1 Introducción al diseño OO

(1ª parte)

Diseño del Software

Grado en Ingeniería Informática del Software

Curso 2022-2023

Escuela de Ingeniería Informática
Universidad de Oviedo

Raúl Izquierdo
César Acebal

Elementos del modelo de objetos

Diseño del Software

Grado en Ingeniería Informática del Software

Curso 2022-2023

Escuela de Ingeniería Informática
Universidad de Oviedo

Raúl Izquierdo
César Acebal

(inherente)

Complejidad del software

Software systems are perhaps the most intricate and complex [...] of the things humanity makes.

—Brooks (1995)

Brooks, F. P., Jr. 1995. The Mythical Man-Month (anniversary ed.). Boston, MA: Addison-Wesley

Einstein arguyó que debe haber explicaciones simplificadas de la naturaleza, porque Dios no es caprichoso ni arbitrario. No hay fe semejante que conforte al ingeniero de software. Mucha de la complejidad que debe dominar es complejidad arbitraria.

—Brooks (1987)

Brooks, F. April 1987. No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer, vol. 20(4), p. 12

No Silver Bullet

Essence and Accidents of Software Engineering

Frederick P. Brooks, Jr.
University of North Carolina at Chapel Hill

Fashioning complex conceptual constructs is the *essence;* *accidental* tasks arise in representing the constructs in language. Past progress has so reduced the accidental tasks that future progress now depends upon addressing the essence.

Of all the monsters that fill the nightmares of our folklore, none terrify more than werewolves, because they transform unexpectedly from the familiar into horrors. For these, one seeks bullets of silver that can magically lay them to rest.

The familiar software project, at least as seen by the nontechnical manager, has something of this character; it is usually innocent and straightforward, but is capable of becoming a monster of missed schedules, blown budgets, and flawed products. So we hear desperate cries for a silver bullet—something to make software costs drop as rapidly as computer hardware costs do.

But, as we look to the horizon of a decade hence, we see no silver bullet. There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity. In this article, I shall try to show why, by examining both the nature of the software problem and the properties of the bullets proposed.

Skepticism is not pessimism, however. Although we see no startling breakthroughs—and indeed, I believe such to be inconsistent with the nature of software—many encouraging innovations are under way. A disciplined, consistent effort to develop, propagate, and exploit these innovations should indeed yield an order-of-magnitude improvement. There is no royal road, but there is a road.

The first step toward the management of disease was replacement of demon theories and humours theories by the germ theory. That very step, the beginning of hope, in itself dashed all hopes of magical solutions. It told workers that progress would be made stepwise, at great effort, and that a persistent, unremitting care would have to be paid to a discipline of cleanliness. So it is with software engineering today.

This article was first published in *Information Processing '86,* ISBN No. 0-444-70077-3, H.-J. Kugler, ed., Elsevier Science Publishers B.V. (North-Holland)

Does it have to be hard?—Essential difficulties

Not only are there no silver bullets now in view, the very nature of software makes it unlikely that there will be any—no inventions that will do for software productivity, reliability, and simplicity what electronics, transistors, and large-scale integration did for computer hardware.

Nosotros nos centraremos en cómo lidiar con esa complejidad desde el punto de vista del diseño.

Diseño orientado a objetos

Algunas definiciones previas y repaso de conceptos fundamentales

¿Qué es diseño?

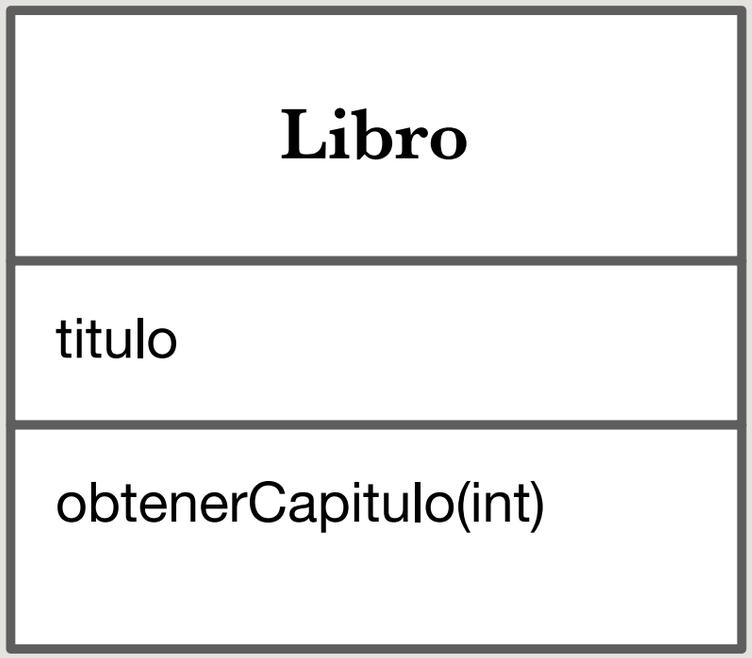
- **El análisis pone énfasis en una investigación del problema y los requisitos, en vez de en la solución**
  - Si se desea un sistema de información informatizado para una biblioteca, ¿cómo se utilizará? (¿qué debe hacer?)
- **El diseño pone énfasis en una solución conceptual que satisface los requisitos**

¿Y qué es diseño orientado a objetos?

- **Durante el diseño orientado a objetos se presta especial atención a la definición de los objetos software y cómo colaboran para satisfacer los requisitos**
  - Por ejemplo, un objeto **`Libro`** podría tener un atributo **`titulo`** y un método **`obtenerCapitulo`**

UML y patrones (2ª edición). Craig Larmen. Prentice Hall, 2003

Representación en UML

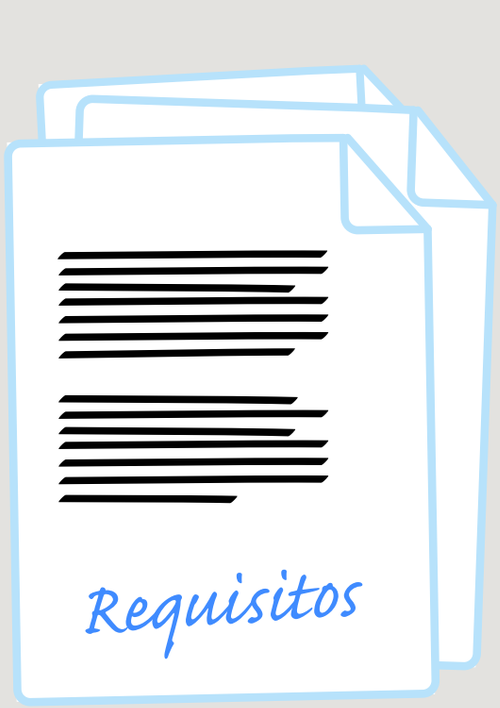


Representación de la clase Libro en UML, mediante un diagrama de clases.

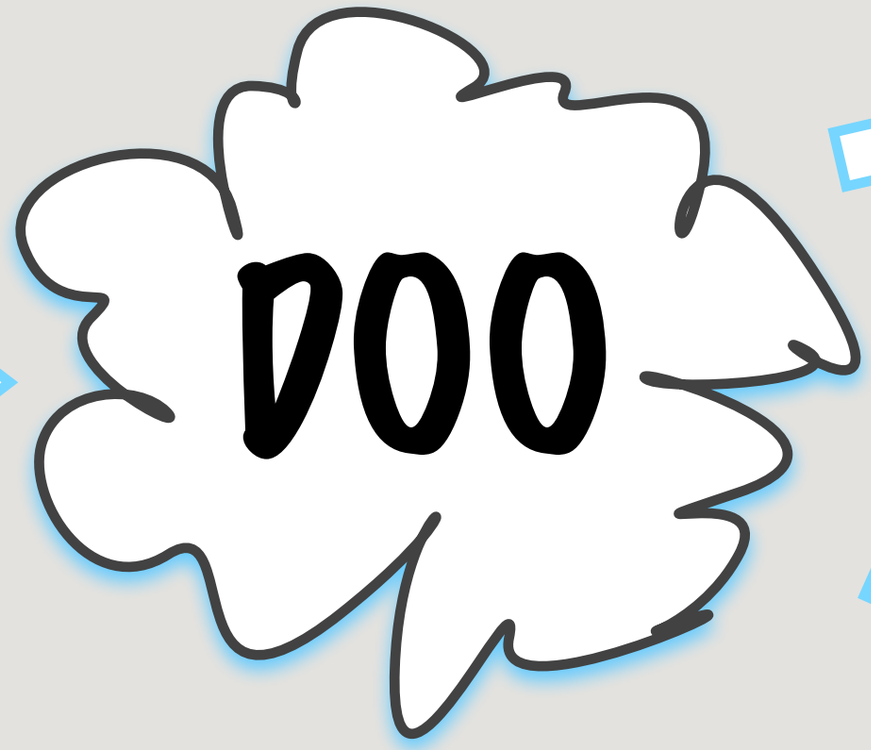
```
public class Libro
{
  private String titulo;

  public Capitulo obtenerCapitulo(
                int numeroDeCapitulo)
  {
    ...
  }
}
```

Implementación

Resultado del diseño (*)



Análisis



Diseño orientado a objetos

Un sistema de objetos que satisfacen los requisitos

Una descripción del comportamiento público de esos objetos

Cómo se comunican esos objetos entre sí

(*) Según Rebecca Wirfs-Brock en su libro «Designing Object-Oriented Software» (Prentice-Hall, 1990)

¿Qué significa «orientado a objetos»?

