SEMINARIO

1

Herencia

Principios de diseño

Diseño del Software

Grado en Ingeniería Informática del Software

Curso 2017-2018

Escuela de Ingeniería Informática
Universidad de Oviedo

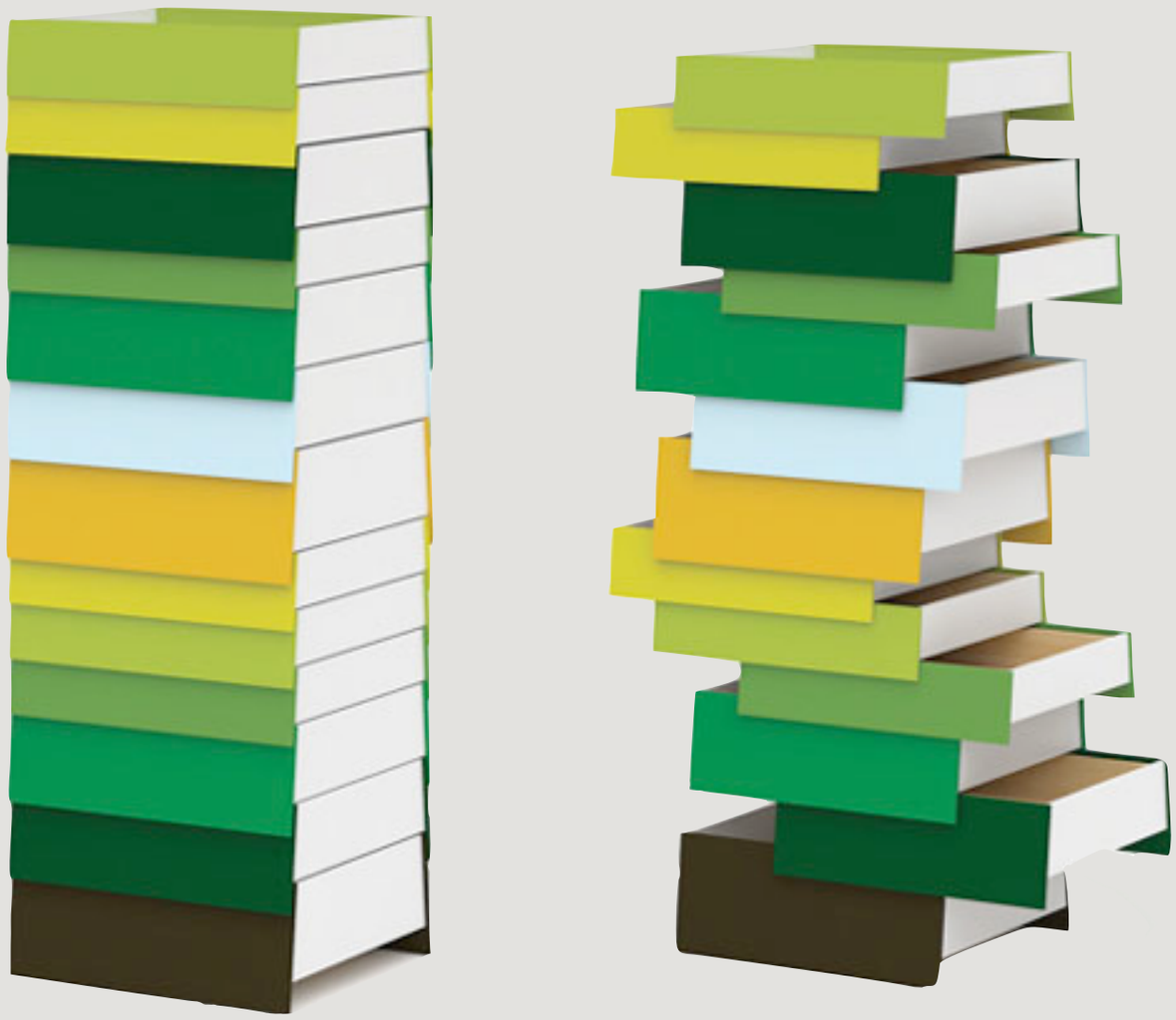
Raúl Izquierdo
César Acebal

Ejercicio 1

Una pila... ¿es un vector?

Ejercicio 1

- ¿Debería la clase `Stack` de Java derivar de `Vector`?



