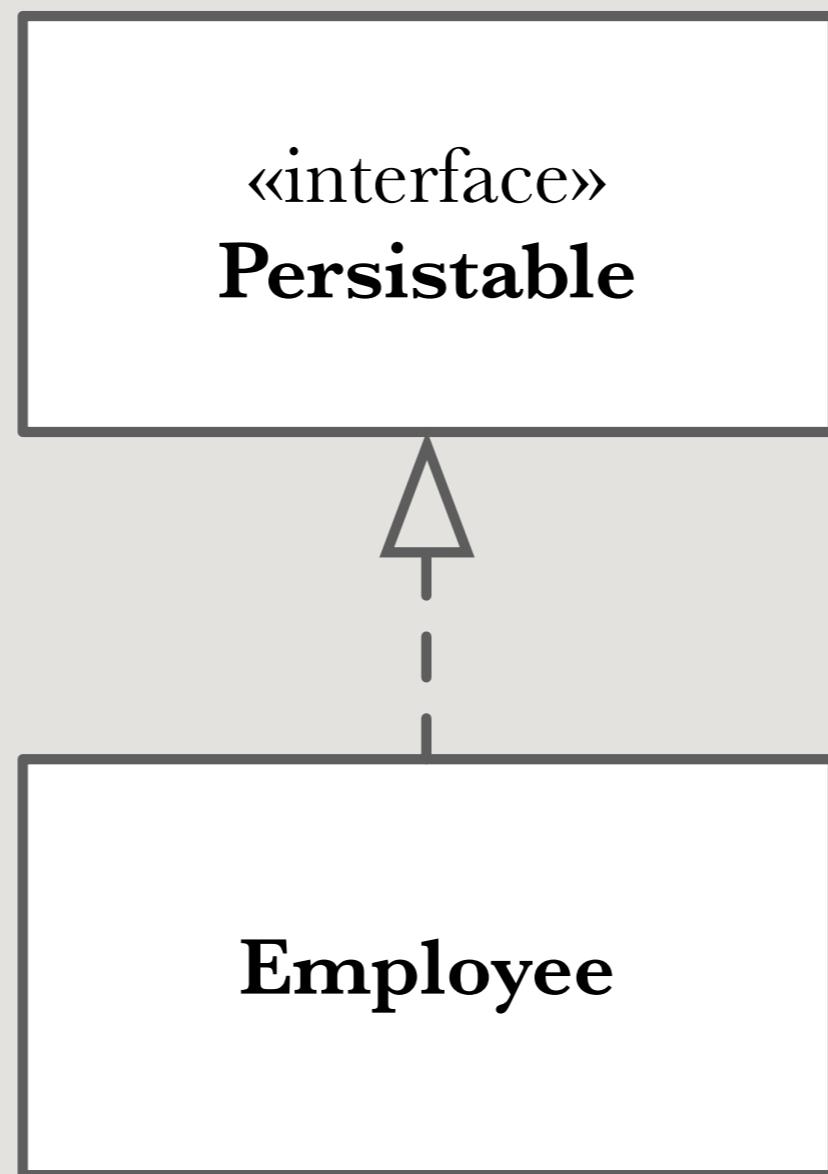
Employee

```
+calculatePay(): double  
+calculateTaxes(): double  
+writeToDisk()  
+readFromDisk()  
+createXML(): String  
+parseXML(String xml)  
+printEmployeeReport(PrintWriter)  
+printPayrollReport(PrintWriter)  
+printTaxReport(PrintWriter)
```

Principio de responsabilidad única (SRP)



