

SEMINARIO

2

Principios solid

Diseño del Software

Grado en Ingeniería Informática del Software

Curso 2024-2025



Uncle Bob

SRP Single Responsibility Principle

OCP Open-Closed Principle

LSP Liskov Substitution Principle

ISP Interface Segregation Principle

DIP Dependency Inversion Principle



SRP

Principio de responsabilidad
única

*Una clase debería tener un
único motivo para cambiar.*

Employee

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

¿Qué podemos decir de Empleado?

Que hace demasiadas cosas.

Calcula el salario e
impuestos del trabajador

Sabe cómo guardarse a sí
mismo y cargarse de una
base de datos

Lo mismo pero leyendo y
escribiendo en un fichero XML

Genera e imprime varios
informes

Si cambiamos de SAX a JDOM, tenemos que cambiar Employee.

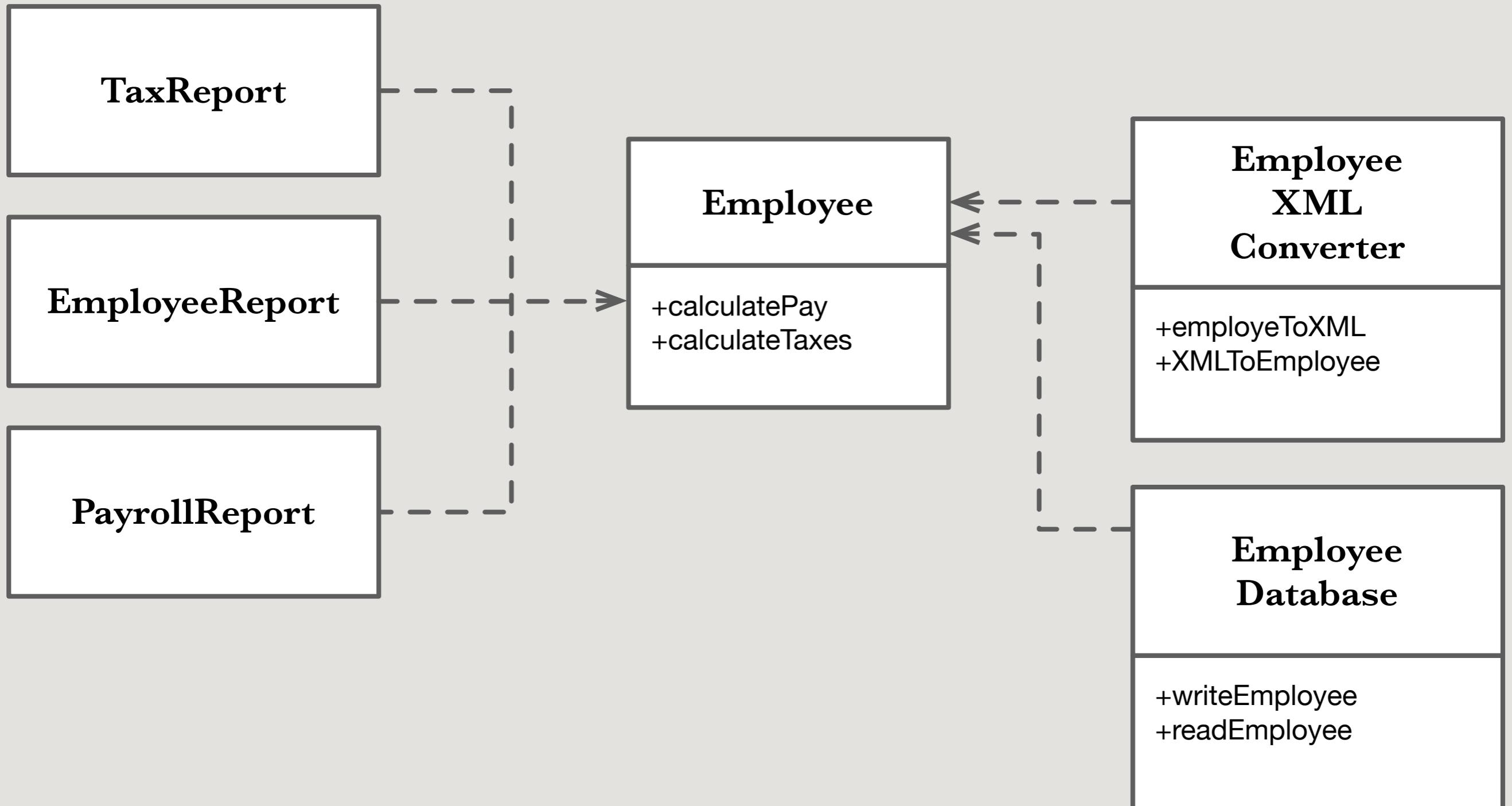
Si cambiamos de Access a Oracle, lo mismo.

...

El diseño está acoplado.