Ejercicio 1

- **¿Debería la clase `Stack` de Java derivar de `Vector`?**
  - Una pila (`Stack`) es una estructura de datos LIFO (*last-input-first-output*)
  - Se caracteriza por (como mínimo) un par de operaciones para meter y sacar elementos:
    - Push
    - Pop
  - Un vector es, en esencia, un `ArrayList`, es decir, una lista

¿Por qué no?

- **Varios problemas:**
  - ¿`Stack` no debería ser una interfaz? *
    - Si fuera una clase que heredase de `Vector`, no definiría simplemente un tipo, sino que nos obligaría a una determinada implementación, que:
      - Puede no servirnos (¿y si queremos una pila pero implementada internamente de otro modo, por eficiencia o lo que sea?)
      - ¿Y si nuestra clase (la que queremos que sea una pila) ya hereda de otra clase?
        - Java no admite la herencia múltiple

(*) En los próximos días, en clase de teoría, veremos qué criterios seguir para decidir entre clases e interfaces.

¿Por qué no?

- Una pila... ¿es un vector?
  - ¡No!

```
Stack<String> stack = new Stack<String>();
stack.push("1");
stack.push("2");
stack.push("3");
stack.insertElementAt("¡Cuélame!", 1);
while (!stack.isEmpty()) {
    System.out.println(stack.pop());
}

// prints "3", "2", "Cuélame", "1"
```

Es decir, estamos heredando operaciones de Vector que claramente no pertenecen al tipo abstracto Pila: violan su contrato.

¿Vemos cómo lo hace Java?

Stack (Java Platform SE 7)

docs.oracle.com/javase/7/docs/api/java/util/Stack.html

Lector

java.util

Class Stack<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.Vector<E>
        java.util.Stack<E>

¡Oh, oh!

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

---

```
public class Stack<E>
extends Vector<E>
```

The `Stack` class represents a last-in-first-out (LIFO) stack of objects. It extends class `Vector` with five operations that allow a vector to be treated as a stack. The usual `push` and `pop` operations are provided, as well as a method to `peek` at the top item on the stack, a method to test for whether the stack is `empty`, and a method to `search` the stack for an item and discover how far it is from the top.

When a stack is first created, it contains no items.

A more complete and consistent set of LIFO stack operations is provided by the `Deque` interface and its implementations, which should be used in preference to this class. For example:

```
Deque<Integer> stack = new ArrayDeque<Integer>();
```

Since:

JDK1.0

See Also:

Serialized Form

Field Summary

Fields inherited from class java.util.Vector

capacityIncrement, elementCount, elementData

¿En qué quedamos?

- **La biblioteca de clases de Java no siempre es perfecta**

Entiéndase consistente con las propias decisiones de diseño y convenios y guías de estilo del lenguaje y del resto de la biblioteca estándar.

> *Inheritance is appropriate only in circumstances where the subclass really is a subtype of the superclass. In other words, a class B should only extend a class A only if an "is-a" relationship exists between the two classes. If you are tempted to have a class B extend a class A, ask yourself this question: Is every B really an A? If you cannot truthfully answer yes to this question, B should not extend A. If the answer is no, it is often the case that B should contain a private instance of A and expose a smaller and simpler API; A is not an essential part of B, merely a detail of its implementation.*
>
> *There are a number of obvious violations of this principle in the Java platform libraries. For example, a stack is not a vector, so Stack should not extend Vector. Similarly, a property list is not a hash table, so Properties should not extend Hashtable. In both cases, composition would have been preferrable.*

Effective Java 2nd Edition, Item 16: Favor composition over inheritance

Deque (Java Platform SE 7)

docs.oracle.com/javase/7/docs/api/index.html?java/util/Deque.html

Lector

java.util

Interface Deque<E>

Type Parameters:

E - the type of elements held in this collection

All Superinterfaces:

Collection<E>, Iterable<E>, Queue<E>

All Known Subinterfaces:

BlockingDeque<E>

All Known Implementing Classes:

ArrayDeque, ConcurrentLinkedDeque, LinkedBlockingDeque, LinkedList

Por cierto, como curiosidad, ésta es la interfaz que se recomienda utilizar (alguna de sus implementaciones) en lugar de Stack a partir de Java 1.6.

```
public interface Deque<E>
extends Queue<E>
```

A linear collection that supports element insertion and removal at both ends. The name *deque* is short for "double ended queue" and is usually pronounced "deck". Most `Deque` implementations place no fixed limits on the number of elements they may contain, but this interface supports capacity-restricted deques as well as those with no fixed size limit.