No resulta fácil de definir

I have a cat named Trash. In the current political climate it would seem that if I were trying to sell him at least to a Computer Scientist, I would not stress that he is gentle to humans and is self-sufficient, living mostly on field mice. Rather, I would argue that he is object-oriented.

Roger Kind (1989)

(Después de todo, un gato exhibe un comportamiento característico, responde a mensajes, posee una larga tradición de respuestas heredadas y se encarga de gestionar su propio estado interno de manera bastante independiente.)

King, R. 1989. My Cat is Object-Oriented. In: *Object-Oriented Concepts, Databases, and Applications*, pp. 23-30. New York (NY): ACM.

Otra «definición»:

X is Good

Object-Oriented is Good

Ergo, X is Object-Oriented

Stroustrup, B. May 1988. What is object-oriented programming? IEEE Computer, vol. 5(3), p. 10-20

¿Y para vosotros?

¿Qué entendéis por diseño?

Y, sobre todo, ¿por qué creéis que es importante? (si es que lo creéis, claro —si no, disimulad—). Es decir, ¿por qué pensáis que debemos prestarle atención? ¿Por qué no ponernos a codificar directamente?

si hay una constante en el
desarrollo de software ésta es...

el CAMBIO

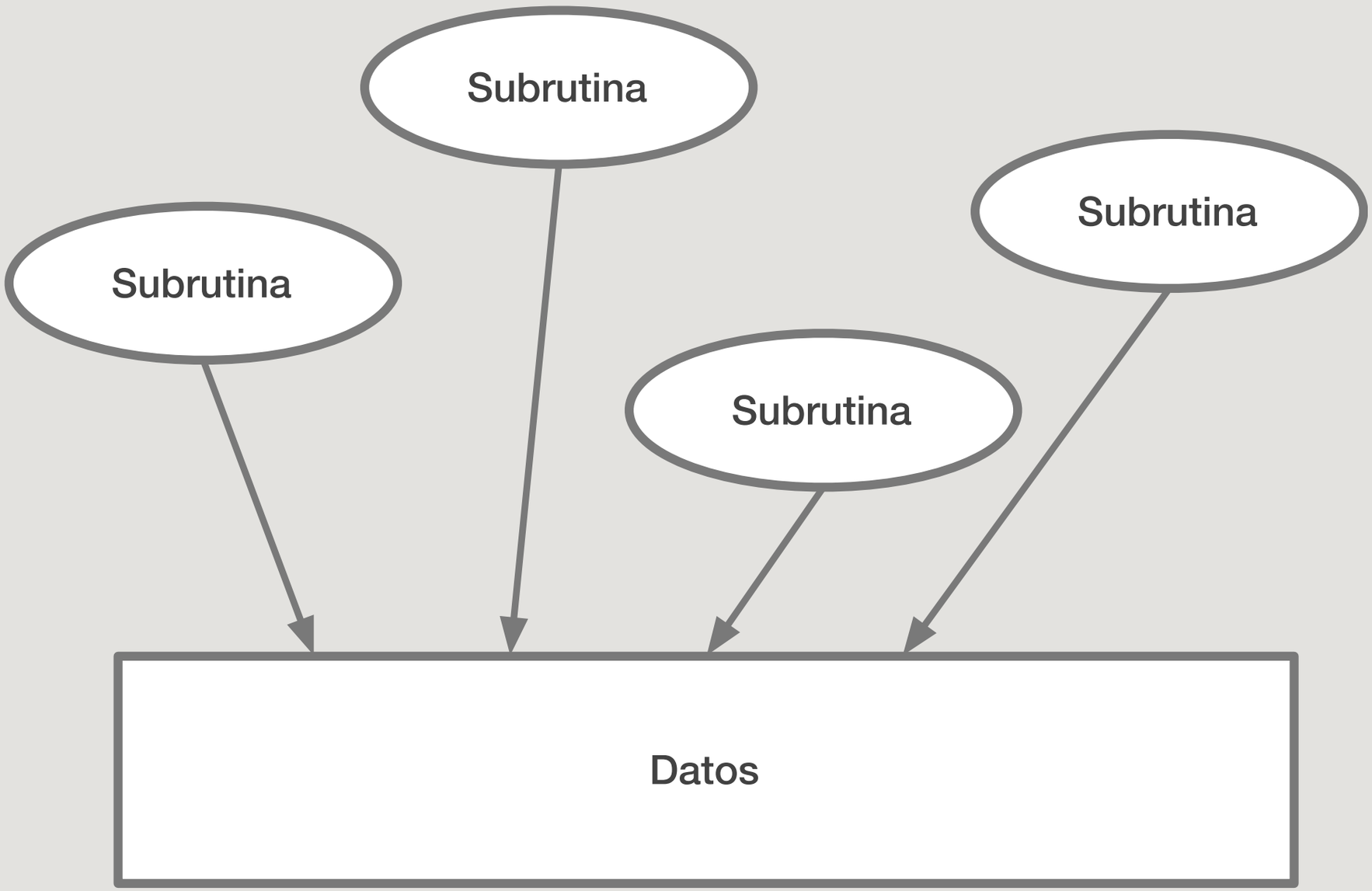
se va a producir.

siempre.

Programación estructurada

- La programación estructurada separaba las estructuras de datos de los algoritmos que las manipulaban

Programación estructurada



Orientación a objetos

- **El diseño orientado a objetos construye las aplicaciones mediante una estructura de objetos que colaboran entre sí**
- **Cada objeto realiza unas operaciones que lo caracterizan**
  - No importa cómo las realice, sino cuáles realiza
- **La implementación de esas operaciones son los métodos**

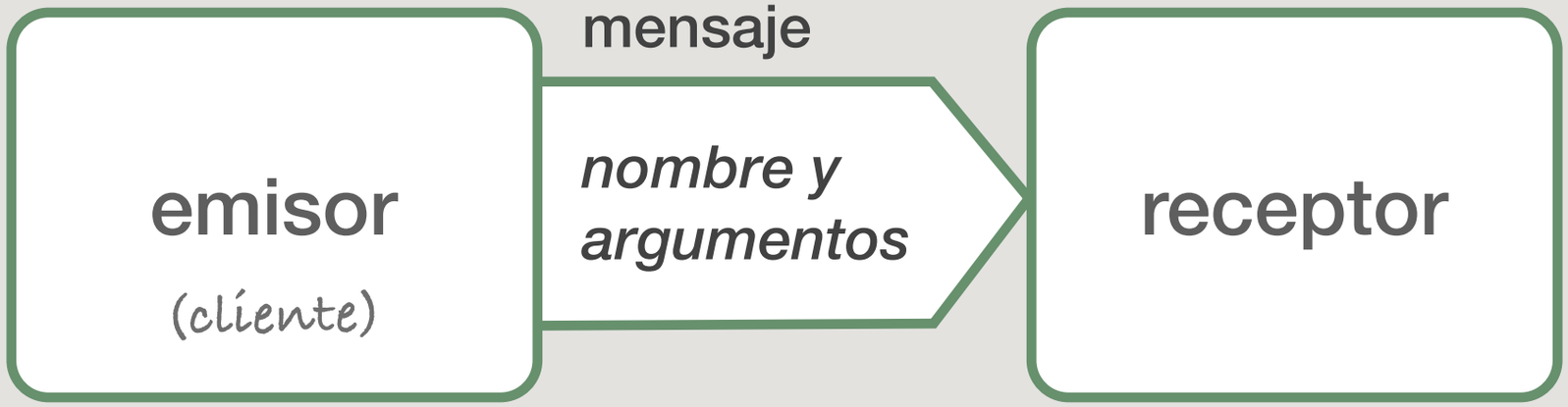
Orientación a objetos

- **Un objeto lleva a cabo una operación cuando recibe una petición (o mensaje) de un cliente**
  - Un mensaje es la única forma de decirle a un objeto que ejecute una operación
  - Las operaciones son el único modo de cambiar el estado interno del objeto
  - Por ese motivo, se dice que el estado interno del objeto está encapsulado
    - No se puede acceder a él directamente, y su representación es invisible desde el exterior del objeto