Employee

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



Otro ejemplo

«interface»
Persistable

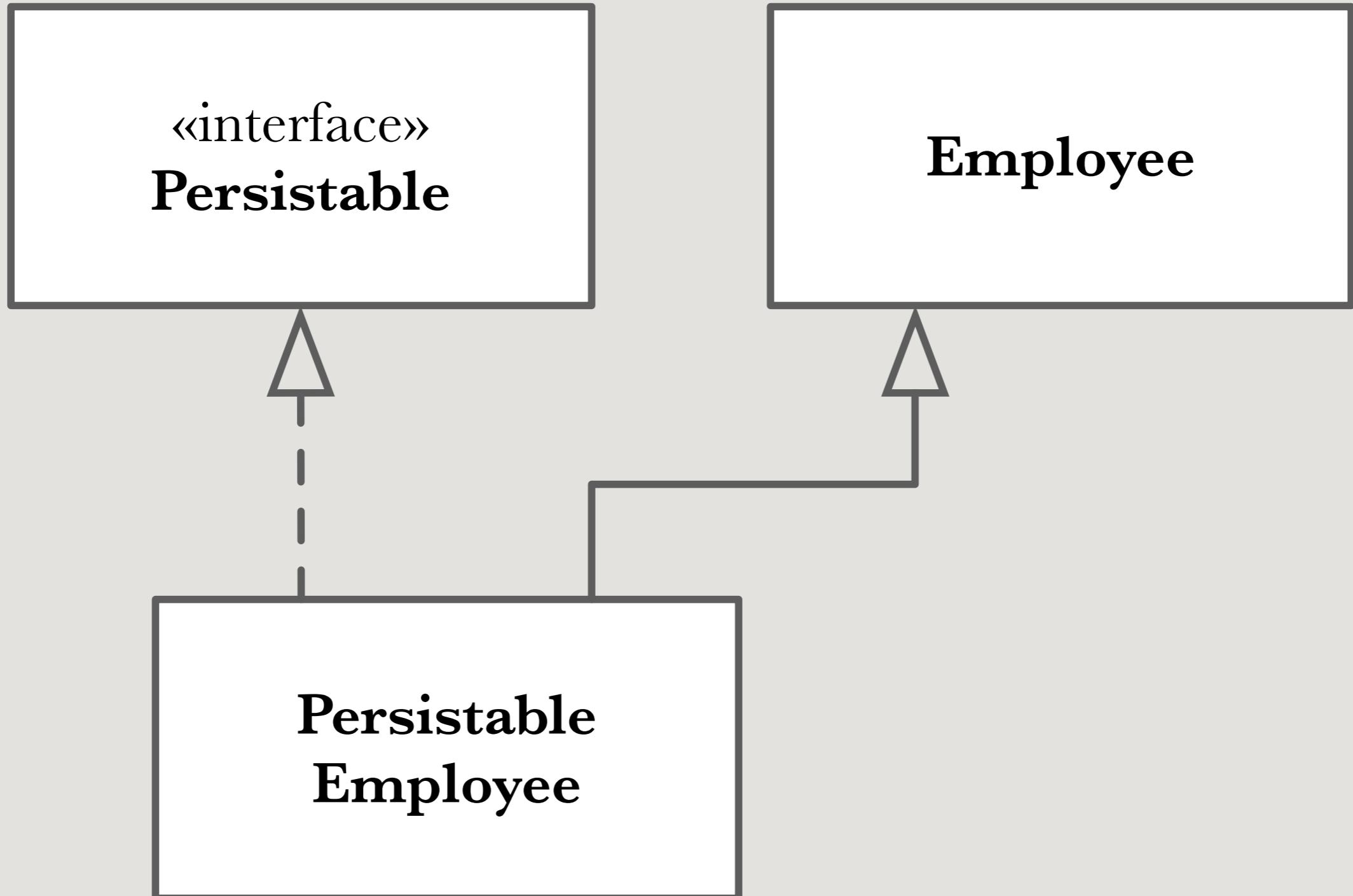


Employee

Todos los usuarios de Employee
dependen, indirectamente, de
Persistable.

La dependencia no es muy grande, pero está ahí.

Los cambios en la interfaz
podrían afectar a todos los
clientes de la clase.



Los objetos de **PersistableEmployee** se podrían pasar al resto del sistema como empleados normales, sin que el resto de clientes se percatasen del acoplamiento.

No siempre es obvio

Aunque 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.



Modem.java

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

¿Algún problema?

¿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.

Podríamos distinguir entre:

La gestión de la conexión

Lamar y colgar

La comunicación en sí

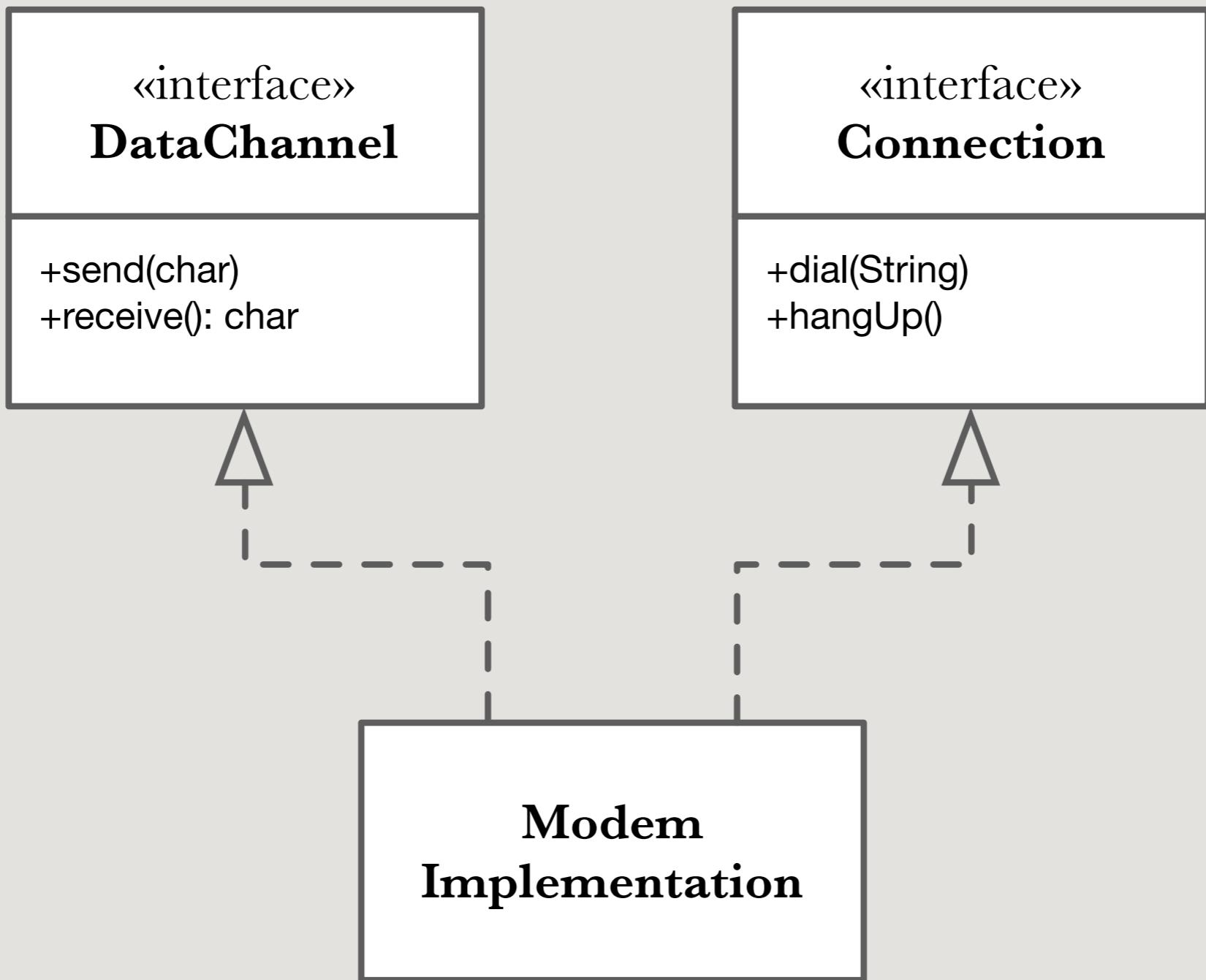
Enviar y recibir datos

¿Deberían estar separadas?

Depende.

¿Puede haber cambios que afecten a las operaciones de conexión y envío de datos por separado?

Si es así, nuestro diseño adolece de cierta rigidez.



Separando ambas responsabilidades

Pero...

¡¿No las seguimos teniendo
juntas en la clase
ModelImplementation?!

Sí, pero a veces puede ser necesario.

En ese caso, ¿tendría sentido lo anterior?

Sí...

Siempre y cuando los clientes solo dependieran de las interfaces DataChannel y Connection, respectivamente.

Nadie, salvo el main (o quien cree dicho objeto) debería conocer siquiera la existencia de ModelImplementation.

Otra posibilidad...