This interface defines methods to access the elements at both ends of the deque. Methods are provided to insert, remove, and examine the element. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either `null` or `false`, depending on the operation). The latter form of the insert operation is designed specifically for use with capacity-restricted `Deque` implementations; in most implementations, insert operations cannot fail.

The twelve methods described above are summarized in the following table:

	First Element (Head)		Last Element (Tail)	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

Pero...

- **¿Es suficiente con el criterio «ES-UN»?**

  ¿Qué creéis?

- **Por ejemplo...**
  - ¿Debería una clase `Window` heredar de `Rectangle`?
  - ¿Una ventana es un rectángulo?
    - Sí, ¿no? ¿Entonces?

¡NO!

«Es un»

- **El famoso «ES UN» no es un criterio válido para aplicar la herencia**
  - ¿Una ventana es un rectángulo?
  - ¿Un pingüino es un pájaro (raro)?
  - ¿Un dibujo es una figura?
  - ¿Un círculo es un punto gordo?

Polimorfismo

- **Recordemos: las jerarquías surgen de una necesidad**
  - ¿El pingüino es un pájaro? Lo será si se quiere que el pingüino sea una de las formas de implementar dicha responsabilidad; es decir, si se quiere que sea usado allá donde se requiera un pájaro
  - ¡Y el pingüino será a su vez una figura si se quiere que se pueda añadir a una pizarra para dibujarse!
  - Y por esa razón una ventana no deriva de rectángulo aunque, efectivamente, «ES UN» rectángulo
    - No pretende usarse como sustituto del mismo en ningún momento

Que en español quede bien una frase no es razón para usar el polimorfismo (máxime cuando puede ser expresada de varias maneras)

Criterio de pertenencia a una jerarquía

- **Un objeto pertenecerá a una jerarquía si se quiere poder conectar a un cliente para que éste lo use como a uno mas de la misma**
  - (Para ofrecerle una implementación alternativa)
  - Para ello deberá cumplir todas las responsabilidades que requiere el interfaz
  - Nótese que con una que no sea aplicable el objeto no pertenecerá a la jerarquía

¡Nunca debe surgir una jerarquía como una mera clasificación!

Volviendo a la pila...

- **¿Es una pila un vector al que se le añaden un par de operaciones push/pop?**
  - NO IMPORTA, NO ES LA FORMA DE AVERIGUARLO
- **De lo que se trata es de:**
  - ¿Será pasado un objeto `Stack` en los lugares donde se requiera un `Vector`? ¿Es una alternativa de implementación?
    - No; por tanto no debería derivar
- **¿Cómo debería hacerse?**
  - `Stack` debería haber utilizado un `Vector` en su implementación privada protegiendo así los métodos de éste que no debieran invocarse

