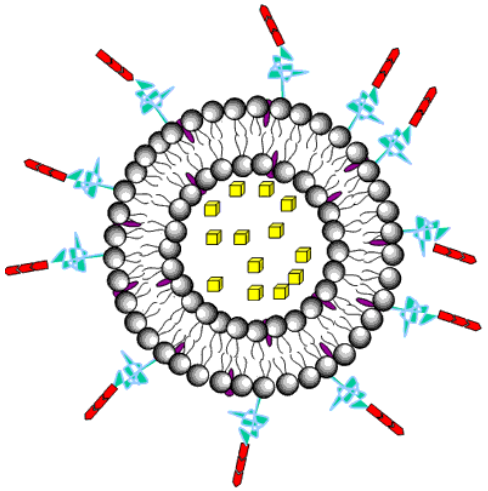
FACULTAD
DE INGENIERIA
Universidad de Buenos Aires



Clases
(construcción)

Carlos Fontela
cfontela@fi.uba.ar

A3F

Temario

- Implementación de clases
  - Atributos
  - Métodos y propiedades
  - Constructores
  - Excepciones
- Diseño contractual
- TDD o diseño guiado por las pruebas

Dijimos…

POO parte de las entidades del dominio del problema

Que son objetos con comportamiento

¿Cómo?

=> Implementando clases

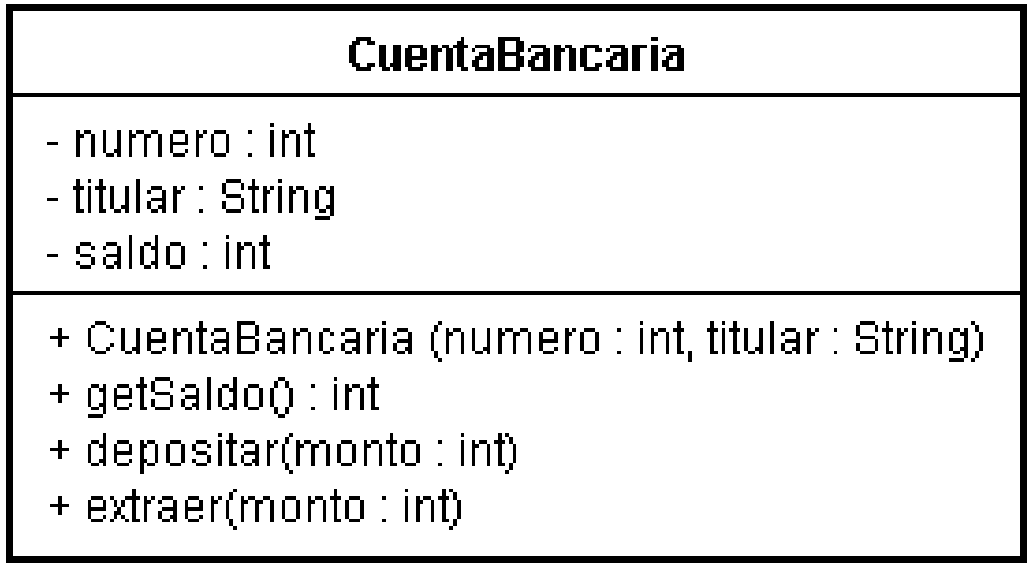
Implementación de clases (1)

- Clase = tipo definido por el programador
  - No los llamamos “abstractos”: en POO eso es otra cosa
  - Ampliar el lenguaje
  - Se definen estructura y operaciones
- Ocultamiento: cliente necesita conocer interfaz
  - No necesita conocer aspectos internos (implementación)
  - Riesgos de la falta de ocultamiento
    - Impedir evolución
    - Violación de restricciones

Implementación de clases (2)

En UML:

- Lo que se indica con (-) está oculto: es “privado” de cada objeto
- Lo que se indica con (+) lo pueden usar los clientes: es “público”



Implementación de clases: ¿cómo?