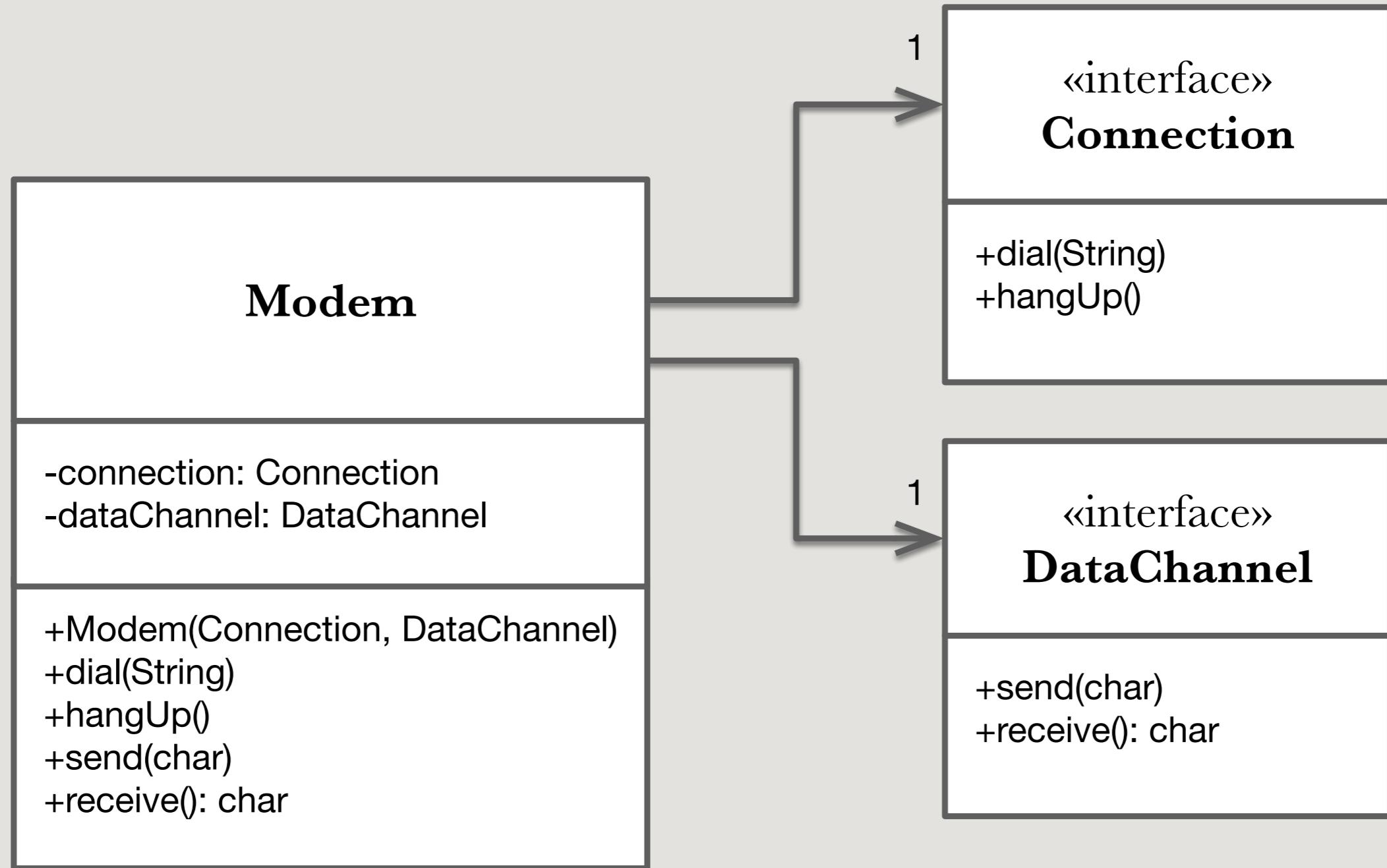
¿Y si quisieramos tener ambas responsabilidades juntas pero cuya implementación pudiera variar por separado?

Pistas...

*Encapsular y aislar
el concepto que varía.*

Pistas...

*Favorecer la
composición frente a
la herencia.*



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.*



```
class Dibujo {  
  
    private Rectángulo[] rectángulos = new Rectángulo[30];  
    private int contador = 0;  
  
    public void añadir(Rectángulo rectángulo) {  
        rectángulos[contador++] = rectángulo;  
    }  
  
    public void dibujar() {  
        for (int i = 0; i < contador; i++) {  
            System.out.println(rectángulos[i].x1 + ", " + rectángulos[i].y1);  
            System.out.println(rectángulos[i].x2 + ", " + rectángulos[i].y2);  
        }  
    }  
}
```

¿Qué pasaba?

Dependiente de la representación interna del rectángulo

Habrá que cambiar el método dibujar cada vez que cambie aquél.

Si aparecen nuevas figuras

Habrá que cambiar el método dibujar para añadir la correspondiente rama condicional con un nuevo if-then-else o un case.

Si cambia la forma de dibujarse
una figura

Habrá que cambiar una de dichas ramas
condicionales.

En un caso real, dicha lógica condicional se repetiría en todos los lugares de la aplicación que traten con figuras.

Para moverlas, cambiar su tamaño...