Ejercicio 2

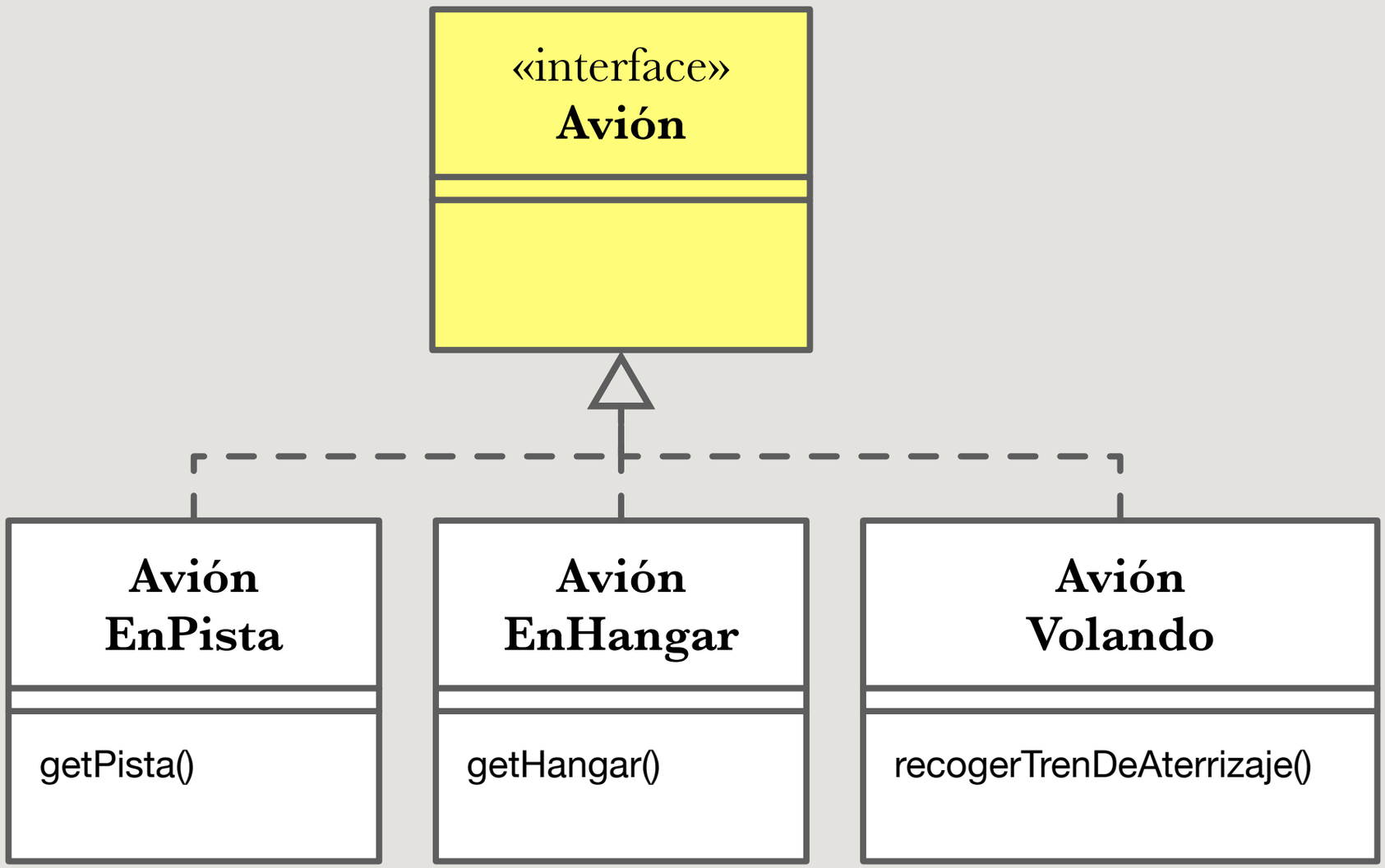
Un avión puede estar en el hangar, en la pista a punto de despegar (o tras el aterrizaje), en el aire... En cada una de esas situaciones tendrán sentido unas operaciones u otras (por ejemplo, saber el hangar o la pista en la que está, según se trate, recoger el tren de aterrizaje, etcétera). ¿Cómo lo plantearíais?

Ejercicio 2

- **En un aeropuerto los aviones pueden estar en un hangar, en pista o volando**
  - Cuando se está en un hangar se quiere poder saber el número de éste
  - Si el avión está en pista se quiere poder saber en cuál de ellas

Ejercicio 2

- ¿Sería correcto este diseño?