- Varios caminos de diseño
  - Modelo contractual
  - Desarrollo guiado por las pruebas
- Son complementarios y no excluyentes

Modelo contractual (1)

Una clase es provista por un proveedor a un cliente en base a un “contrato”

Bertrand Meyer y “diseño por contrato”

Modelo contractual (2)

El contrato se evidencia por

- Firmas de métodos y propiedades
- Precondiciones de métodos y propiedades
- Postcondiciones de métodos y propiedades
  - Incluyendo resultados obtenidos
  - Incluyendo casos de excepción
- Invariantes de la clase
  - Restricciones que siempre cumplen todas las instancias

Cuenta bancaria: firmas de métodos

Constructor: (para crear e inicializar)

CuentaBancaria (int numero, String titular);

Métodos (y propiedades):

cuenta.depositar (int monto);

cuenta.extraer (int monto);

int cuenta.getSaldo( );

String cuenta.getTitular( );

int cuenta.getNumero( );

Cuenta bancaria: atributos

Son variables internas de cada objeto, que sirven para mantener el estado de los mismos

Para una cuenta bancaria:

String titular;

int numero;

int saldo;

Podría haber más, que descubramos más adelante

Cuenta bancaria: precondiciones

CuentaBancaria (int numero, String titular);

- Precondición 1: titular no debe valer null ni referenciar una cadena vacía
- Precondición 2: numero > 0

cuenta.depositar (int monto);

- Precondición 1: cuenta no debe valer null (debe referenciar un objeto)
- Precondición 2: monto > 0

Tarea: definir precondiciones para los otros métodos

Cuenta bancaria: postcondiciones

CuentaBancaria (int numero, String titular);

Postcondición 1: se creó un objeto de la clase CuentaBancaria, con el número y el titular indicados

Postcondición 2 (alternativa): si titular es null o referencia una cadena vacía, se arroja una excepción de tipo IllegalArgumentException

cuenta.depositar (int monto);

Postcondición 1: el saldo de la cuenta aumentó en el valor del monto

Postcondición 2 (alternativa): si monto < 0 , se arroja una excepción de tipo IllegalArgumentException

Cuenta bancaria: invariantes

Son las restricciones en los valores de los atributos, válidas para todas las instancias

Son postcondiciones de todos los métodos (públicos), incluyendo el constructor

Corolario: el constructor debe dejar a la instancia creada en un estado válido

Invariantes de CuentaBancaria:

saldo >= 0

titular != null

titular != “”

numero > 0

Implementación de la clase (1)

```
public class CuentaBancaria {
    // atributos:
    private int numero;
    private String titular;
     private double saldo;
    // propiedad (sólo lectura):
    public double getSaldo ( ) {
          return saldo;
    }
```

Implementación de la clase (2)

```
// métodos:
    public void depositar (double monto) {
     if (monto < 0)
          throw new IllegalArgumentException ( );
     saldo += monto;
    }
    public void extraer (double monto) {
     if (monto > saldo)
          throw new IllegalArgumentException ( );
     saldo -= monto;
    }
```

Referencia this

Objeto o referencia “this”

```
public void depositar (double monto) {
  if (monto < 0)
        throw new IllegalArgumentException ( );
  this.saldo = this.saldo + monto;
}
```

Invocación con el “objeto actual”

```
c.depositar (2000);
```

“this” referencia al “objeto actual”

Implementación de la clase (3)

```
// constructor:
public CuentaBancaria (int numero, String titular) {
 if (numero <= 0 || titular == null || titular == “”)
        throw new IllegalArgumentException ( );
    this.numero = numero;
    this.titular = titular;
    this.saldo = 0;
}
} // fin de la clase
```

Modelo contractual en la práctica

- Precondiciones
  - Si no se cumplen, lanzamos una excepción
- Postcondiciones
  - Si no se cumplen, podríamos lanzar una excepción
  - Pero la prueba unitaria es un mejor camino
    - Veremos más adelante
- Invariantes
  - Son postcondiciones permanentes

Excepciones: lanzamiento

Las excepciones son objetos

Se crean y se lanzan hacia el módulo invocante

Sintaxis:

```
throw new ClaseException ( );
```

Ya vimos cómo capturarlas

Veremos formalmente excepciones más adelante

Los objetos deben saber cómo comportarse

Ya lo dijimos:

- Diferencia más importante con programación estructurada

Corolarios:

- Deben manejar su propio comportamiento
- No debemos manipular sus detalles desde afuera