El impacto es enorme.



```
class Dibujo {  
  
    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).*

¿Seguro?

¿Y si ahora queremos que los círculos
se dibujen 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.

Por cierto...

¿Qué podríamos
hacer en ese caso?

Cuando ocurra un cambio para el que
nuestro diseño no estaba preparado...

Que solo ocurra una vez.

No nos limitaremos a intentar meterlo como sea en el diseño actual.

Habrá que rediseñar.

De manera, ahora sí, que quede cerrado para otros cambios similares.

En este caso, por ejemplo, ¿tal vez un mecanismo para ordenar las figuras por distintos criterios antes de ser dibujadas?

LSP

Principio de sustitución de Liskov



Barbara Liskov

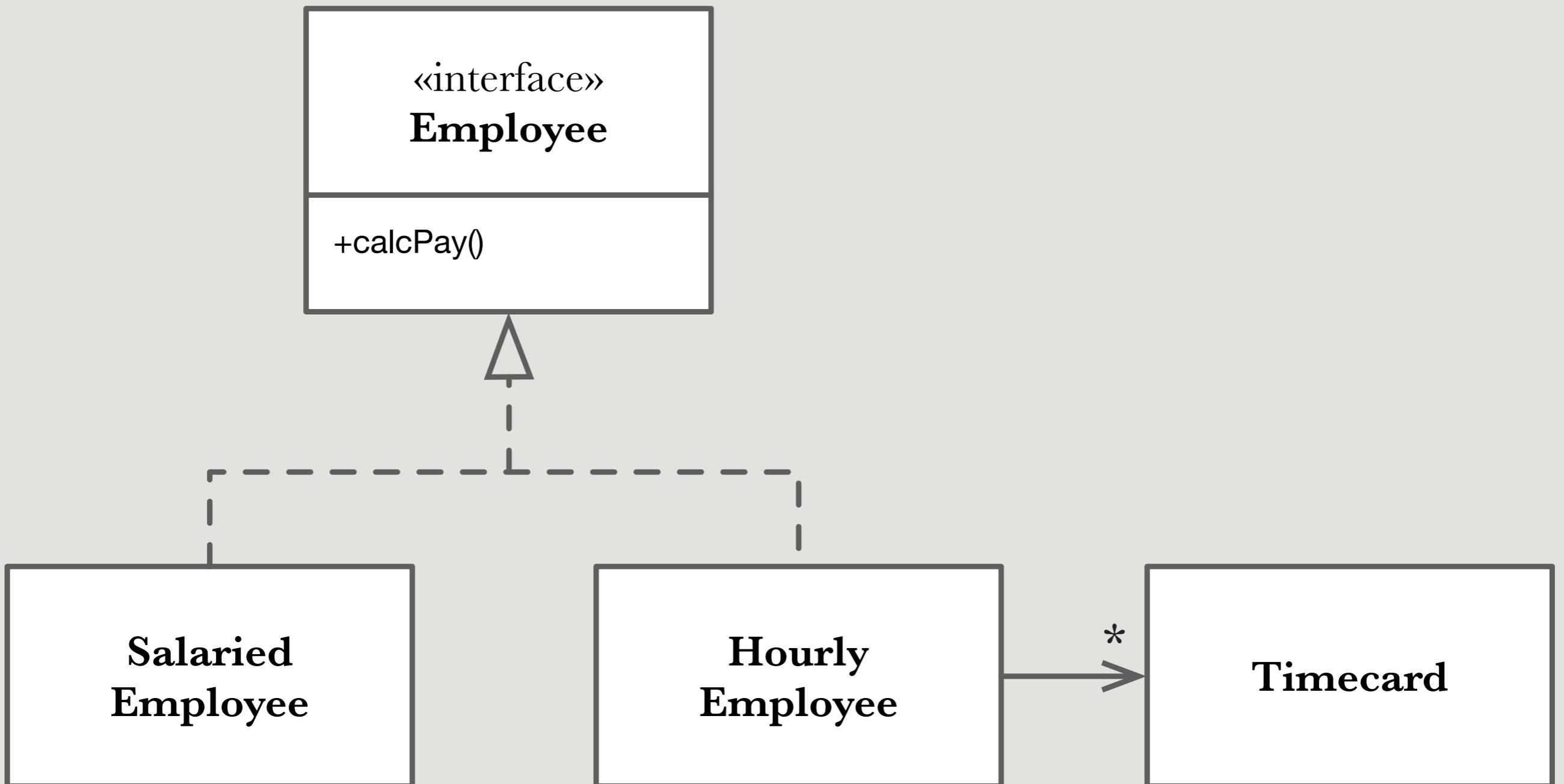
*Los subtipos deben poder
sustituir a sus tipos base.*

¿Algunas veces habéis visto código con
un montón de expresiones instanceof
en las cláusulas de un if anidado?

(No disimuléis).

Normalmente es el resultado de una violación del LSP.

Que a su vez implica romper también el OCP.



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



VolunterEmployee.java

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

Una posibilidad

Devolver cero parece indicar que, después de todo, calcular el salario de un becario tiene sentido.

Cuando probablemente no sea así.

Cobrar cero, ¿es lo mismo que no cobrar?

En determinados contextos, probablemente no.



VolunterEmployee.java

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

Otra

Al fin y al cabo, no se hicieron las excepciones para señalar situaciones ilegales como esta?

Pero ahora todas las llamadas a calcPay pueden lanzar una excepción.

¡Un requisito de la clase derivada ha afectado a todos los clientes de la clase base!

Por cierto, en tal caso... ¿de qué tipo debería ser la excepción?



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

Al final seguramente acabaríamos haciendo cosas como esta.

Que, naturalmente, es...

¡Una chapuz!

Los usuarios de los tipos base no deberían hacer nada especial para tratar a los tipos derivados.

Ni ‘instanceofs’ ni ‘downcasts’.

De hecho, no deberían ser conscientes siquiera de la existencia de tales subtipos.

The screenshot shows a web browser window displaying the Java API documentation. The URL is docs.oracle.com. The top navigation bar includes links for OVERVIEW, MODULE, PACKAGE, CLASS (which is highlighted in orange), USE, TREE, PREVIEW, NEW, DEPRECATED, INDEX, and HELP. On the right, it says Java SE 22 & JDK 22. Below the navigation bar, there are links for SUMMARY: NESTED | FIELD | CONSTR | METHOD and DETAIL: FIELD | CONSTR | METHOD. A search bar with the placeholder "Search" and a magnifying glass icon is also present. The main content area starts with "Module java.base" and "Package java.util". The title "Interface Iterator<E>" is displayed in a large, bold font. Below the title, there are sections for "Type Parameters:", "All Known Subinterfaces:", and "All Known Implementing Classes:". The "Type Parameters:" section defines E as "the type of elements returned by this iterator". The "All Known Subinterfaces:" section lists EventIterator, ListIterator<E>, PrimitiveIterator<T, T_CONS>, PrimitiveIterator.OfDouble, PrimitiveIterator.OfInt, PrimitiveIterator.OfLong, XMLEventReader. The "All Known Implementing Classes:" section lists BeanContextSupport.BCSIterator, EventReaderDelegate, Scanner. A horizontal line separates this from the next section. The "public interface Iterator<E>" section describes the interface as an iterator over a collection, noting its place in the Java Collections Framework and how it differs from enumerations. It lists two bullet points: "Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics." and "Method names have been improved." Below this, it states that the interface is a member of the Java Collections Framework. An "API Note" section mentions that an Enumeration can be converted into an Iterator using the Enumeration.asIterator() method. Finally, a "Since" section indicates the interface was introduced in Java SE 22.

Module [java.base](#)

Package [java.util](#)

Interface Iterator<E>

Type Parameters:

E - the type of elements returned by this iterator

All Known Subinterfaces:

[EventIterator](#), [ListIterator<E>](#), [PrimitiveIterator<T, T_CONS>](#), [PrimitiveIterator.OfDouble](#), [PrimitiveIterator.OfInt](#), [PrimitiveIterator.OfLong](#), [XMLEventReader](#)

All Known Implementing Classes:

[BeanContextSupport.BCSIterator](#), [EventReaderDelegate](#), [Scanner](#)

public interface Iterator<E>

An iterator over a collection. Iterator takes the place of [Enumeration](#) in the Java Collections Framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the [Java Collections Framework](#).

API Note:

An [Enumeration](#) can be converted into an [Iterator](#) by using the `Enumeration.asIterator()` method.

Since:

OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP Java SE 22 & JDK 22

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD SEARCH Search X

Since: 1.2

See Also: Collection, ListIterator, Iterable

Method Summary

All Methods	Instance Methods	Abstract Methods	Default Methods
Modifier and Type	Method	Description	
default void	<code>forEachRemaining(Consumer<? super E> action)</code>	Performs the given action for each remaining element until all elements have been processed or the action throws an exception.	
boolean	<code>hasNext()</code>	Returns true if the iteration has more elements.	
E	<code>next()</code>	Returns the next element in the iteration.	
default void	<code>remove()</code>	Removes from the underlying collection the last element returned by this iterator (optional operation).	

Method Details

`hasNext()`

OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP Java SE 22 & JDK 22

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD SEARCH Search X

remove

```
default void remove()
```

Removes from the underlying collection the last element returned by this iterator (optional operation). This method can be called only once per call to [next\(\)](#).

The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method, unless an overriding class has specified a concurrent modification policy.

The behavior of an iterator is unspecified if this method is called after a call to the [forEachRemaining](#) method.

Implementation Requirements:

The default implementation throws an instance of [UnsupportedOperationException](#) and performs no other action.

Throws:

[UnsupportedOperationException](#) - if the `remove` operation is not supported by this iterator
[IllegalStateException](#) - if the `next` method has not yet been called, or the `remove` method has already been called after the last call to the `next` method

forEachRemaining