Ejercicio 3

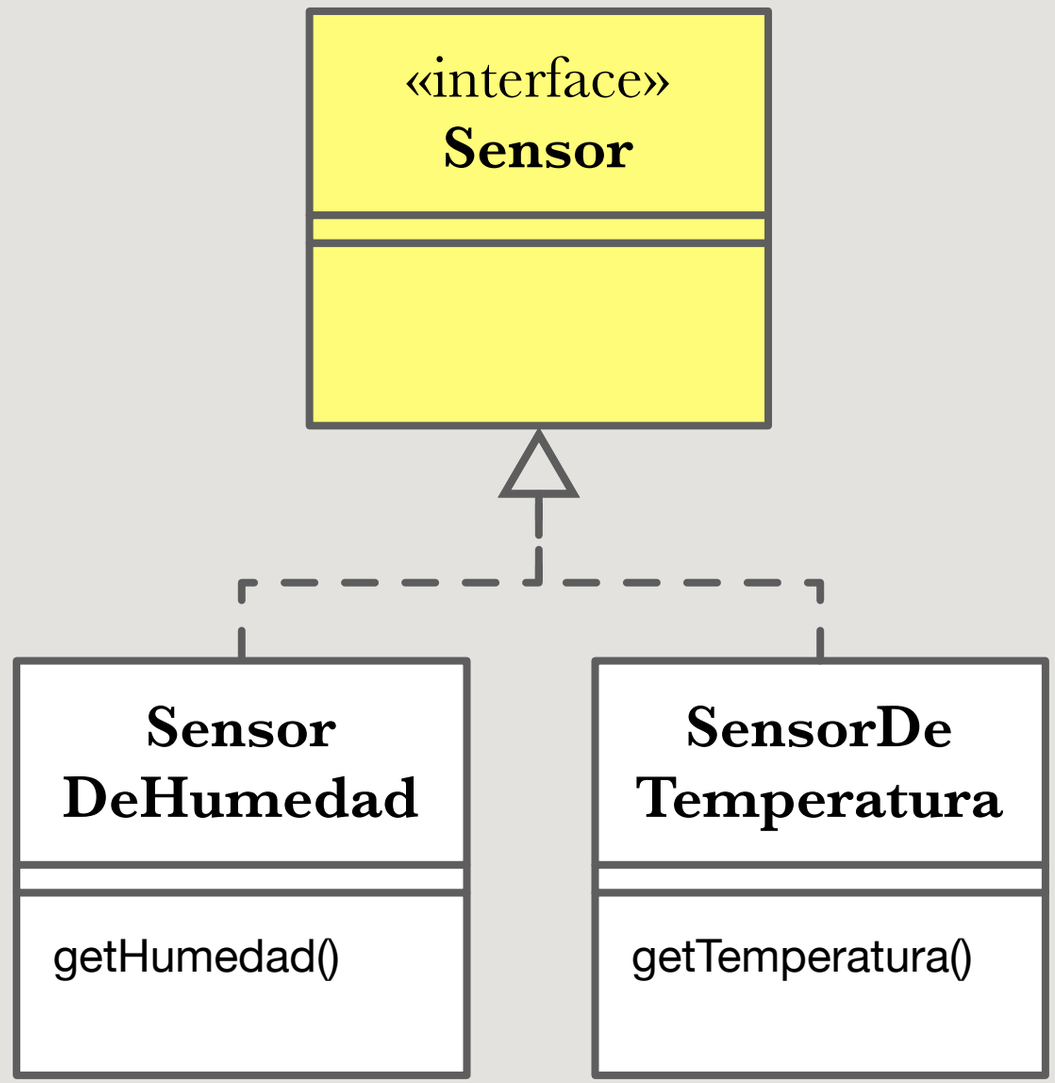
Sensores de humedad y de temperatura: ¿jerarquía?

Sensores

- **En un invernadero tenemos distintos tipos de sensores**
  - Los de temperatura permiten saber a cuántos grados centígrados se encuentra el aire
  - Los de humedad indican la humedad relativa del ambiente

Sensores

- ¿Sería correcto este diseño?