En vez de:

```
cuenta.setSaldo ( cuenta.getSaldo ( ) + monto );
```

Hacemos:

```
cuenta.depositar(monto);
```

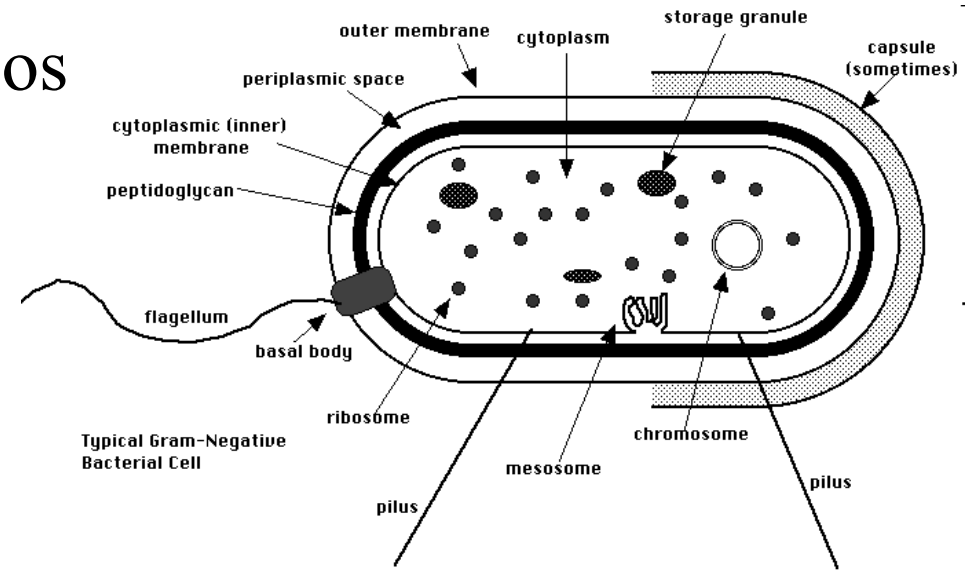
Encapsulamiento

Alan Kay, creador de Smalltalk, y del término "Programación Orientada a Objetos" se basó en sus conocimientos de bacterias

La membrana de una bacteria nos aísla de la complejidad interna

La bacteria interactúa con el mundo a través de su interfaz

- Respondiendo a estímulos
- Realizando acciones



¿Y en C#?

Hay propiedades como elemento del lenguaje

Uso:

```
Console.WriteLine ( c.Saldo );
```

Definición:

```
public double Saldo {
  get {
       return saldo;
  }
}
```

Constructores (más)

Se usan para inicializar

Si no se implementan hay uno por omisión

- Para que toda clase tenga el suyo

Pero se pueden programar otros, como hicimos

- En este caso, deja de existir el default

Debería dejar al objeto en un estado válido

- => debe cumplir con los invariantes
- El default no es seguro

Visibilidad

Atributos, propiedades y métodos privados

Sólo se pueden usar desde dentro de la clase en que están definidos

Atributos, propiedades y métodos públicos

Se los puede usar desde cualquier lado

Visibilidad de paquete (default)

Se los puede usar desde su mismo paquete

Paquetes y fuentes

- Toda clase está en un paquete
  - En Java existe el paquete “default”, pero no es recomendable usarlo
- En C#, se enmarca el código de la clase en una cláusula “namespace”:
  - namespace carlosFontela.cuentas { … }
- En Java, cada clase pública va en un archivo fuente separado
- El paquete se indica en una cláusula “package”
  - package carlosFontela.cuentas;

Diseño guiado por pruebas

Test-Driven Development = Test-First + automatización + refactorización

Test-First:

- Escribir código de pruebas antes del código productivo

Automatización:

- Las pruebas deben expresarse como código, que pueda indicar si todo sale bien de manera simple y directa
- El conjunto de pruebas debe poder ir creciendo
- Las pruebas deben correrse por cada cambio