```
default void forEachRemaining(Consumer<? super E> action)
```

Performs the given action for each remaining element until all elements have been processed or the action throws an exception. Actions are performed in the order of iteration, if that order is specified. Exceptions thrown by the action are relayed to the caller.

The behavior of an iterator is unspecified if the action modifies the collection in any way (even by calling the `remove` method).

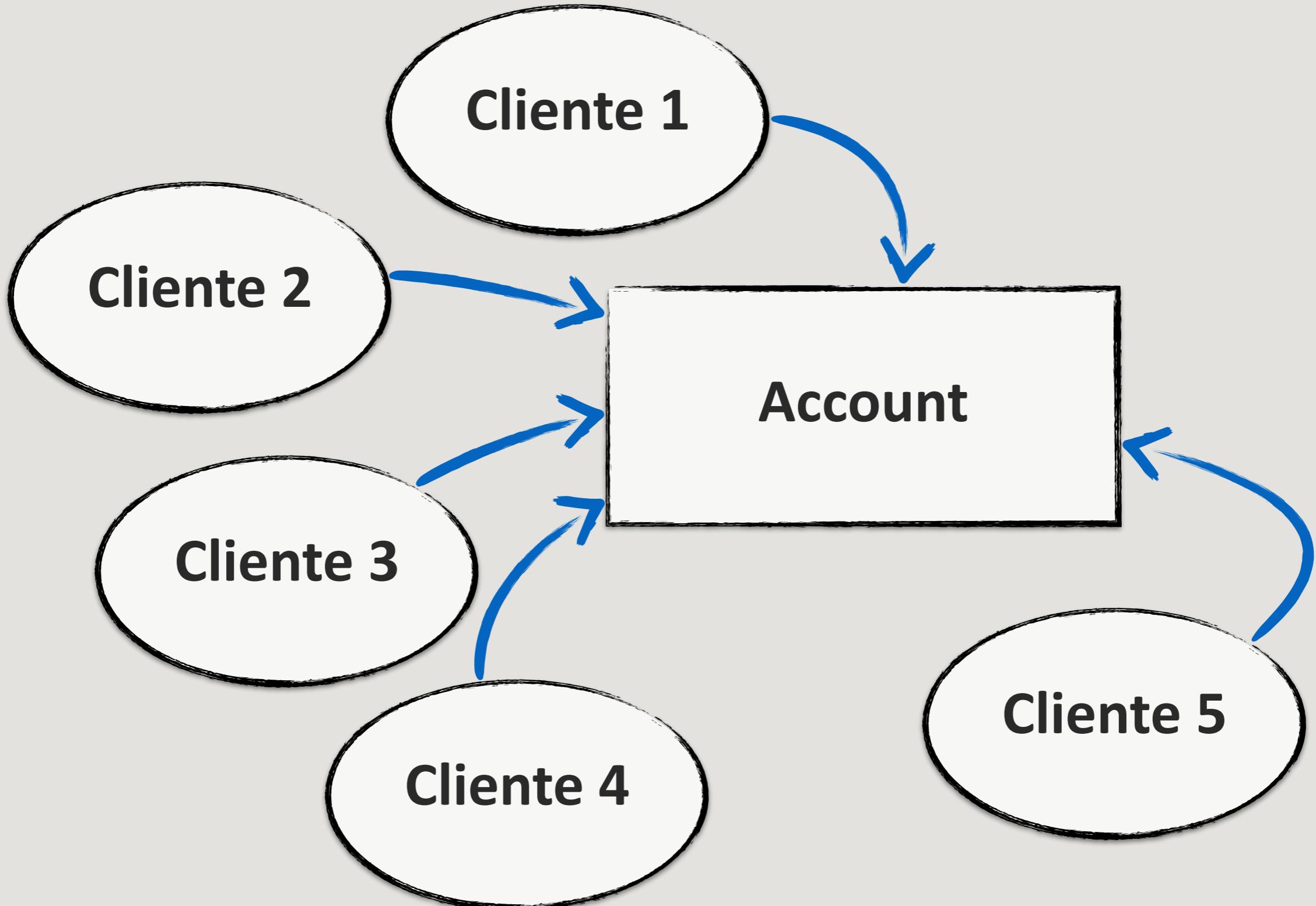
ISP

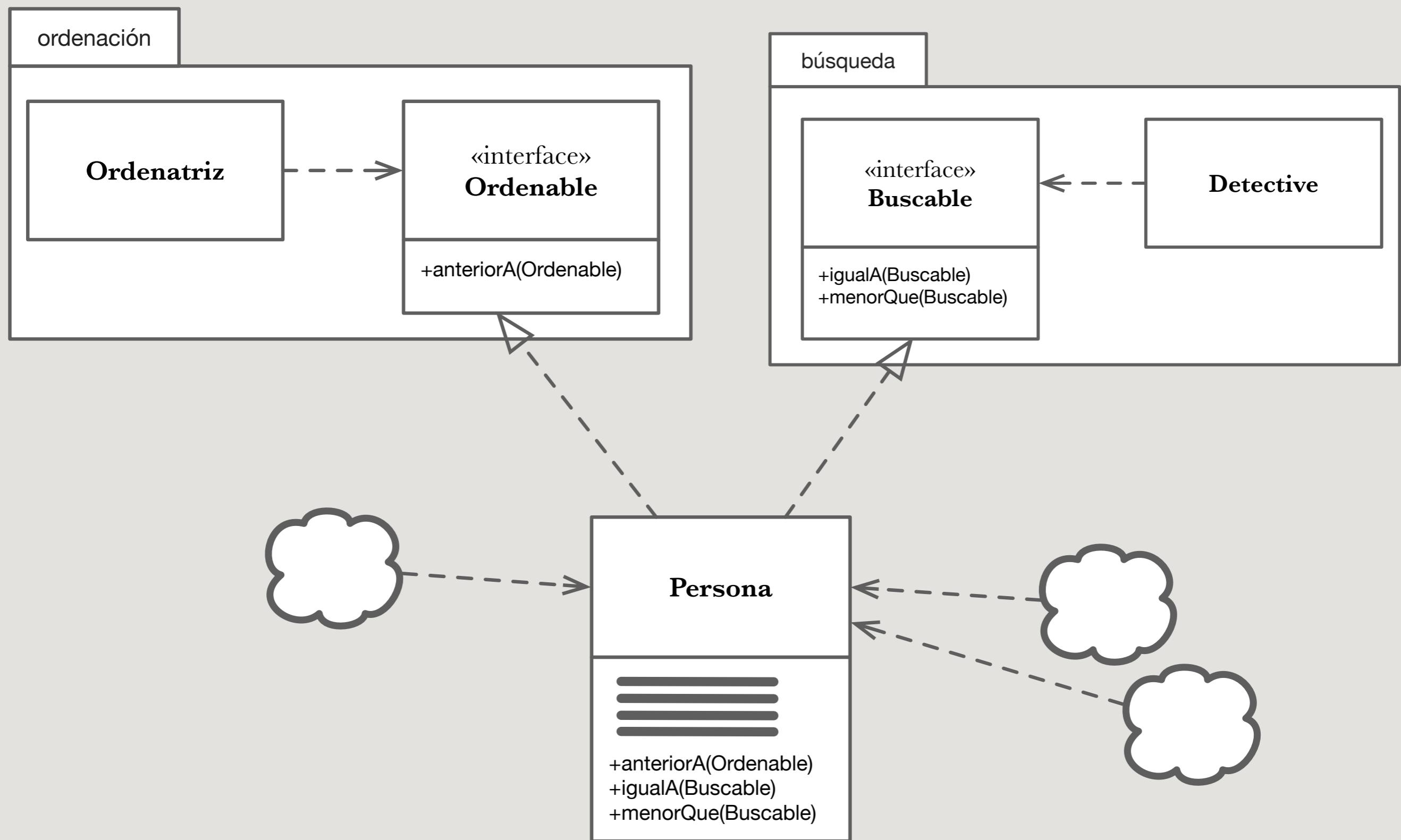
Principio de segregación de
interfaces

*Es mejor muchas interfaces
específicas para cada cliente
que una sola interfaz de
propósito general.*

**Los clientes no deberían
depender de métodos que
no usan.**