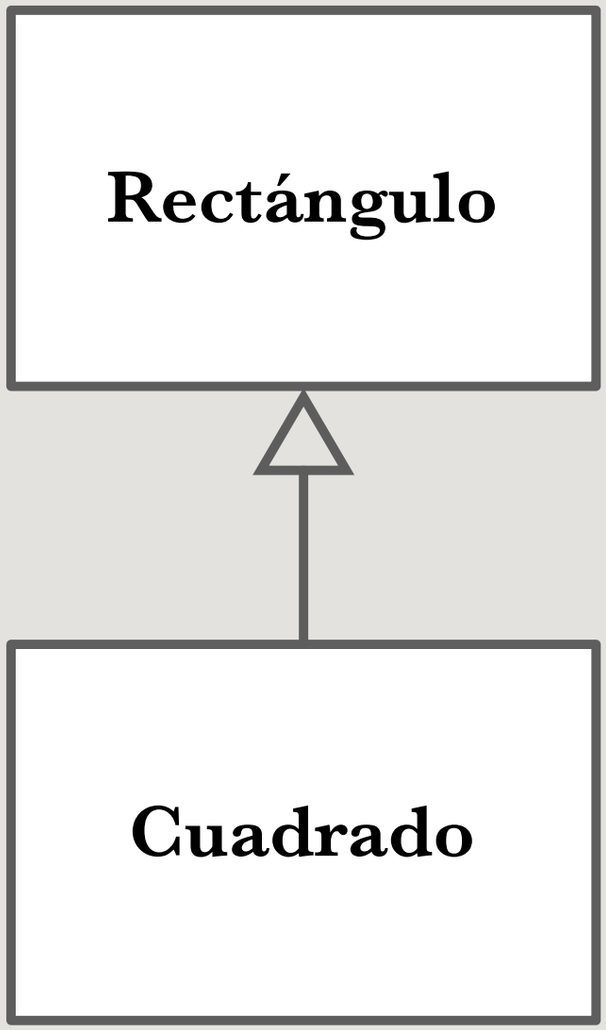
Sensores

- **Las interfaces surgen por la necesidad de declarar unas responsabilidades que un objeto ha delegado**
- **La jerarquía surge con alternativas de implementación de dichas responsabilidades**
- **Nunca debe surgir una jerarquía como mera clasificación**
- **Aquí son dos objetos diferentes sin relación de herencia**
  - Nunca se pasará un sensor sin que importe cuál de los dos es
  - No es que tengan distintos métodos: aunque los dos se llamasen `getValor` y se subieran a la clase base estarían mal

Ejercicio 4

Rectángulo - Cuadrado

- Un cuadrado... ¿es un rectángulo?



LSP

- **Al igual que en los casos anteriores, veamos si cumple el principio de sustitución de Liskov** *
  - ¿Podemos sustituir un rectángulo por un objeto cuadrado en todos los sitios en que se espere una referencia al primero?

(*) Que, de nuevo, veremos en las próximas clases de teoría.

```
class Rectangle
{
    private Point topLeft;
    private double width;
    private double height;

    public void setWidth(double width)
    {
        this.width = width;
    }

    public void setHeight(double height)
    {
        this.height = height;
    }

    ...
}
```

Problema

- **Al heredar dichas implementaciones, un cuadrado violaría su definición**
  - Dejarían de ser iguales
- **No obstante, el problema anterior tiene una fácil solución**
  - Aunque el código ya no sea el más elegante ni legible

```
class Square extends Rectangle
{
    public void setWidth(double width)
    {
        super.setWidth(width);
        super.setHeight(width);
    }

    public void setHeight(double height)
    {
        super.setHeight(height);
        super.setWidth(height);
    }
}
```

¿Resuelto?

- **Ahora, cuando alguien cambia el ancho del cuadrado, su altura cambia también (y viceversa)**
  - Así, las invariantes[(*)] del cuadrado se mantienen intactas

(*) Es decir, las propiedades de la clase que deben ser ciertas siempre, independientemente del estado.

Pero...

```
void g(Rectangle r)
{
    r.setWidth(5);
    r.setHeight(4);
    assert r.area() == 20;
}
```

Pero...

```
void g(Rectangle r)
{
    r.setWidth(5);
    r.setHeight(4);
    assert r.area() == 20;
}
```

¿Qué ha pasado?

- **Un método de algún cliente cambió el ancho y alto de lo que creía ser un objeto rectángulo**
- **El área sería, naturalmente, incorrecto**
  - ¡Es lógico pensar que cambiar el ancho de un rectángulo no va a cambiar también su alto!
- **Para ese método, rectángulos y cuadrados no serían intercambiables**

Violando así el LSP

Sobre la validez

- **El LSP nos lleva a una interesante conclusión**
- **Un modelo, considerado aisladamente, no se puede decir si es o no válido**
  - La validez sólo puede ser expresada en función de sus clientes
- **Así, al considerar si un diseño particular es apropiado o no, no podemos ver sólo la solución como algo aislado**
  - Debemos verlo en términos de las suposiciones razonables que harán los usuarios de dicho diseño

«ES-UN»

- **Pero... ¿qué ocurre entonces? Después de todo, ¿un rectángulo no es un cuadrado? ¿No cumple la famosa relación «ES-UN»?**
  - ¡No para el autor del método *g*!
- **Para él, un cuadrado, definitivamente, no es un rectángulo**
- **¿Por qué?**

¡Porque no se comportan igual!

Comportamiento

- Y en el software (y en orientación a objetos en particular) el comportamiento lo es todo
- Así, el LSP deja claro que la relación «ES-UN», en OOD, se refiere al comportamiento que puede ser razonablemente supuesto, del que dependen los clientes