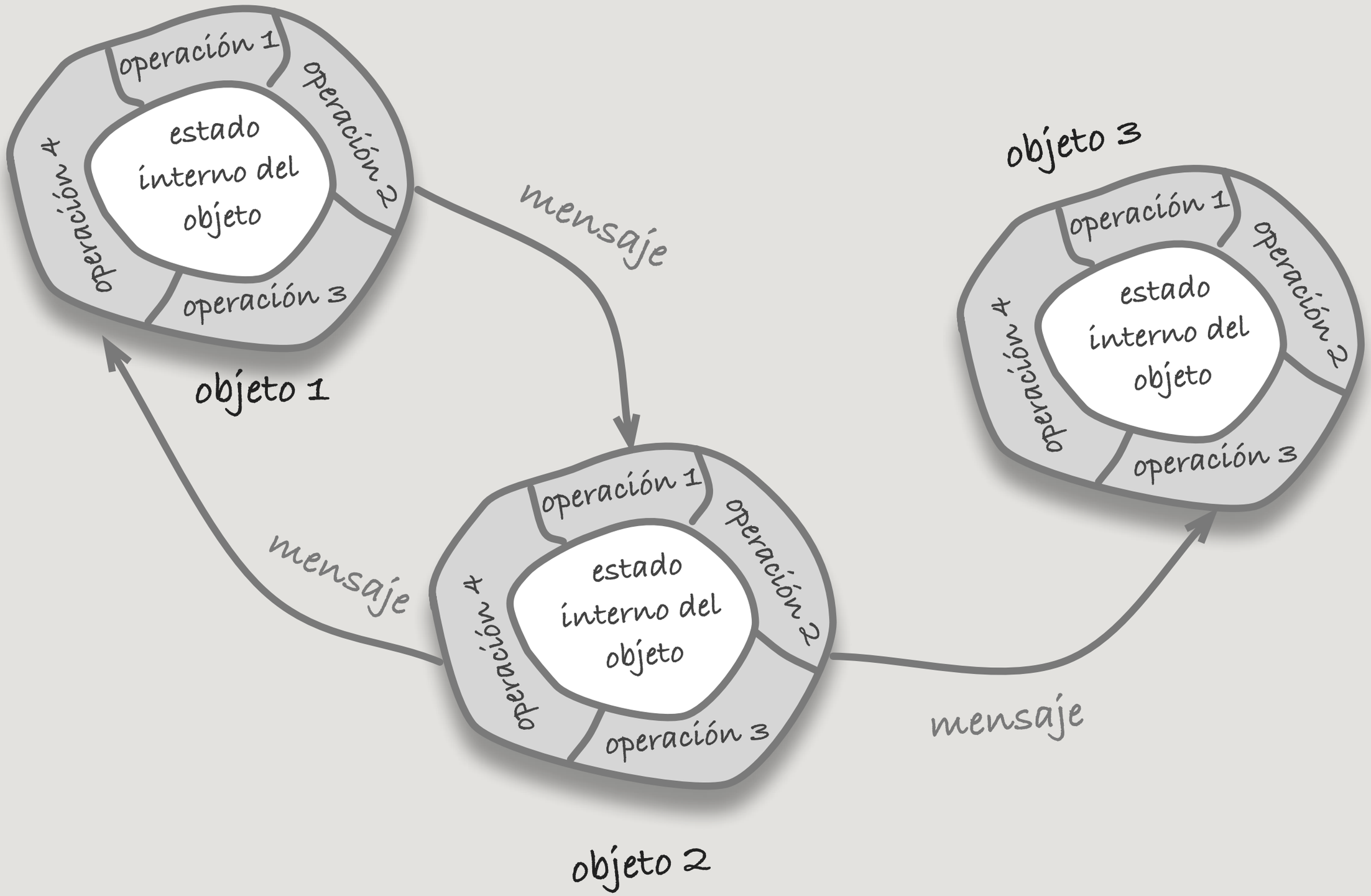
Mensajes

- Una aplicación realiza sus operaciones mediante la combinación de las operaciones de uno o más objetos
- Cuando se desea que un objeto realice una operación se le envía un mensaje
- Para realizarla podrá enviar a su vez mensajes a otros objetos

Mensajes



Objetos, operaciones, mensajes... colaboraciones



Especificar la interfaz de los objetos

Interfaz de un objeto

- **Cada operación declarada por un objeto especifica: su nombre, los objetos que recibe como parámetros y el tipo de valor devuelto por la operación**
  - Es lo que se conoce como signatura de la operación
- **Interfaz de un objeto: el conjunto de todas las signaturas (públicas) definidas por él**
  - Son las peticiones que pueden hacérsele

Interfaz de un objeto

- **Un tipo es un nombre usado para denotar una determinada interfaz**
  - Decimos que un objeto es del tipo **`Window`** si puede aceptar todas las peticiones para las operaciones definidas en la interfaz **`Window`**
  - Un objeto puede tener muchos tipos, así como objetos muy diferentes pueden compartir un mismo tipo
    - Parte de la interfaz de un objeto puede estar caracterizada por un tipo, y el resto por otros tipos

Interfaz de un objeto

- **Un tipo es subtipo de otro si su interfaz contiene a la interfaz de su supertipo**
  - Decimos que el subtipo «hereda» la interfaz de su supertipo
- **Las interfaces son fundamentales en el diseño orientado a objetos**
  - Todo lo que sabemos de un objeto nos lo dice su interfaz (que, a su vez, no nos dice nada de su implementación)

Enlace dinámico

Y su consecuencia: el polimorfismo

Enlace dinámico

- **La operación concreta que se ejecuta cuando se envía una petición a un objeto depende de:**
  - La petición
  - El objeto receptor

A la asociación en tiempo de ejecución que tiene lugar entre la petición a un objeto y una de sus operaciones se le denomina enlace dinámico

Enlace dinámico

- **El enlace dinámico permite:**
  - Escribir programas que esperen un objeto con una interfaz determinada, sabiendo que cualquier objeto que tenga esa interfaz aceptará la petición
  - Más aún, sustituir en tiempo de ejecución un objeto por otro con la misma interfaz

↳ Polimorfismo

Polimorfismo

Hemos hablado de las referencias y sus tipos, por un lado, y del enlace dinámico, por otro. Antes de continuar (y aunque todavía no hemos tratado con la herencia en sí), repasemos el aspecto fundamental de la orientación a objetos, su razón de ser (y, por ende, del diseño orientado a objetos): el polimorfismo.