En clase de teoría lo explicábamos con el siguiente ejemplo.



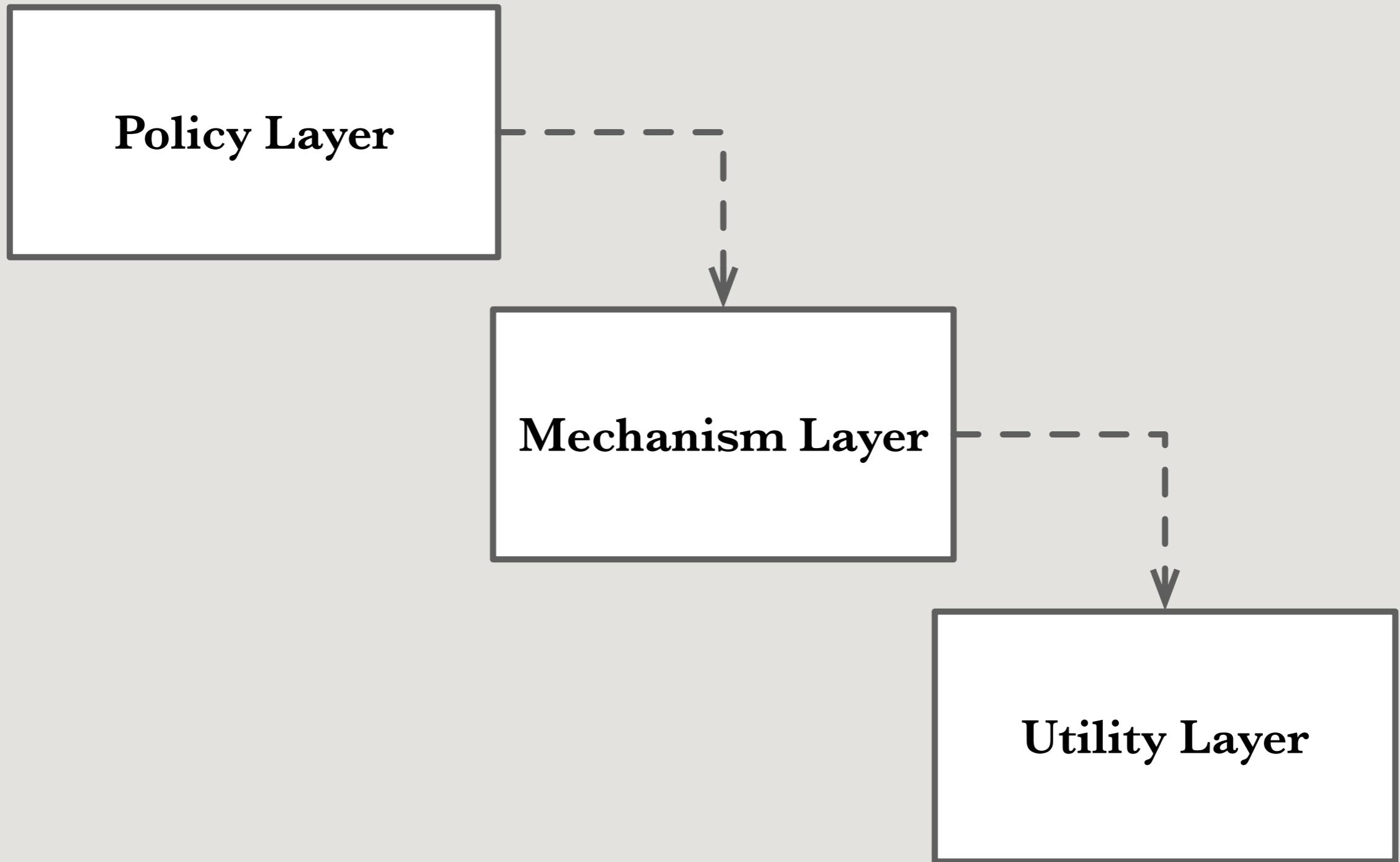


DON'T CALL US

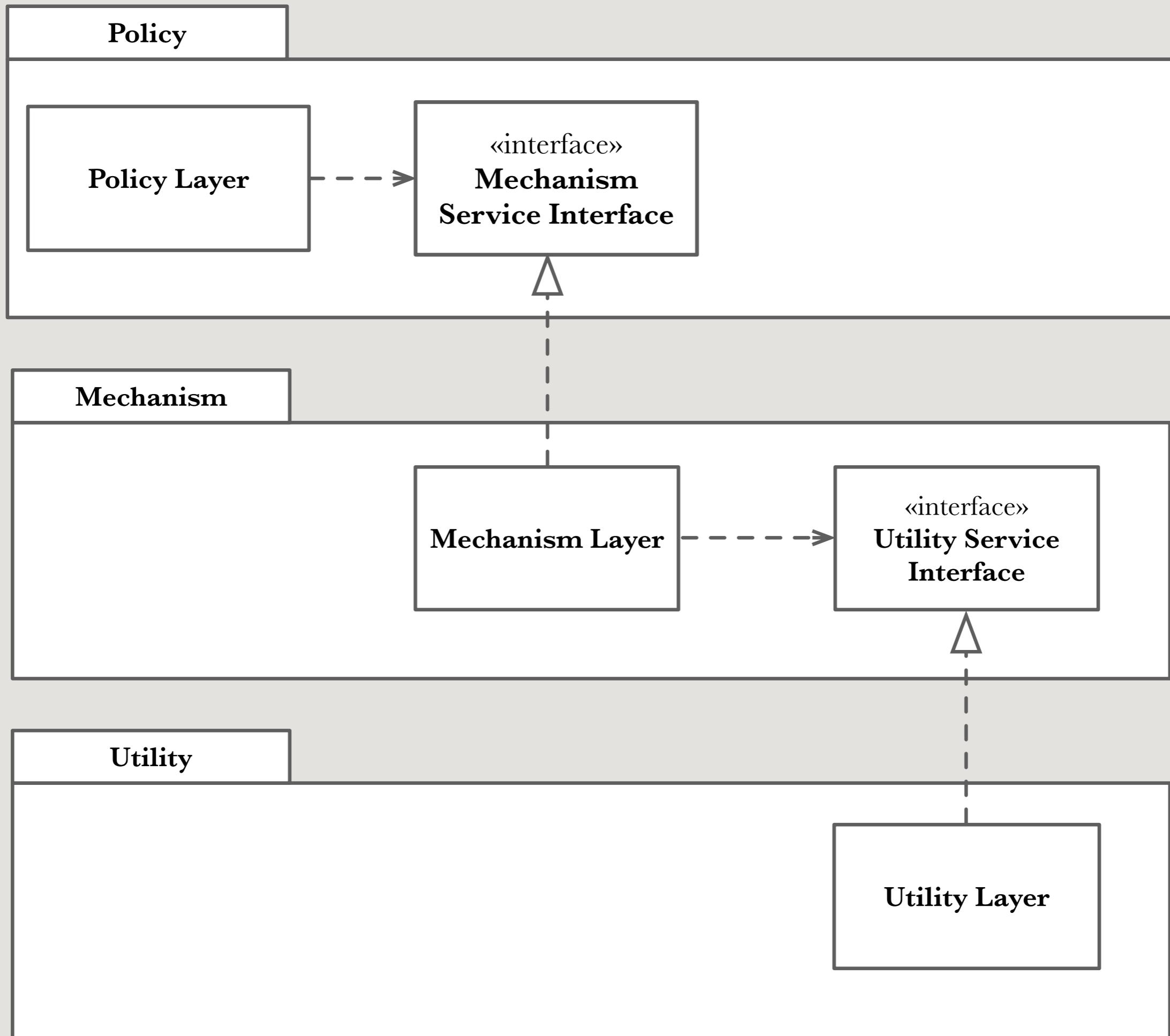
WE'LL CALL YOU

DIP: 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 estos de las abstracciones.*



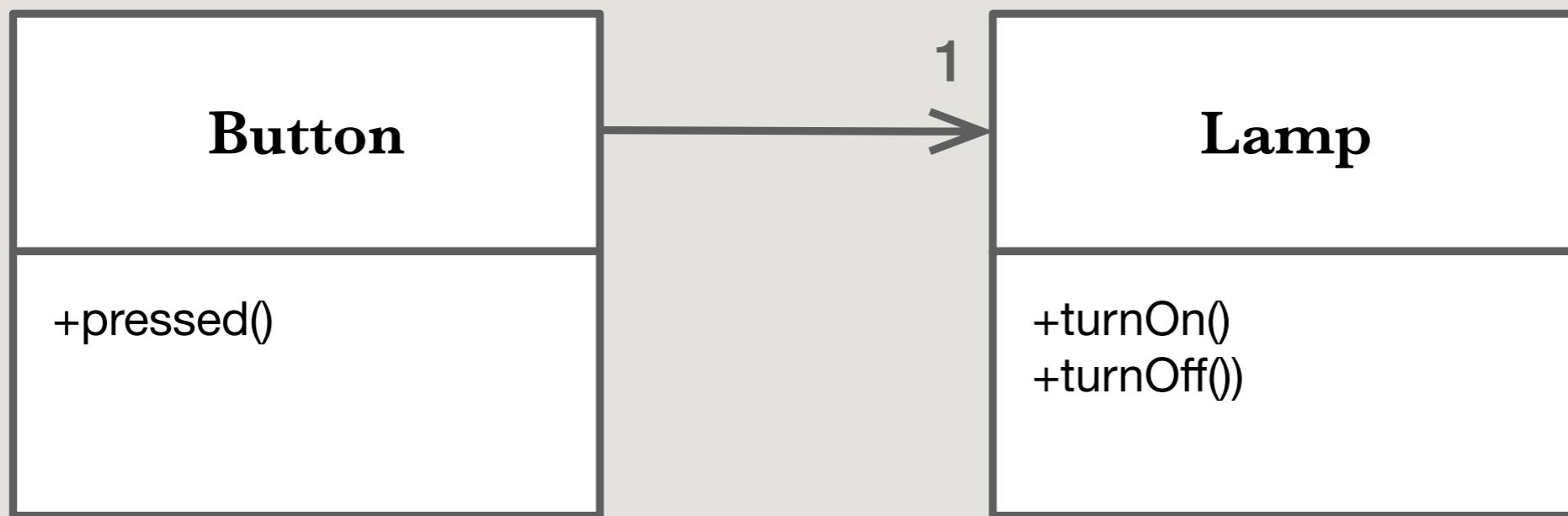
Métodos estructurados

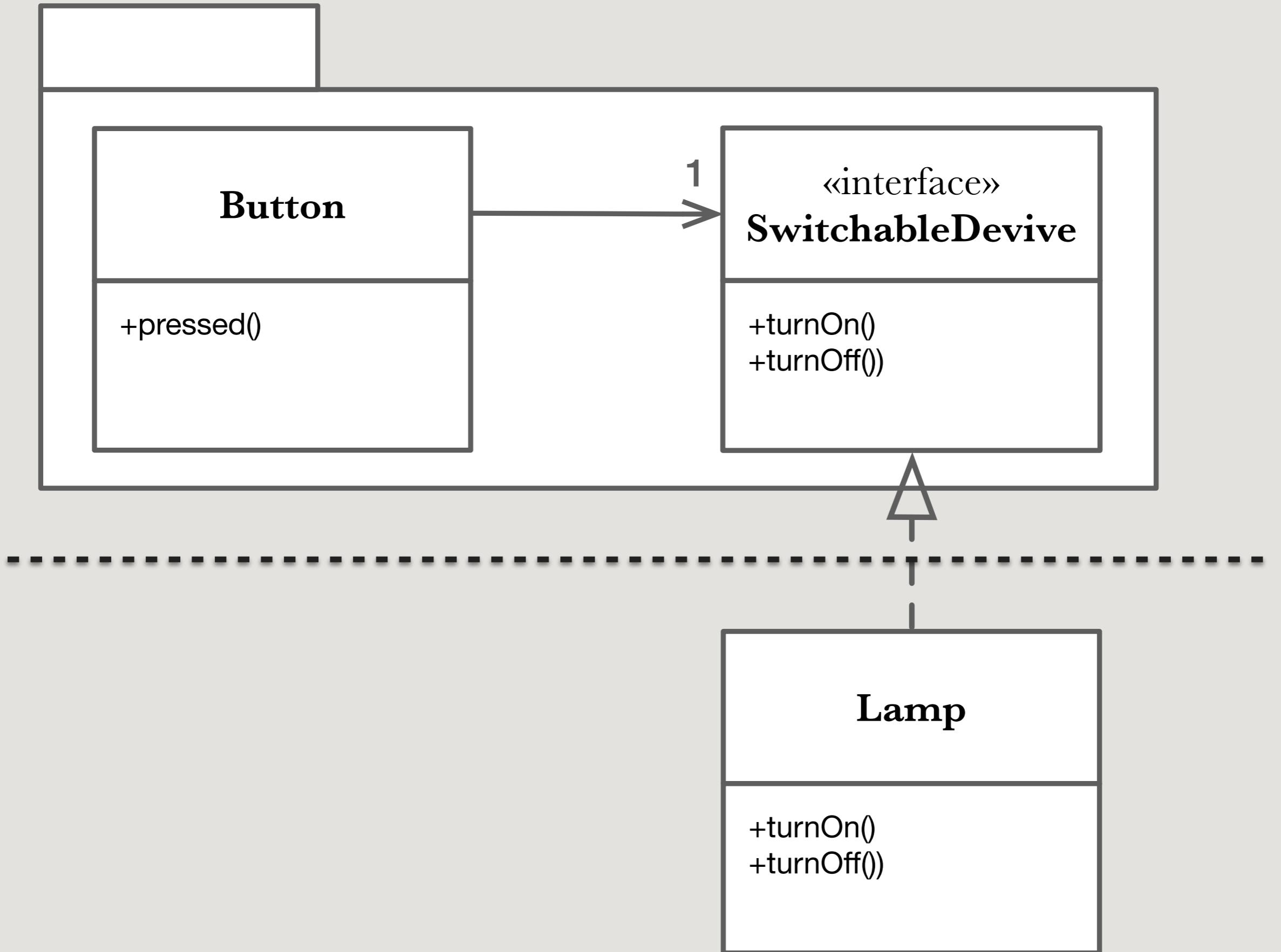




Button.java