Refactorización: mejora de calidad del diseño sin cambio de funcionalidad

TDD: frameworks de pruebas automatizadas

Ejemplo de CuentaBancaria con JUnit

Ver en Eclipse

Llegamos a la misma clase

En .NET existe NUnit

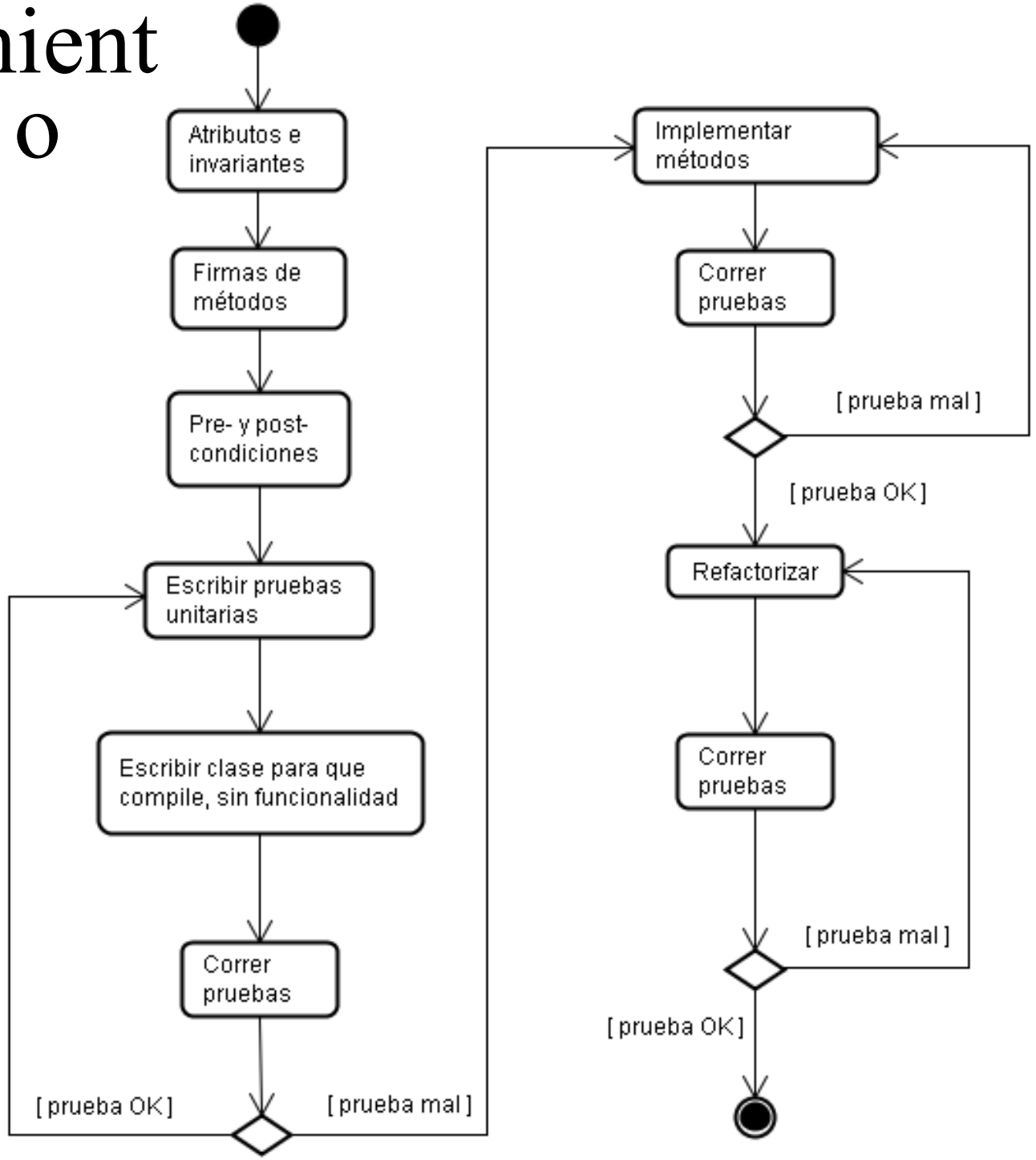
En Smalltalk, SUnit

Muy importantes en refactorización: ¿lo vemos?

¿Cómo implementamos las clases?