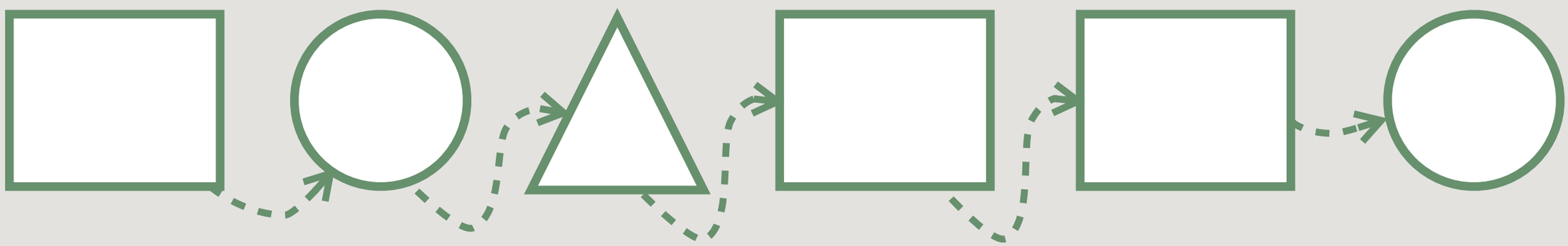
Polimorfismo

- **Es la capacidad de dos o más tipos de objetos de responder al mismo mensaje, cada uno a su manera**
- **Un objeto no necesita saber a quién está enviando un mensaje** (no necesita «conocerlo»)
  - Un objeto envía un mensaje: si el receptor implementa un método con la misma signatura, responderá
  - El emisor no necesita saber la clase concreta del receptor

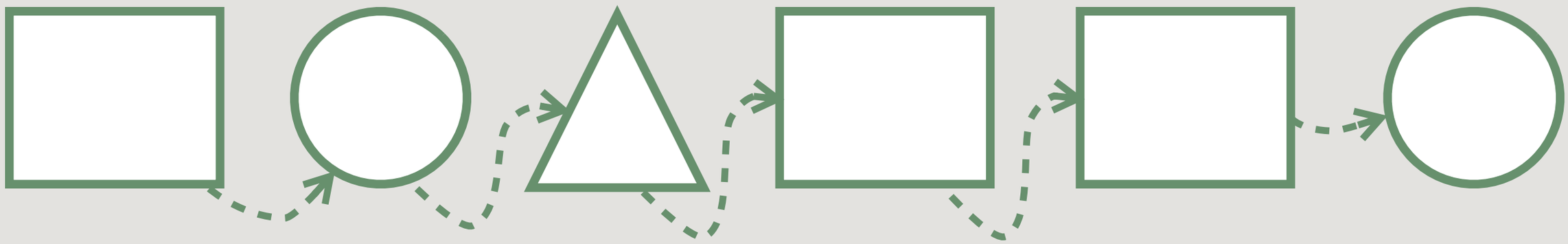
Polimorfismo

- **El polimorfismo es lo que nos permite, como sabemos, tener una lista de figuras y poder dibujarlas todas sin conocerlas**
  - (Sin saber de qué tipo concreto es cada una)

```
List<Figura> figuras = new ArrayList<Figura>();
```

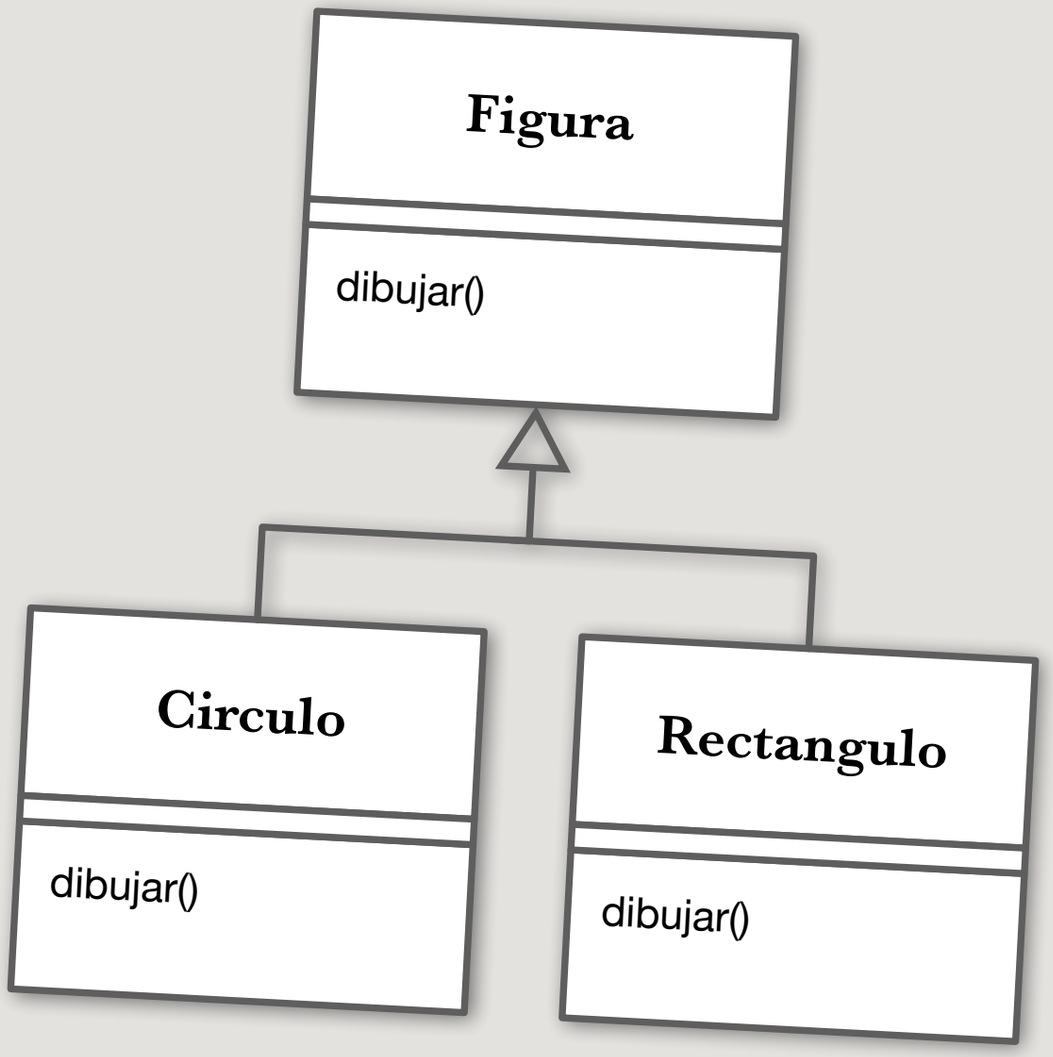


```
List<Figura> figuras = new ArrayList<Figura>();
```



```
for (Figura figura : figuras) {
        figura.dibujar();
}
```

una única operación, tres implementaciones (métodos) distintas

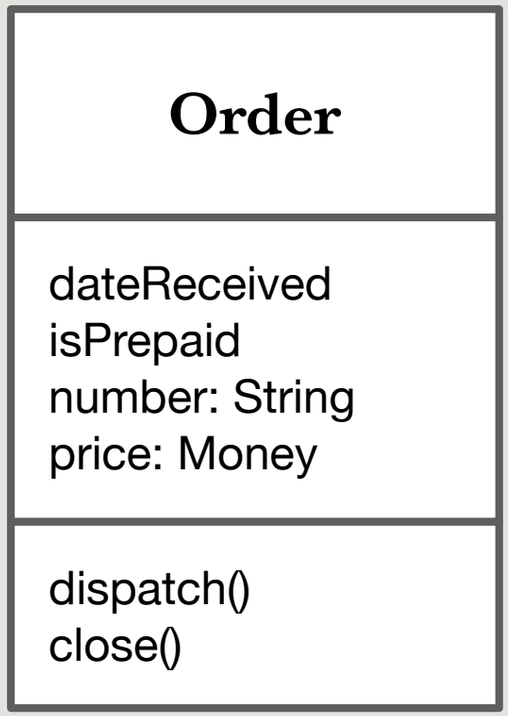


Clases

O cómo especificar la implementación de los objetos

Especificar las implementaciones de los objetos

- **¿Cómo se define realmente un objeto?**
  - Por su clase (es la implementación del objeto)
- **La clase es *una* de las formas de crear tipos en Java**



Representación de una clase en UML

Definición de clases

- **Paso 1**
  - Determinar el nombre que se quiere dar al conjunto de las operaciones
- **Paso 2**
  - Elección de las operaciones que realizarán los objetos del nuevo tipo

Qué mensajes podrán recibir

  - Las operaciones pueden ser de dos tipos:
    - Acciones: ordenan al objeto hacer algo
    - Consultas: piden al objeto alguna información

Definición de clases

- **Paso 3**
  - Implementación de los mensajes

  - Nótese la distinción hecha entre los pasos dos y tres
    - En un caso real habría varios días de por medio
  - En un método se pueden ver dos cosas distintas
    - Define un mensaje que es capaz de recibir el objeto
    - Especifica cómo reacciona el objeto ante la recepción del mismo
  - Para que un objeto realice una operación se le envía un mensaje
    - La recepción del mismo producirá la ejecución de su implementación

Ejemplo: cronómetro

Cronómetro

Paso 1

¿Qué nombre le damos a este tipo de objetos?

```
class Cronometro
{
   ...
}
```

Cronómetro

- **Paso 2**

  ¿Qué responsabilidades tiene un cronómetro?

```
class Cronometro
{
  public void ponerEnMarcha() {...}
  public void parar() {...}
  public long tiempoTranscurrido() {...}
}
```

Cronómetro

- **Paso 3**

Ahora implementamos los métodos

```
class Cronometro
{
  public void ponerEnMarcha()
  {
    // ...
  }

  public void parar()
  {
    // ...
  }

  public long tiempoTranscurrido()
  {
    // ...
  }
}
```

Creación de objetos

Vale, ya sabemos cómo se definen las clases. Repasemos ahora brevemente en qué consiste la creación de objetos.