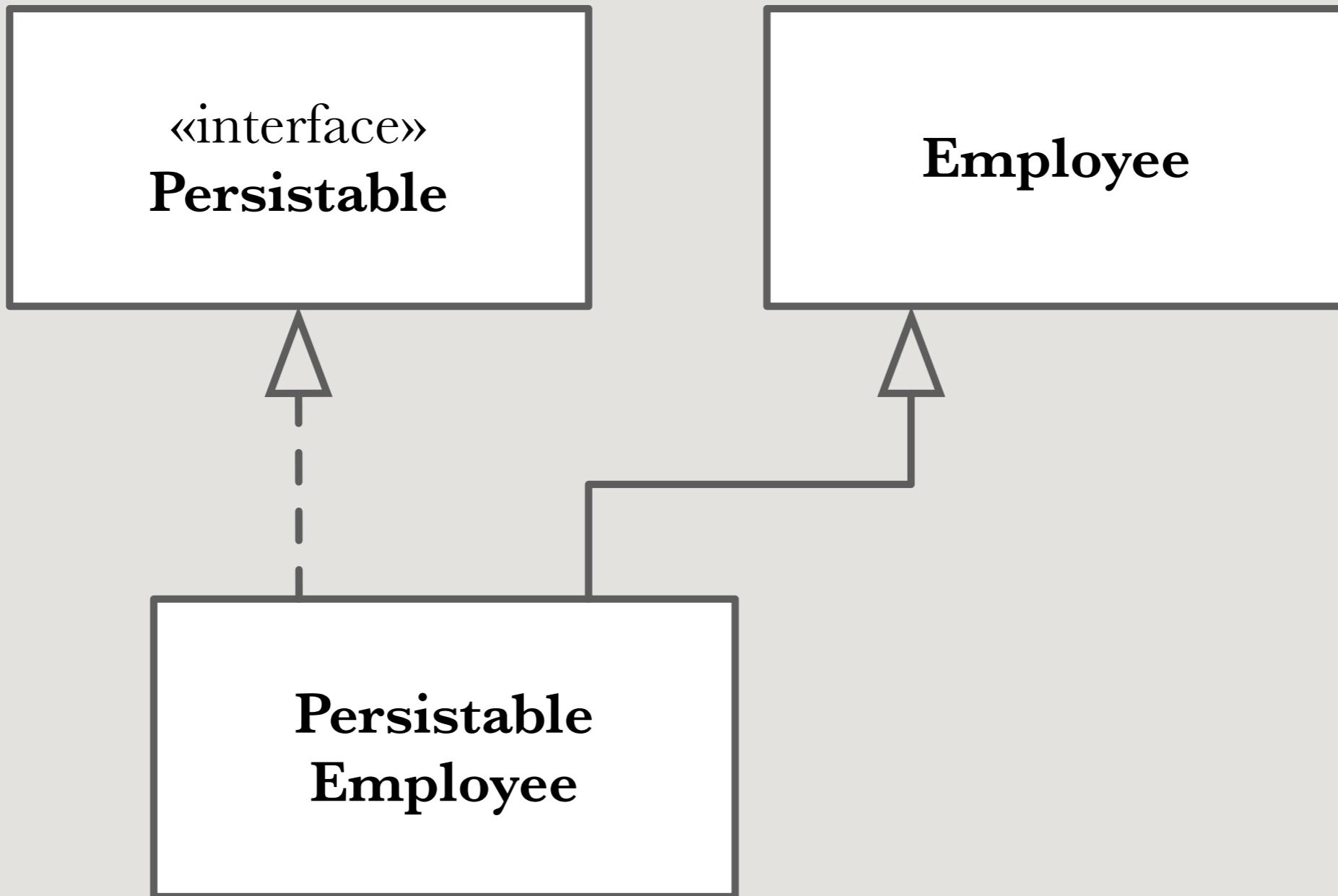
Otro ejemplo



Empleados persistentes

- En el diagrama anterior, **Employee** y **Persistable** están acoplados
 - Todos los usuarios de Employee a su vez dependen de Persistable
 - Los cambios en la interfaz de Persistable pueden afectar a todos los clientes de Employee (aunque no los necesitasen)
 - ▶ Por ejemplo, si cambia la signatura de los métodos de persistencia

Más independientes



Más independientes

- Las instancias de **PersistableEmployee** se pueden pasar al resto del sistema como empleados, sin que los clientes se percaten del acoplamiento

No siempre es tan obvio

Aunque muy fácil de entender, el principio SRP no siempre resulta sencillo de aplicar ni es tan evidente como en el primer ejemplo del empleado que hacía de todo.

Módem

```
public interface Modem
{
    void dial(String number);
    void hangUp();
    void send(char c);
    char receive();
}
```

¿Algún problema?

- Al fin y al cabo, las cuatro operaciones son claramente responsabilidades de un módem
 - Parece perfectamente razonable
- Sin embargo, recordemos que, en el contexto del SRP, una responsabilidad se define como...