- Ayudarse por los dos caminos
  - Modelo contractual
  - TDD
- Pruebas automatizadas sirven para
  - TDD
  - Invariantes y postcondiciones del modelo contractual
- Otra herramienta: aserciones en el modelo contractual
  - Es redundante y preferimos las otras

Proced imient o



Atributos y propiedades: ojo con las apariencias

≠

No todos los atributos tienen “getters” y “setters”

- Sólo los necesarios

Hay propiedades que no corresponden a atributos

- String.length() => ¿tiene que haber un atributo?
- Complejo.getModulo( ); => propiedad calculable

Noción: (propiedad == “atributo conceptual”)

=> Los atributos conceptuales deberían estar implementados como propiedades

Atributos de clase

Supongamos que necesitamos que el número de cuenta fuera incremental

Solución:

Agregar un atributo “numeroAcumulado” que mantenga un único valor para la clase

Eso es un **atributo de clase**

En Java y C# “de clase” se dice “static”

OJO: Los miembros de clase no se heredan

Ejercicio: cambiar CuentaBancaria para que número de cuenta sea incremental

Lenguajes: clases utilitarias

En Java y .Net toda función debe ir en una clase

- Aunque trabajemos sin objetos

Usual agrupar funciones

Clase utilitaria

- Agrupa métodos estáticos y/o constantes en una clase
- Esta clase no se puede instanciar

Ejemplos

- Arrays.sort(v);
- double x = Math.pow(2,3);
- double p = Math.PI;

Lenguajes: sobrecarga

```
public CuentaBancaria (int numero, String titular) {
    this ( numero, titular, 0);
}
public CuentaBancaria (int numero, String titular, int saldoInicial) {
    if (numero <= 0 || titular == null || titular == “”)
            throw new IllegalArgumentException ( );
    this.numero = numero;
    this.titular = titular;
    this.saldo = saldoInicial;
}
```

Java: no todo es un objeto

Las instancias de tipos primitivos no son objetos

- int, long, double, char, boolean
- Y se manejan por valor (no por referencia)

Hay "encajonamiento" o "autoboxing"

- Integer x = 4; // significa Integer x = new Integer(4);

Arreglos "primitivos" no son objetos

- int [ ] v = new int [4];
- Pero se manejan por referencia
- Tamaño se establece en tiempo de ejecución
  - Una vez establecido, no puede variar
- Hay chequeo de rangos
- Tipo de los elementos definido estáticamente

C#: casi todo es un objeto

- Los tipos primitivos son tipos por valor
  - Pero hay encajonamiento
- Los arreglos son instancias de System.Array
- Hay muchos tipos por valor
  - Pero se manejan como clases
  - Y se pueden definir por el programador
  - Con algunas restricciones

Smalltalk: todo son objetos y mensajes

No hay variables que no referencien objetos

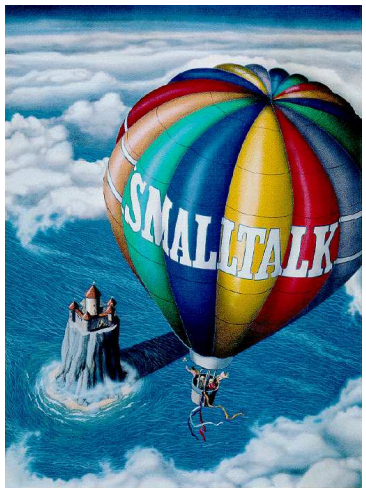
Las clases son objetos

Los métodos son objetos

Las estructuras de control son métodos

=> Modelo de objetos “puro”

Sólo objetos y mensajes



Claves

Clases se implementan en base a un modelo cliente-proveedor

Las clases son tipos definidos por el programador

- Que representan entidades del dominio del problema

Diseño con dos modelos

- Contratos
- TDD

Las pruebas deben ser automatizadas

Lecturas opcionales

Object-Oriented Software Construction, Bertrand Meyer

- Está en la biblioteca
- Especialmente capítulos 7, 8, 11 y 12

Test Driven Development: By Example, Kent Beck

- No está en la Web ni en biblioteca

Code Complete, Steve McConnell, Capítulo 6: “Working Classes”

- No está en la Web ni en biblioteca

Implementation Patterns, Kent Beck, Capítulos 3 y 4: “A Theory of Programming” y “Motivation”

- No está en la Web ni en biblioteca

Orientación a objetos, diseño y programación, Carlos Fontela 2008, capítulo 4 “Construcción de clases”

Qué sigue

Delegación y herencia

Polimorfismo

Smalltalk