Creación de objetos

- Los objetos se crean «instanciando»† clases
- Un objeto se dice que es una «instancia»† de una clase

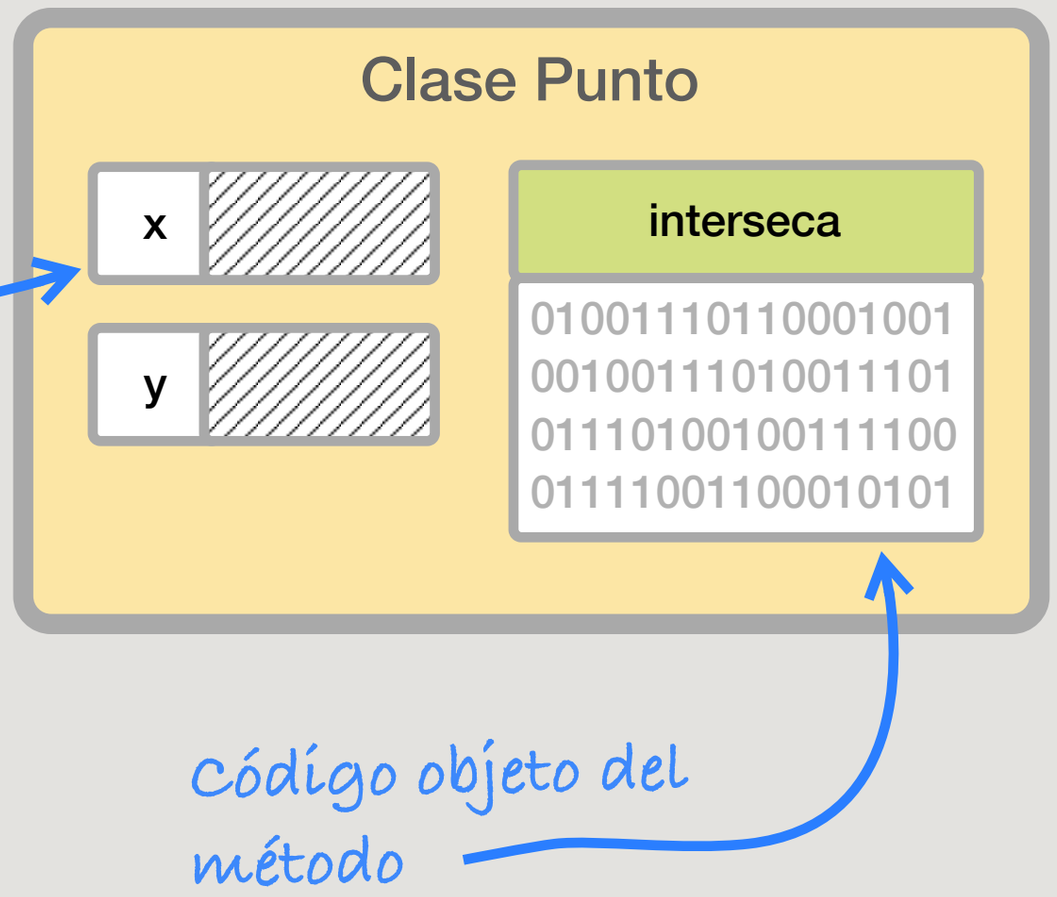
† los términos "instancia" e "instanciar" son barbarismos. Hubiera sido mejor hablar de "ejemplares" y de "crear un ejemplar" o, simplemente, "crear un objeto".

Crear un objeto

- El proceso de creación de un objeto almacena memoria para las variables del objeto y asocia sus operaciones (definidas en la clase) con esos datos

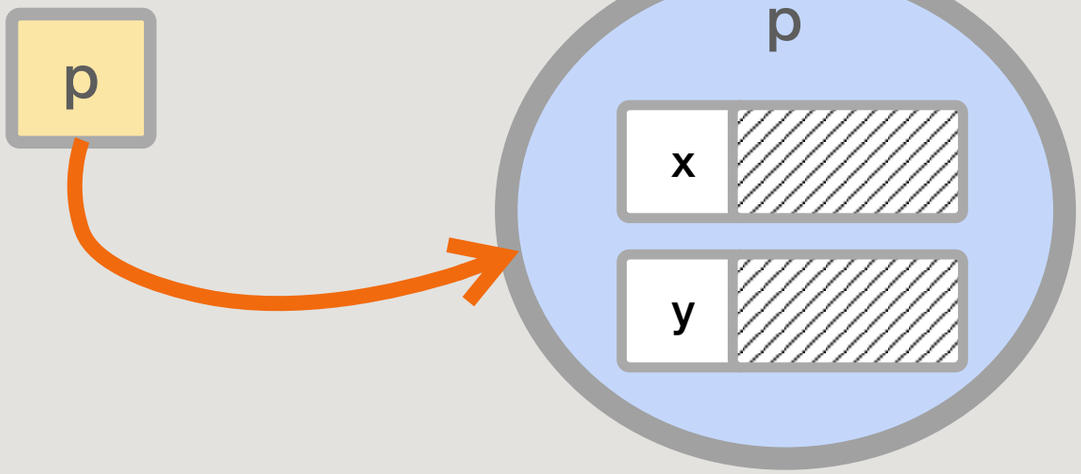
Nombre y tipo de los atributos

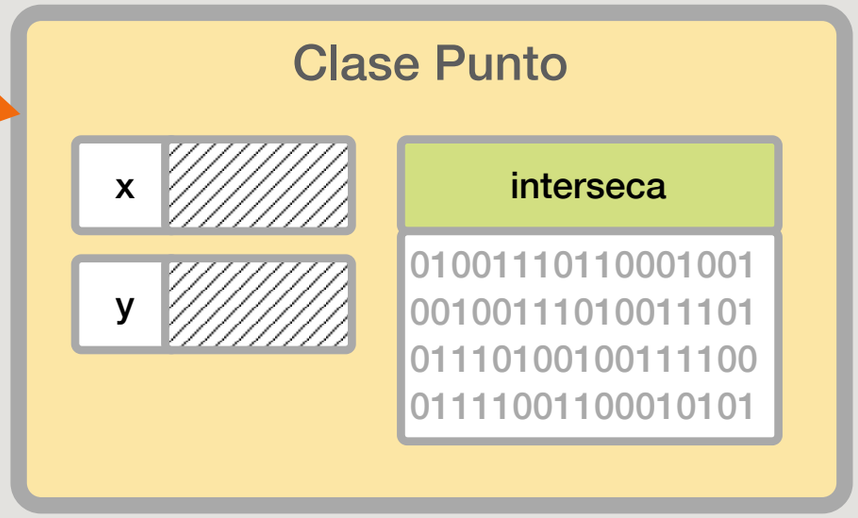
```
class Punto
{
   int x;
   int y;

   boolean interseca(Punto punto)
   {
      return punto.x == x && punto.y == y;
   }
}
```



Crear un objeto

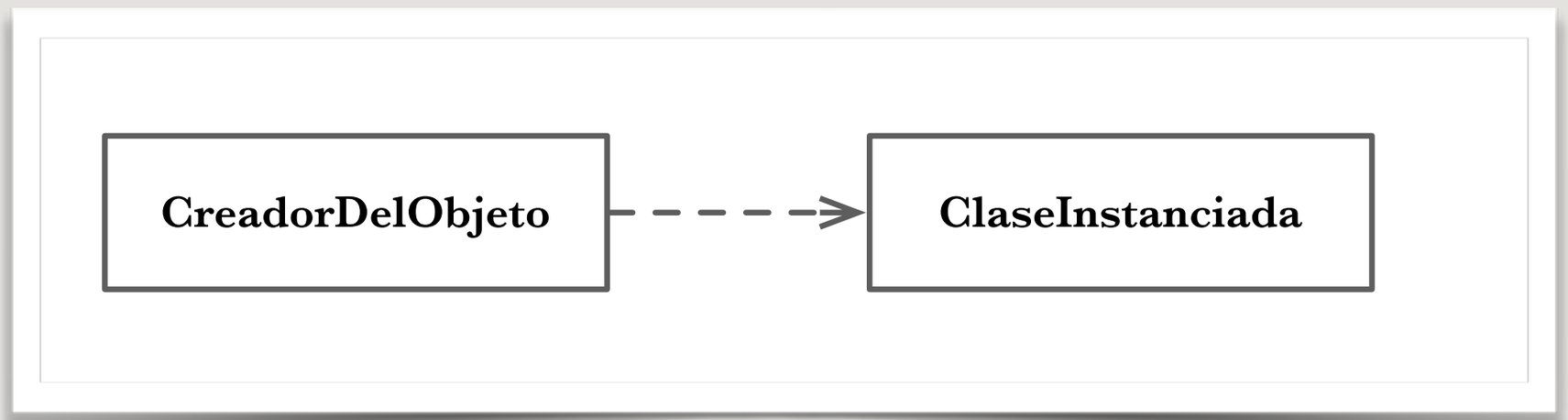
```
Punto p;
p = new Punto();
```





¿Qué es «p»?