«una razón para el cambio»

Dos responsabilidades

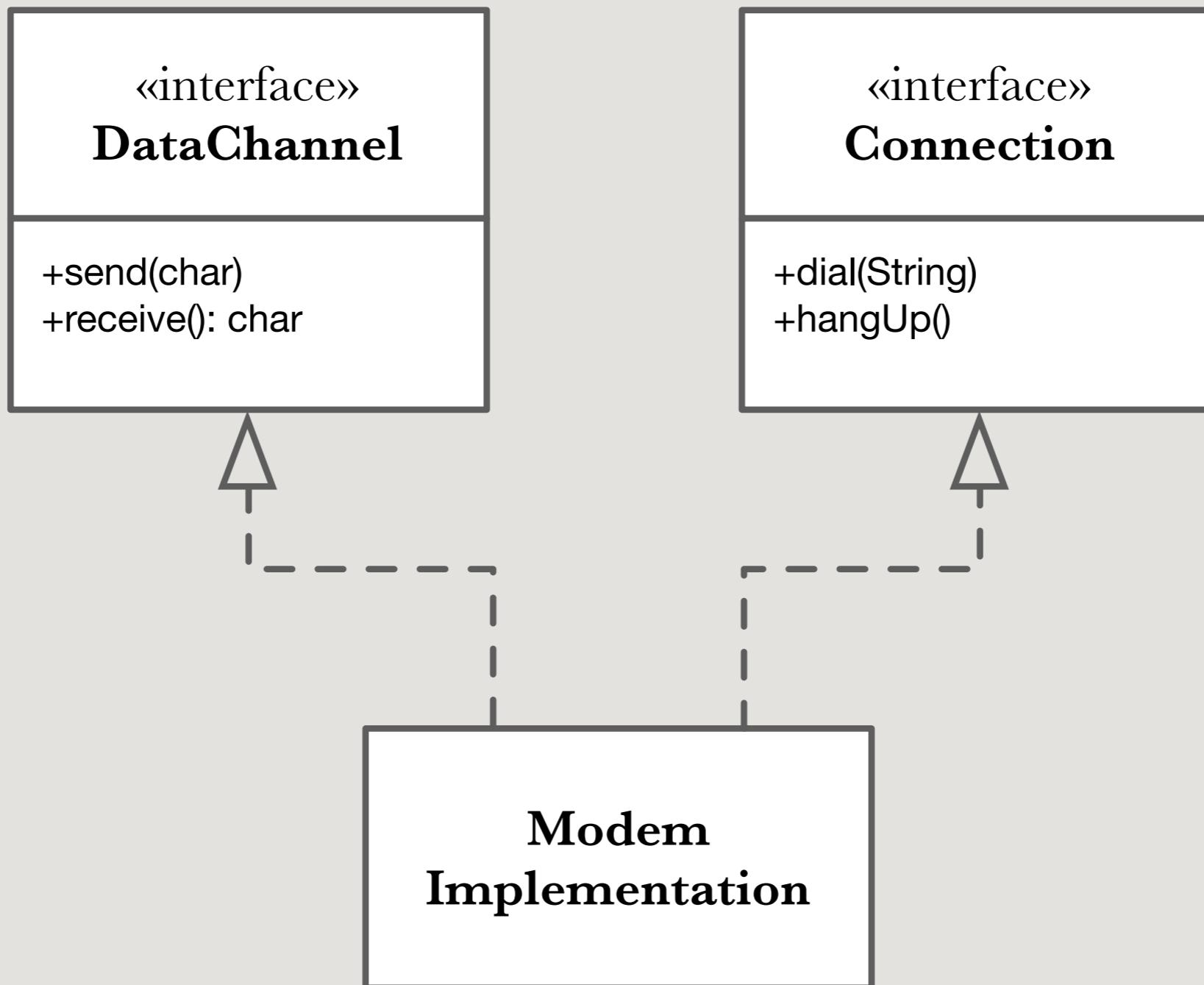
- Podríamos distinguir entre:
 - La gestión de la conexión
 - ▶ «llamar» y «colgar»
 - La comunicación en sí
 - ▶ enviar y recibir los datos
- ¿Deberían estar separadas?

Depende

¿De qué?

- De cómo se prevea que vaya a cambiar la aplicación
- Si puede haber cambios que afecten a la firma de las funciones de conexión, entonces nuestro diseño adolece de cierta rigidez

Separando las responsabilidades



Pero sin pasarse

- Claro que lo anterior sólo sería necesario en caso de que ambas responsabilidades variasen en nuestra aplicación de forma independiente
- Si no, no habría razón para separarlas
 - Sería un ejemplo de «complejidad innecesaria» (*Needless Complexity*)

No sobrediseñemos

corolario

*Un motivo para el cambio lo
es únicamente si el cambio
realmente tiene lugar*

OPEN

**SORRY WE'RE
CLOSED**

*Las clases deberían estar
abiertas para la extensión,
pero cerradas para la
modificación*