```
public class Button {  
    private boolean on;  
    private Lamp lamp;  
    // ...  
    public void pressed() {  
        if (on) {  
            lamp.turnOff();  
            on = false;  
        } else {  
            lamp.turnOn();  
            on = true;  
        }  
    }  
}
```





El botón especifica el contrato de los dispositivos que puede controlar a través de una interfaz.

Lo hace el propio diseñador de la clase del botón; la interfaz «le pertenece», son indisociables.

Ahora el botón puede controlar cualquier dispositivo que implemente la interfaz proporcionada por él.

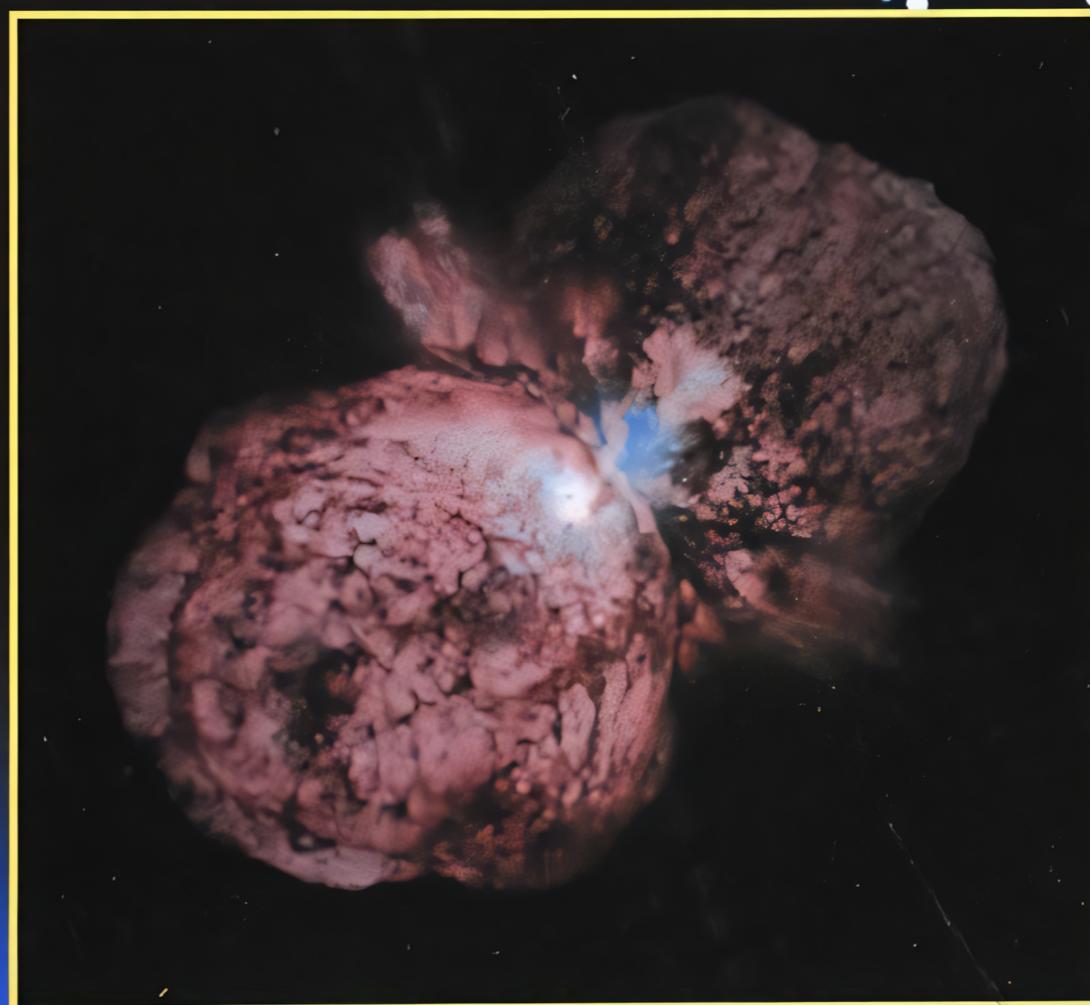
¡Incluso aquellos que aún no han sido inventados!

Bibliografía



AGILE SOFTWARE DEVELOPMENT

Principles, Patterns, and Practices



Robert C. Martin
with contributions by James W. Newkirk and Robert S. Koss

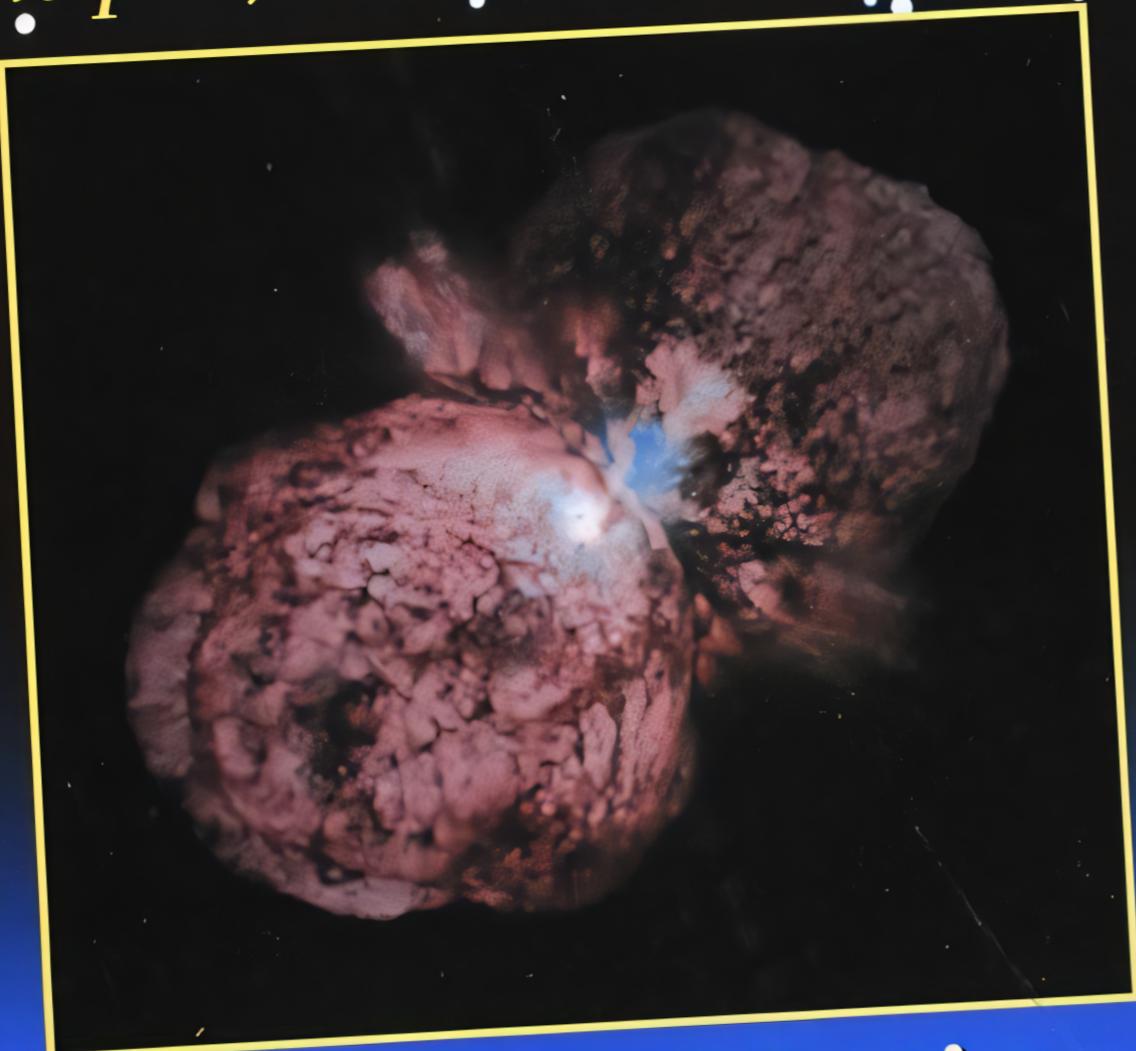
Robert C. Martin Series

UML FOR JAVA PROGRAMMERS

Robert C. Martin

AGILE SOFTWARE DEVELOPMENT

Principles, Patterns, and Practices



Robert C. Martin
with contributions by James W. Newkirk and Robert S. Koss

Robert C. Martin Series

UML FOR JAVA PROGRAMMERS

Robert C. Martin