Una referencia



En UML, la creación de un objeto se puede representar por medio de una flecha rayada (relación de dependencia entre el creador del objeto y la clase a instanciar)

Referencias

- **En Java todo objeto se manipula a través de una referencia**

```
Cronometro garminFenix7x;
```

- **Una referencia guarda el identificador de un objeto**
  - Apunta al objeto creado en memoria

Referencias

- En Java todo objeto se manipula a través de una referencia

```
Cronometro garminFenix7x;
```

- Una referencia apunta al objeto creado en memoria

```
garminFenix7x = new Cronometro();
```

El operador «new» crea un nuevo objeto de la clase indicada y devuelve una referencia a él

Referencias

- **El tipo de la referencia indica qué mensajes se pueden enviar al objeto**

Nos estamos refiriendo siempre a lenguajes orientados a objetos con comprobación estática de tipos (Java, C++, C#...).

Existen lenguajes orientados a objetos que no tienen clases.

- Smalltalk, JavaScript...

Creación de objetos

- **En los lenguajes imperativos el modelo de invocación que se sigue es el de las funciones matemáticas**
  - Se aplica una función a unos datos y se obtiene un resultado

```
f(x)
```

  - Se decide qué hay que hacer y se distribuye en una jerarquía de funciones

Creación de objetos

- En orientación a objetos no hay que seguir ese modelo
- Ahora se trata de decir a alguien que haga algo

```
esteObjeto.hazEsto();
```

- Ya no sólo hay que decidir *qué* hacer, sino también *quién* debe hacerlo

Si se aprende a hacer las cosas sin un quién luego no se ve la necesidad de introducirlo.

Creación de objetos

- **Síntomas de programación estructurada encubierta:**
  - Métodos largos
    - Más de 10 líneas

    No se está delegando
  - Abundancia de llamadas a métodos del mismo objeto

    Lo mismo: «C con clases»
  - Preguntarse cómo se calcula un factorial en un lenguaje OO

    Lo importante es el quién, no el cómo
  - Típica clase **`Utilidades`**

    No se encuentra un quién: no se han asignado bien las responsabilidades

Fields

Modifier and Type	Field and Description
static double	**E** The double value that is closer than any other to *e*, the base of the natural logarithms.
static double	**PI** The double value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

Method Summary

Methods

Modifier and Type	Method and Description
static double	**abs**(double a) Returns the absolute value of a double value.
static float	**abs**(float a) Returns the absolute value of a float value.
static int	**abs**(int a) Returns the absolute value of an int value.
static long	**abs**(long a) Returns the absolute value of a long value.
static double	**acos**(double a) Returns the arc cosine of a value; the returned angle is in the range 0.0 through *pi*.
static double	**asin**(double a) Returns the arc sine of a value; the returned angle is in the range -*pi*/2 through *pi*/2.
static double	**atan**(double a) Returns the arc tangent of a value; the returned angle is in the range -*pi*/2 through *pi*/2.
static double	**atan2**(double y, double x) Returns the angle *theta* from the conversion of rectangular coordinates (x, y) to polar coordinates (r, *theta*).
static double	**cbrt**(double a) Returns the cube root of a double value.
static double	**ceil**(double a) Returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer.
static double	**copySign**(double magnitude, double sign) Returns the first floating-point argument with the sign of the second floating-point argument.
static float	**copySign**(float magnitude, float sign) Returns the first floating-point argument with the sign of the second floating-point argument.
static double	**cos**(double a) Returns the trigonometric cosine of an angle.
static double	**cosh**(double x) Returns the hyperbolic cosine of a double value.
static double	**exp**(double a)

Atributos

La memoria del objeto
(aunque no su cerebro)

Atributos

- Volvamos a la implementación de nuestro cronómetro

```
class CronometroTest
{
    public static void main(String[] args)
    {
        Cronometro cronometro = new Cronometro();
        cronometro.ponerEnMarcha();

        // operación a cronometrar

        cronometro.parar();

        // ¿tiempo transcurrido?

        System.out.println(cronometro.tiempoTranscurrido());
    }
}
```

Atributos

- Para poder hacerlo cuando el objeto reciba el mensaje `parar` necesitará saber a qué hora recibió el mensaje `ponerEnMarcha`

```
class Cronometro
{
    public void ponerEnMarcha() {
        // Recordar a qué hora recibió este mensaje
    }

    public void parar() {
        // Calcular cuánto tiempo ha transcurrido
    }
}
```

Atributos

- Los objetos necesitan alguna forma de guardar información para su posterior uso por otros mensajes
- Esta información se guarda en los atributos

Estado interno del objeto

Atributos

```
class Cronometro
{
    private boolean parado = true;
    private long horaDeComiento;
    private long tiempo;

    public void ponerEnMarcha() {
        parado = false;
        horaDeComienzo = System.currentTimeMillis();
    }

    public void parar() {
        parado = true;
        tiempo = System.currentTimeMillis() -
                        horaDeComienzo;
    }
}
```

Atributos

- Nótese el papel de los atributos: usados exclusivamente para dar soporte a la implementación de los mensajes
- Si se hubiera implementado de otra forma hubieran surgido otros atributos
- Y cada vez que ésta cambie probablemente los atributos también cambiarán

Atributos

- Así que si se empieza el diseño de una clase por sus atributos...

¡Mal!

Atributos

- **No se trata de teclearlos debajo o de hacerlos privados**
  - Se trata de no plantear la implementación como formas de modificar dichos atributos
- **Si en vez de eso se plantean las acciones que debe realizar el objeto**
  - Unas veces harán falta atributos para darles soporte
  - Pero otras veces resultará que es mejor ***delegar*** en otros objetos

Atributos

- Nunca poner como atributo lo que debería ser una variable local de un método
- El criterio a seguir es:

> ¿Ese dato afecta al comportamiento de otra operación como para que deba recordarlo?

Atributos

- **La siguiente definición aparece en un libro:**

> *Un objeto es un conjunto de datos que opcionalmente puede incluir operaciones para manipular los mismos.*

¿Qué os parece?

¡Lo importante de los objetos son sus responsabilidades!

Recapitulemos

Antes de continuar, un pequeño recordatorio de los aspectos clave vistos hasta ahora.

Recapitulemos

- Los objetos realizan un conjunto de operaciones
- Dichas operaciones las define el tipo del objeto (o tipos)
- La forma de utilizar un objeto es enviando mensajes a través de una referencia
- Cada mensaje tiene una implementación (por ahora)
- Los atributos son elementos auxiliares de dichas implementaciones y les permite actuar en función de los mensajes recibidos previamente

Mantienen el estado del objeto

Mensajes, operaciones, métodos

Criterios

- Un objeto no puede presuponer una secuencia de mensajes determinada
- Cada mensaje debe ser tratado de acuerdo al estado actual del objeto
- No debe haber ninguna secuencia de mensajes que corrompa al objeto
- Un objeto tiene que estar centrado en una funcionalidad

Parámetros

- Cuando a un objeto se le envía un mensaje aquél puede necesitar información adicional para llevarlo a cabo
- Los parámetros son la forma de enviar información junto con el mensaje

```
class Calculadora
{
    public int suma(int a, int b)
    {
        return a + b;
    }
}
```

Opciones y operandos

- Supóngase el siguiente método:

```
class Printer
{
    public void print(String document,
                      int paperSize,
                      boolean color,
                      int resolution)

    {
        ...
    }
}
```

¿Algo raro?