¿Qué pasaba?

```
class Pizarra
{
    private Rectangulo[] rectangulos = new Rectangulo[30];
    private int contador = 0;

    public void añadir(Rectangulo rectangulo)
    {
        rectangulos[contador++] = rectangulo;
    }

    public void dibujar()
    {
        for (int i = 0; i < contador; i++) {
            System.out.println(rectangulos[i].x1 + ", " + rectangulos[i].y1);
            System.out.println(rectangulos[i].x2 + ", " + rectangulos[i].y2);
        }
    }
}
```

Nuevas figuras

- Cada vez que aparezca una nueva figura habrá que modificar el método de dibujar
 - El típico if anidado o switch
- En la vida real, esa lógica condicional se repetirá en todos los sitios de la aplicación que traten con figuras
 - Para moverlas, cambiar su tamaño, duplicarlas...
- El impacto es enorme

```
class Pizarra
{
    private Figura[] figuras = new Figura[30];
    private int contador = 0;

    public void añadirFigura(Figura figura)
    {
        figuras[contador++] = figura;
    }

    public void dibujar()
    {
        for (int i = 0; i < contador; i++) {
            figuras[i].dibujar();
        }
    }
}
```

Ahora nuestro programa satisface el OCP

*Podemos cambiar el programa
añadiendo código nuevo (en vez
de modificando el existente)*

(Hasta cierto punto)

- ¿Y si ahora resulta que los círculos deben ser dibujados antes que los rectángulos?
- Es imposible preverlo todo
 - Siempre habrá algún cambio para el que nuestro diseño no estará «cerrado»
- Decidir qué cambios serán más probables es cuestión de experiencia, intuición... y sentido común

¿Qué haríamos en ese caso?

Cuando ocurre...

- Que sólo nos pase una vez
- Intentaremos blindarnos ante cambios similares
 - En este caso, diferentes formas de ordenar las figuras antes de ser dibujadas