Opciones y operandos

- **En la implementación de una operación se utilizan:**
  - Opciones
  - Operandos
- **Es importante diferenciarlos para diseñar bien las clases**
  - Un operando es un objeto que el método necesita para operar
  - Una opción representa una forma de operar con los operandos

Opciones y operandos

- **¿Cómo diferenciarlos?**
  - Los operandos permanecen estables mientras que las opciones pueden variar a medida que la clase evolucione
  - Un parámetro es una opción si se puede encontrar un valor predeterminado adecuado

- **Según eso, ¿de qué tipo son cada uno de los parámetros del ejemplo anterior?**

```
public void print(String document,
                  int paperSize,
                  boolean color,
                  int resolution)
```

Opciones y operandos

- **Criterio:**
  - Los argumentos de un mensaje deben ser sólo operandos
  - Las opciones son cambios en el estado del objeto que serán tenidos en cuenta por algunos métodos
    - Se indicarán mediante mensajes

Ejercicio

- Según eso, ¿cómo quedaría el nuevo diseño de nuestra clase impresora?

```
class Printer
{
    public void print(String document,
                      int paperSize,
                      boolean color,
                      int resolution)

    {
        ...
    }
}
```

Ejercicio

- Según eso, ¿cómo quedaría el nuevo diseño de nuestra clase impresora?

```
clase Printer
{
    void setPaperSize(int size) { ... }
    void setResolution(int resolution) { ... }
    void setColor(boolean color) { ... }

    void print(String document)
    {
       // actúa en función del estado
    }
}
```

Ejercicio

- **Ejemplo:**

```
class Test {
    public static void main(String[] args) {
        Printer printer = new Printer();
        printer.setColor(true);
        printer.print("myDocument.doc");
    }
}
```

Ventajas de separarlos

Opciones y operandos

- **Más fácil invocar el mensaje**
  - Todo lo que no se haya establecido previamente con las opciones se trata con los valores por omisión
- **A medida que la clase evolucione se podrán añadir mas opciones, pero los clientes de la clase no se verán afectados**

Opciones y operandos

- Con la primera versión de impresora:

```
class Empleado
{
    public void trabaja(Printer impresora) {
        impresora.print("documento.doc", "A4",
                        false, 600);

    }
}
```

¿Qué pasa si se añade un modo de ahorro de tinta?

Opciones y operandos

```
void print(String document,
           String paperSize,
           boolean color,
           int resolution,
           boolean econoMode)
{ . . . }
```

Habría que cambiar todas las invocaciones del método

Opciones y operandos

- Con la versión mejorada no hubiera habido problemas

```
class Empleado
{
    public void trabaja(Printer impresora) {
        impresora.print("documento.doc");
    }
}
```

Opciones y operandos

- Incluso aunque pasasen a utilizar las nuevas opciones

```
class Ejemplo {
    public static void main(String[] args) {
        Empleado pepe = new Empleado();
        Printer impresora = new Printer();

        impresora.setEconoMode(true);
        pepe.trabaja(impresora);
    }
}
```

Opciones y operandos

- **Nótese que en la primera versión el cliente debía conocer todas las opciones permitidas por la impresora**
  - Aunque no las utilizase
- **En la segunda forma al cliente se le pasa la impresora en el estado adecuado para que éste la utilice como se desea**

Ejemplo en Java

```
class MiComponente extends JPanel
{
  Componente interno;

  public void draw(Graphics g)
  {
    Graphics2D g2 = (Graphics2D) g;

    g2.setRenderingHint(
         RenderingHints.KEY_ANTIALIASING,
         RenderingHints.VALUE_ANTIALIAS_ON);

   interno.draw(g);

    g2.setRenderingHint(
         RenderingHints.KEY_ANTIALIASING,
         RenderingHints.VALUE_ANTIALIAS_DEFAULT);
  }
}
```

Opciones y operandos

- **Mezclar en los parámetros de un mensaje tanto opciones como operandos es síntoma de programación estructurada encubierta**
  - Significa no aplicar aún el concepto de estado de un objeto

Estado de un objeto

Estado de un objeto

- **Definición**
  - Es la combinación de los valores de los atributos de un objeto
- **Los valores de los atributos determinarán el comportamiento del objeto ante los distintos mensajes que reciba**

Pero...

- **Hemos dicho que los atributos son un mero soporte para la implementación de las operaciones del objeto**

¿Se puede definir algo tan importante (el estado del objeto) en función de algo tan irrelevante (sus atributos)?

Estado de un objeto

- **El estado de un objeto determina la respuesta del objeto ante cada uno de los mensajes**
  - Determina qué mensajes no son permitidos
  - Determina el modo de actuar de los permitidos
- **Guardar el estado de un objeto supone guardar la suficiente información para que al recuperarla se obtengan exactamente las mismas respuestas ante la misma secuencia infinita de mensajes tanto del original como del objeto recuperado**

Nótese como se ha definido el estado en función del comportamiento (no de los atributos)

Estado de un objeto

- **Generalmente basta con guardar los atributos del objeto**
  - (De ahí la razón de la primera definición)
- **Aunque habrá casos en los que no valga solamente con eso**
  - Especialmente, en los casos de objetos que guardan referencias a otros objetos
    - Delegan en ellos, les envían mensajes (Es la clave de la OO)
    - Habrá que determinar qué relaciones es necesario guardar para preservar el estado del objeto

En definitiva el estado es algo mas que los valores de los atributos...

Métodos de acceso

Encapsulación

- **Normalmente no se debe dar acceso a los datos internos de un objeto**

Criterio de encapsulación

```
class Persona
{
     private int edad;

     ...
}
```

Métodos de acceso

```
class Persona
{
    private int edad;

    int getEdad()
    {
       return edad;
    }

    void setEdad(int edad)
    {
       this.edad = edad;
    }
}
```

Métodos de acceso

- **Pero...**

¿No estamos en las mismas sólo que con más código?

Ventajas

- Permite controlar las modificaciones del atributo
- Ocultar los atributos permite modificar la implementación sin afectar a los clientes

Ventajas

- Permite controlar las modificaciones del atributo

```
class Persona
{
    private int edad;
    ...
     void setEdad(int edad)
    {
       if (edad >= 0 && edad <= 150) {
         this.edad = edad;
       }
    }
}
```

Los errores ya no se propagan sin control: se sabe quién está corrompiendo los datos

Ventajas

- O mejor aún:

```
class Persona
{
    private int edad;
    ...
     void setEdad(int edad)
     {
        if (edad < 0 || edad > 150) {
          throw new IllegalArgumentException("...");
        }
        this.edad = edad;
     }
}
```

Ojo con la programación defensiva. Muchas veces se torna en programación ofensiva. Normalmente, si una operación no puede realizarse, valdrá más que falle en vez de, silenciosamente, no hacer nada; el cliente de la clase supondrá que la edad sí se ha cambiado: sólo estamos difiriendo el error (y haciéndolo mucho más difícil de detectar, depurar y solucionar).

http://c2.com/cgi/wiki?OffensiveProgramming

Ventajas

- **Ocultar los atributos permite modificar la implementación sin afectar a los clientes**

```
class Persona
{
    private Fecha fechaDeNacimiento;
    ...
     int getEdad()
    {
        ...
    }
}
```

Ahora el atributo edad es redundante (se puede calcular a partir de la fecha de nacimiento)

Criterio de independencia

Criterio de independencia

- **Una de las ventajas fundamentales de la orientación a objetos es poder trabajar con objetos con independencia de la implementación de éstos**

criterio fundamental OO

Cuanto menos se sepa de la implementación de un objeto menos afectarán sus cambios

Criterio de independencia

- **Por eso una clase no debe saber de otra**
  - Ni su implementación
  - Ni sus atributos
  - Ni sus modos de operación
- **Lo único importante de un objeto son los mensajes que puede recibir**

Estado concreto y estado abstracto

- **Si en una clase `Temperatura` al pedir los datos en Celsius están en Fahrenheit... ¿se cambia el estado del objeto?**
  - No, porque no cambia sus respuestas, es decir, no cambia su comportamiento

- **Hay que diferenciar entre:**
  - Estado concreto
  - Estado abstracto

Se define en función de los atributos

Éste, dependiendo de la implementación de dicha clase, sí podría variar

Se define en función de las respuestas a los mensajes

Cuando hablamos de estado siempre nos referimos al estado abstracto

Diseño de clases

Diseño de clases

- **Lo difícil del DOO es averiguar qué clases poner y con qué mensajes cada una**
- **Hay muchos libros que dedican muchos capítulos a intentar explicar cómo extraer las clases**
- **El problema es que los métodos que dan son demasiado abstractos o genéricos**
  - Que es la única forma de solucionar algo irresoluble de forma sistemática
- **Aquí simplemente una pequeña guía práctica**
  - Y genérica, por supuesto

Lo que no funciona

- **Basarse en objetos reales**
- **No sirve extraer nombres y verbos del lenguaje natural**
  - «El ascensor tiene una puerta que se abre y se cierra.»
- **Además puede expresarse de varias maneras**
  - «En cada piso hay una puerta que da paso al hueco del ascensor.»

El qué y el quién

- Muchos de estos métodos intentan sacar clases sin saber todavía qué tienen que hacer

  (Difícil)

- Otra forma es centrarse en las tareas a realizar: el *qué*

  Pero... ¡un momento! ¿No es eso acaso programación estructurada? ¿Todo este rollo para esto? ¡Me siento engañado!, ¡falso converso!

- Es cierto, pero faltaba un matiz: no separarlo jamás del *quién*

Guía práctica

1. **Elegir una tarea a realizar: el qué**
    a) Decidir qué tipo de objeto la hará
        - Aquí está la diferencia: en programación estructurada se pasaba directamente a la codificación
        - Primero se determina quién la tiene que hacer: la respuesta tendrá que ser un objeto

Guía práctica

1. **Elegir una tarea a realizar: el qué**

    b) ¿Puede hacerla alguna de las clases que ya tenemos?

    - (Naturalmente, cumpliendo todos los criterios vistos: independencia de implementación y estados, una única funcionalidad... y los que quedan)
    - Si es así, añadir la nueva operación a dicha clase
    - En caso contrario ya tenemos nueva clase (¡y con una responsabilidad concreta!)

Guía práctica

2. **Decidida la clase y la responsabilidad, tocan los mensajes**
   - Dividir la responsabilidad en operaciones
     - Atómicas, para que se puedan combinar
   - Parámetros con información imprescindible
     - El resto, opciones
   - Sacar partido de que los objetos «recuerdan» lo que se les ha mandado
     - Crear objetos inteligentes

Usabilidad también en el diseño y programación OO:
pensar en los clientes de la clase

Guía práctica

3. Implementación de las operaciones (los métodos)

- Cualquiera que no tenga errores
  - Usar constructores para iniciar el objeto en un estado válido
  - Dejar siempre en un estado válido al salir de un método
  - Comprobar parámetros
- No preocuparse por la eficiencia hasta el final
  - Generalmente es suficientemente rápida

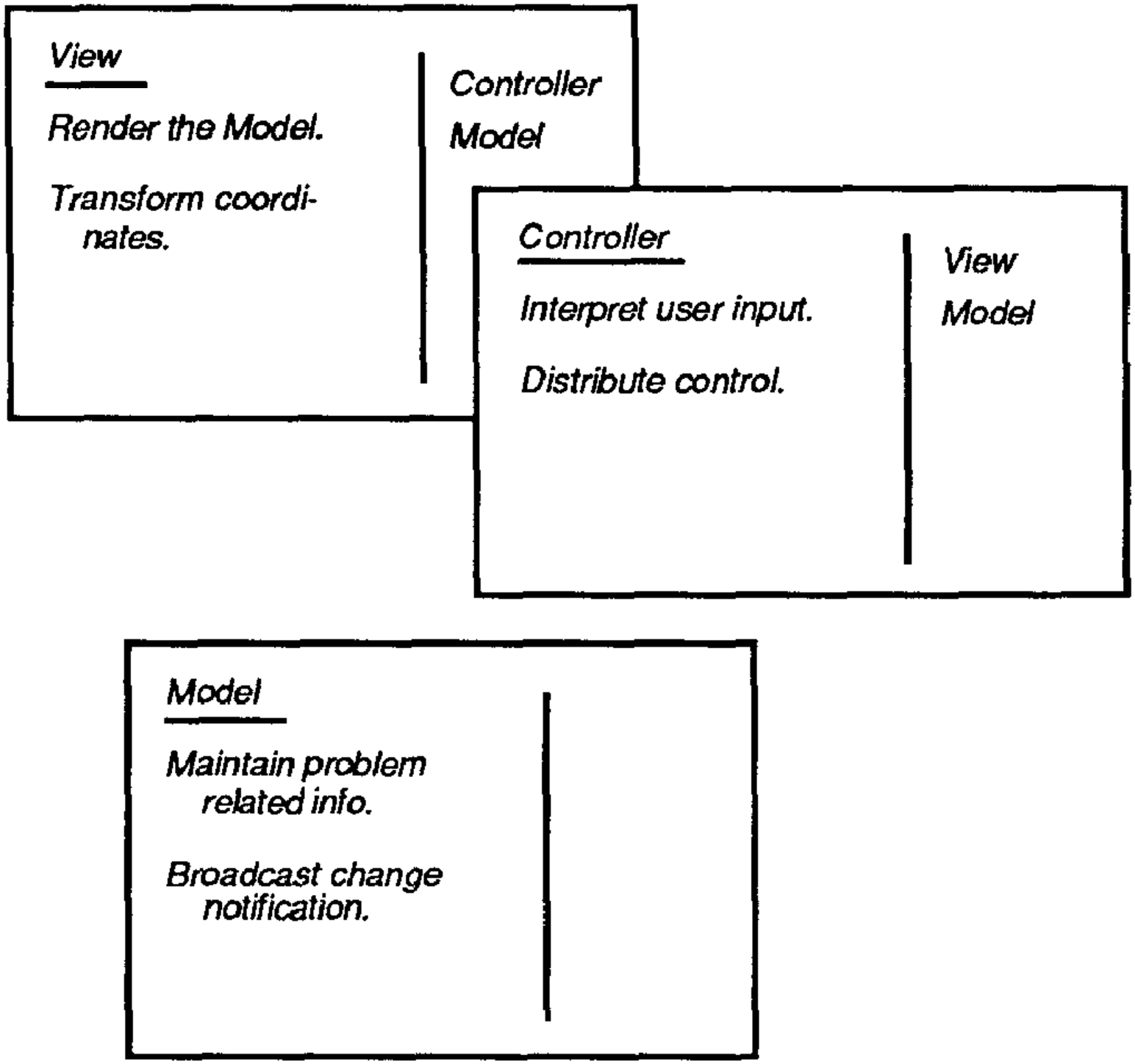
    Normalmente los compiladores, las máquinas virtuales... hacen su trabajo bastante mejor de lo que lo haríamos nosotros

  - No optimizar algo que puede desaparecer al factorizar

Diseño de clases y patrones de diseño

- Los patrones de diseño pretenden dar respuesta a los dos primeros pasos para determinados problemas de diseño habituales

Fichas CRC


View
Render the Model.
Transform coordi-
nates.
Controller
Model
Controller
Interpret user input.
Distribute control.
View
Model
Model
Maintain problem
related info.
Broadcast change
notification.

Fichas CRC

Clase-Responsabilidades-Colaboraciones

- **Inventadas por Ward Cunningham y Kent Beck en 1989**
- **Son una técnica más que puede ser útil para identificar clases**
  - Especialmente en las fases tempranas del diseño conceptual
- **El acrónimo viene de «clase, responsabilidades y colaboraciones»**

(o colaboradores)

Formato

Clase

Responsabilidades

Colaboraciones

Formato

- **Hay pequeñas variantes, pero en esencia es el de la figura anterior**
  - El nombre de la clase
    - Opcionalmente, se puede indicar su superclase
  - Responsabilidades
    - Una serie de sentencias breves que describen las responsabilidades de la clase
      - «Conocer las figuras que contiene»
      - «Controlar la reproducción de las canciones»

Formato

- Colaboraciones
  - El nombre de las clases con las que colabora para llevar a cabo dichas responsabilidades

Más sobre fichas CRC

- **A Laboratory For Teaching Object-Oriented Thinking**
  - Ward Cunningham y Kent Beck
  - http://c2.com/doc/oopsla89/paper.html
- **CRC Cards: An Agile Thinking Tool**
  - Rebecca Wirfs-Brock
  - http://www.informit.com/articles/article.aspx?p=1391208