LSP: Principio de sustitución de Liskov

*Los subtipos deben poder
sustituir a sus tipos base*

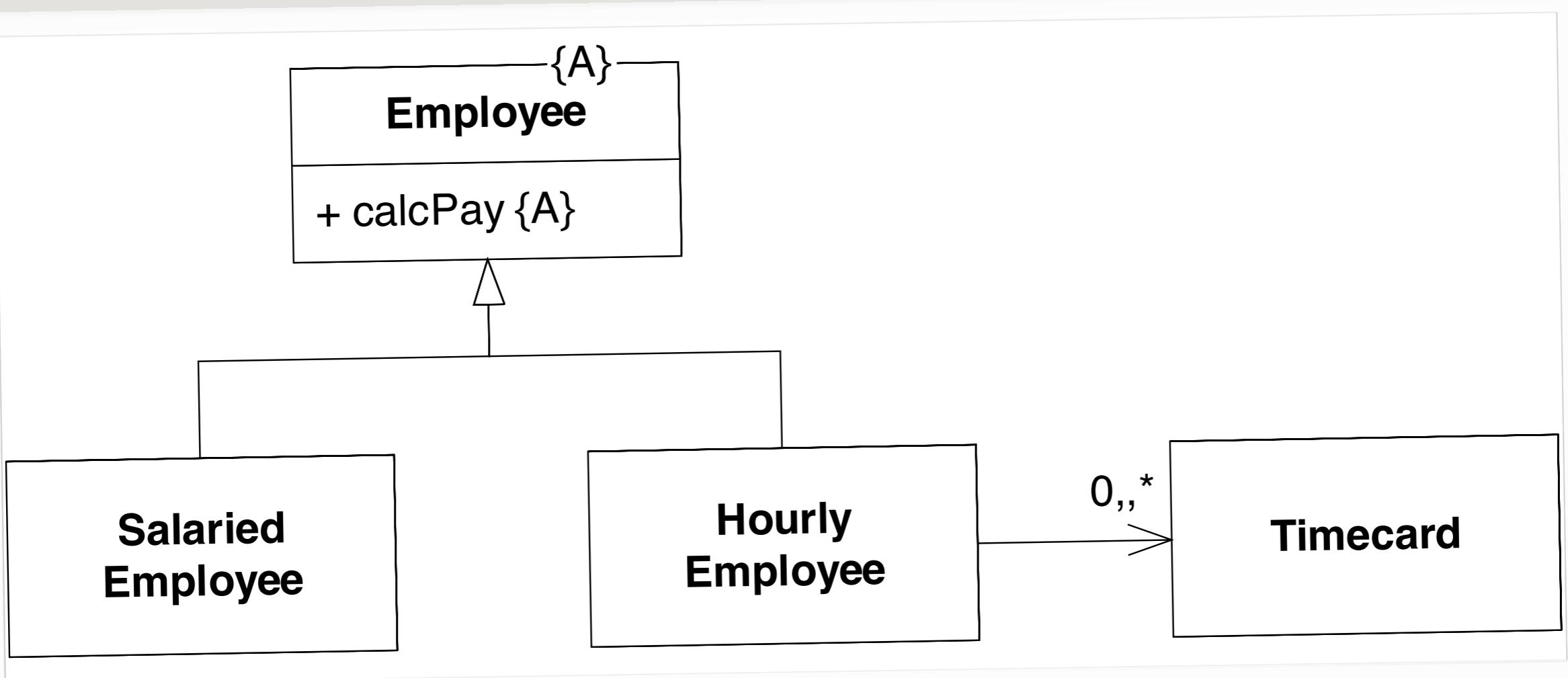
LSP

- ¿Alguna vez habéis visto código con un montón de expresiones **instanceof** en las cláusulas de un if anidado?
- Normalmente es el resultado de una violación del LSP
 - Que a su vez implica romper también el OCP

LSP

- Los usuarios de los tipos base no deberían hacer nada especial para tratar a los tipos derivados
 - Ni `instanceof` ni `downcasts`
- De hecho, no deberían ser conscientes siquiera de la existencia de esos subtipos

Ejemplo



¿Qué ocurre si aparece un empleado voluntario
(que no cobra)?

Una posibilidad

```
public class VolunteerEmployee extends Employee
{
    public double calcPay()
    {
        return 0;
    }
}
```

¿Qué opináis?

Una posibilidad

- Devolver cero parece implicar que, después de todo, calcular el salario de un voluntario tiene sentido (cuando no es así)

Otra

```
public class VolunteerEmployee extends Employee
{
    public double calcPay()
    {
        throw new UnpayableEmployeeException();
    }
}
```

¿Y ahora?

Otra

- Al fin y al cabo, ¿no se hicieron las excepciones para señalar situaciones ilegales como ésta?
- Pero ahora todas las llamadas a **calcPay** pueden lanzar una excepción
 - Así que hay todos los clientes tienen que capturarla o declararla
 - ¡Un requisito de una clase derivada ha afectado a todos los clientes de la clase base!

Al final...

- Seguramente acabaríamos escribiendo código como éste:

```
for (int i = 0; i < employees.size(); i++) {  
    Employee e = (Employee) employees.elementAt(i);  
    if (!(e instanceof VolunteerEmployee)) {  
        totalPay += e.calcPay();  
    }  
}
```

Que naturalmente es...

¡una chapulza!

¿Qué ha pasado?

- Que hemos violado el LSP
- Lo sabemos cada vez que:
 - Intentamos hacer que sea ilegal invocar un método en un tipo derivado
 - O si le damos una implementación vacía (o similar)

¿Y la solución?

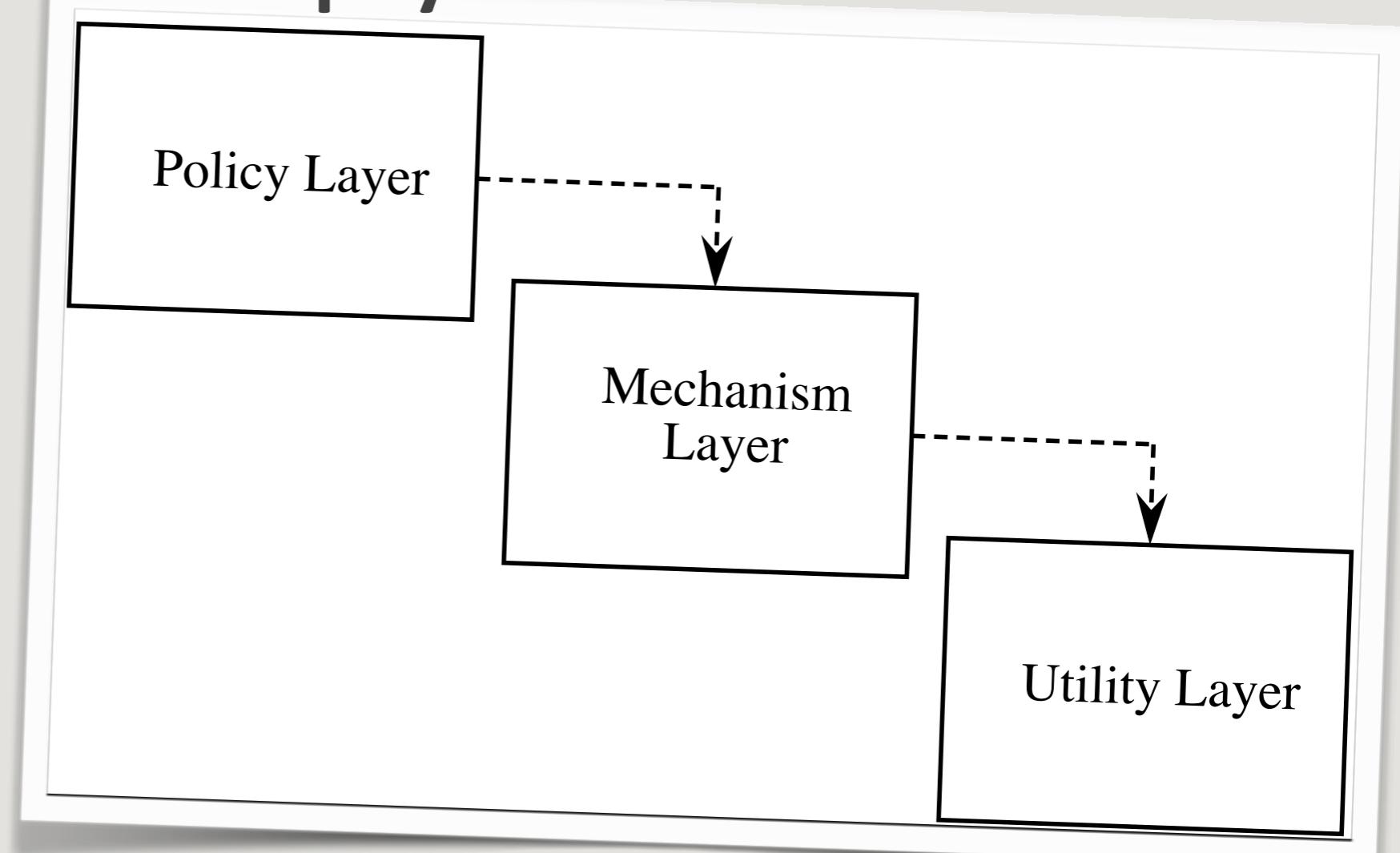
- En este caso, los voluntarios no son empleados
- No tiene sentido llamar a calcular salario sobre ellos
- Así que no deben heredar de empleado, no son un subtipo en este contexto
 - Y por tanto no deberían ser pasados a métodos que necesiten llamar a calcular salario

DIP: El principio de inversión de dependencias

- a. Los módulos de alto nivel no deben depender de los de bajo nivel; ambos deben depender de abstracciones*
- b. Las abstracciones no deben depender de los detalles, sino éstos de las abstracciones*

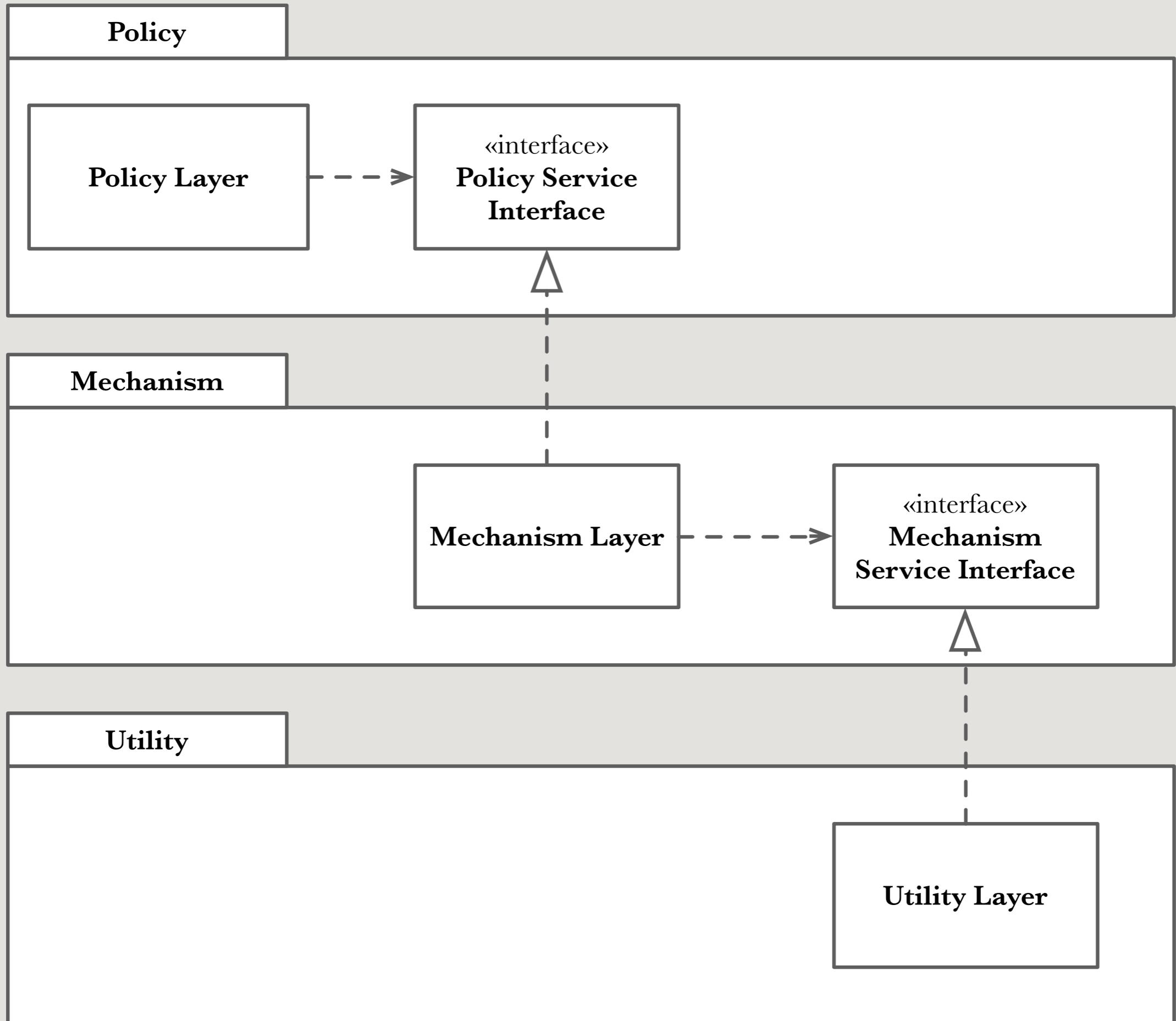
¿Por qué «inversión»?

- Porque los métodos estructurados tradicionales promovían que los módulos de alto nivel se apoyasen en los de nivel inferior

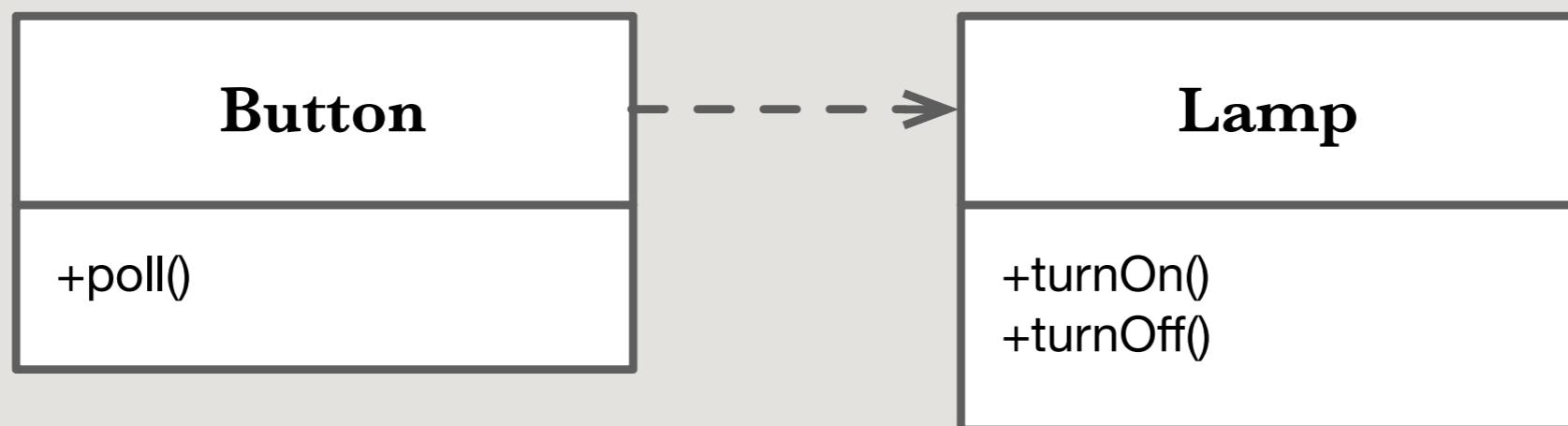


¿Por qué «inversión»?

- La dependencia en un programa orientado a objetos bien diseñado suele ser a la inversa

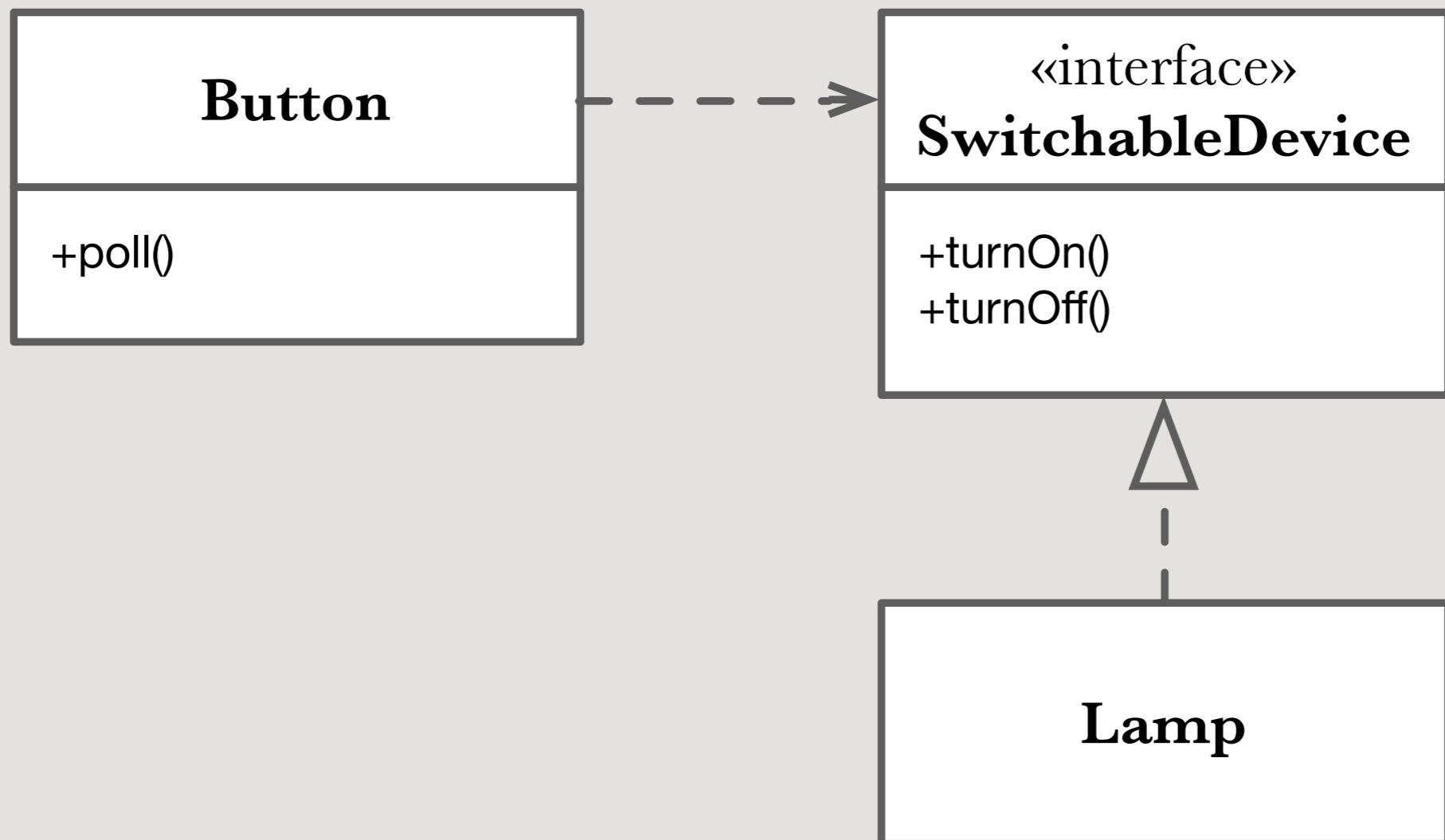


Ejemplo



```
public class Button
{
    private Lamp lamp;
    public void poll()
    {
        if /* some condition */
            lamp.turnOn();
    }
}
```

Aplicando el DIP



¿Qué hemos logrado?

- Ahora el botón puede controlar cualquier dispositivo que implemente la interfaz proporcionada por él
 - ¡Incluso aquéllos que aún no han sido inventados!



Principio de Hollywood