



Clases (construcción)

Carlos Fontela
cfontela@fi.uba.ar



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 “abstractos”: en POO 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

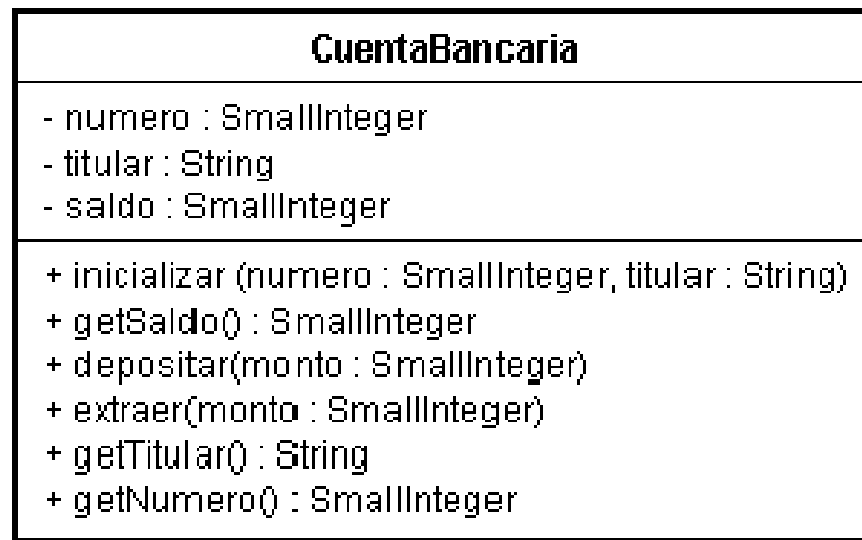


Implementación de clases (2)

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?



Mensajes => Métodos u operaciones

Varios caminos de diseño

Modelo contractual

Desarrollo guiado por las pruebas

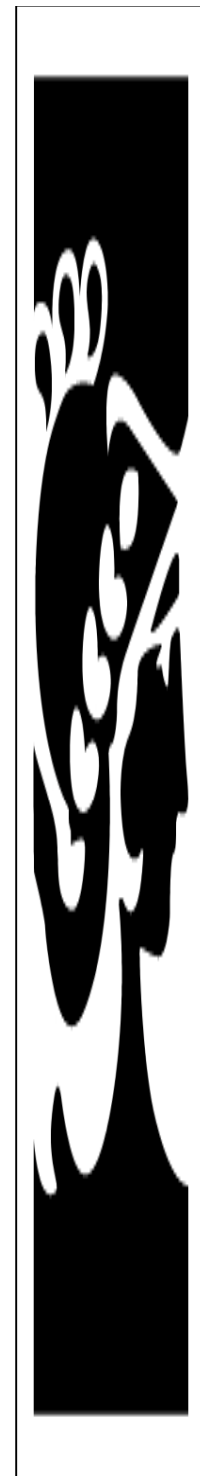
Son complementarios y no excluyentes



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

Precondiciones de métodos

Postcondiciones de métodos

Incluyendo resultados obtenidos

Incluyendo casos de excepción

Invariantes de la clase

Restricciones que siempre cumplen todas las instancias



Firmas de métodos

Constructor: (para crear e inicializar) => ver luego

CuentaBancaria inicializarConNumero: numero conTitular: titular

Métodos (uso):

cuenta depositar: monto

cuenta extraer: monto

cuenta getSaldo

cuenta getTitular

cuenta getNumero

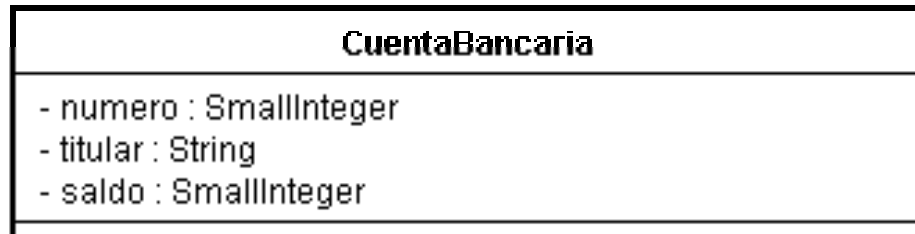
Ojo que en Smalltalk no se definen los tipos



Cuenta bancaria: atributos

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

Para una cuenta bancaria:



Ojo que en Smalltalk no se define el tipo antes de usarlos

Podría haber más, que descubramos más adelante



Cuenta bancaria: invariantes

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

Son postcondiciones de todos los métodos, incluyendo el constructor

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

Invariantes de CuentaBancaria:

saldo ≥ 0

titular $\neq \text{nil}$

titular $\neq \text{''}$

numero > 0



Cuenta bancaria: precondiciones

CuentaBancaria inicializarConNumero: numero conTitular:
titular

Precondición 1: titular no debe valer nil ni referenciar una
cadena vacía

Precondición 2: numero > 0

cuenta depositar: monto

Precondición 1: cuenta no debe valer nil (debe referenciar
un objeto)

Precondición 2: monto > 0

Tarea: definir precondiciones para los otros métodos



Cuenta bancaria: postcondiciones

CuentaBancaria inicializarConNumero: numero conTitular:
titular

Postcondición 1: se creó un objeto de la clase

CuentaBancaria, con el número y el titular indicados

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

cuenta depositar: 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 Error



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: postcondiciones permanentes



Excepciones: lanzamiento

Las excepciones son objetos

Se crean y se lanzan hacia el módulo invocante

Sintaxis:

`ClaseException new signal.`

`ClaseException new signal: 'un texto'.`

Ya vimos cómo capturarlas

El texto lo obtenemos con el método `messageText`

Veremos formalmente excepciones más adelante



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 SUnit

Llegamos a la misma clase

En Java existe JUnit

En .NET , NUnit

Muy importantes en refactorización

Los van a ver en la práctica



¿Cómo implementamos las clases?

Ayudarse por los dos caminos

Modelo contractual

TDD

Pruebas automatizadas sirven para

TDD

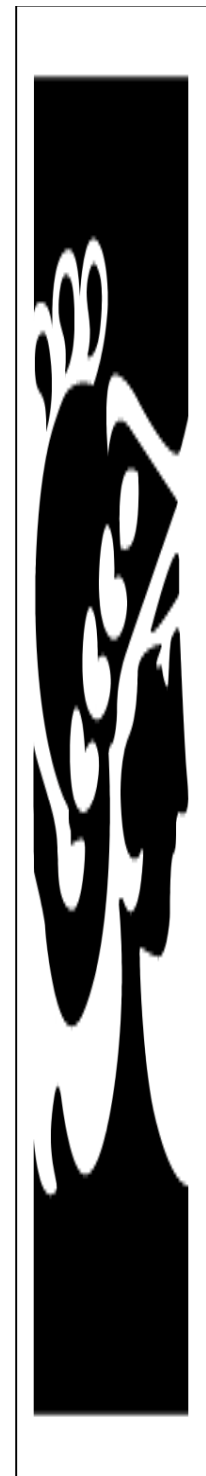
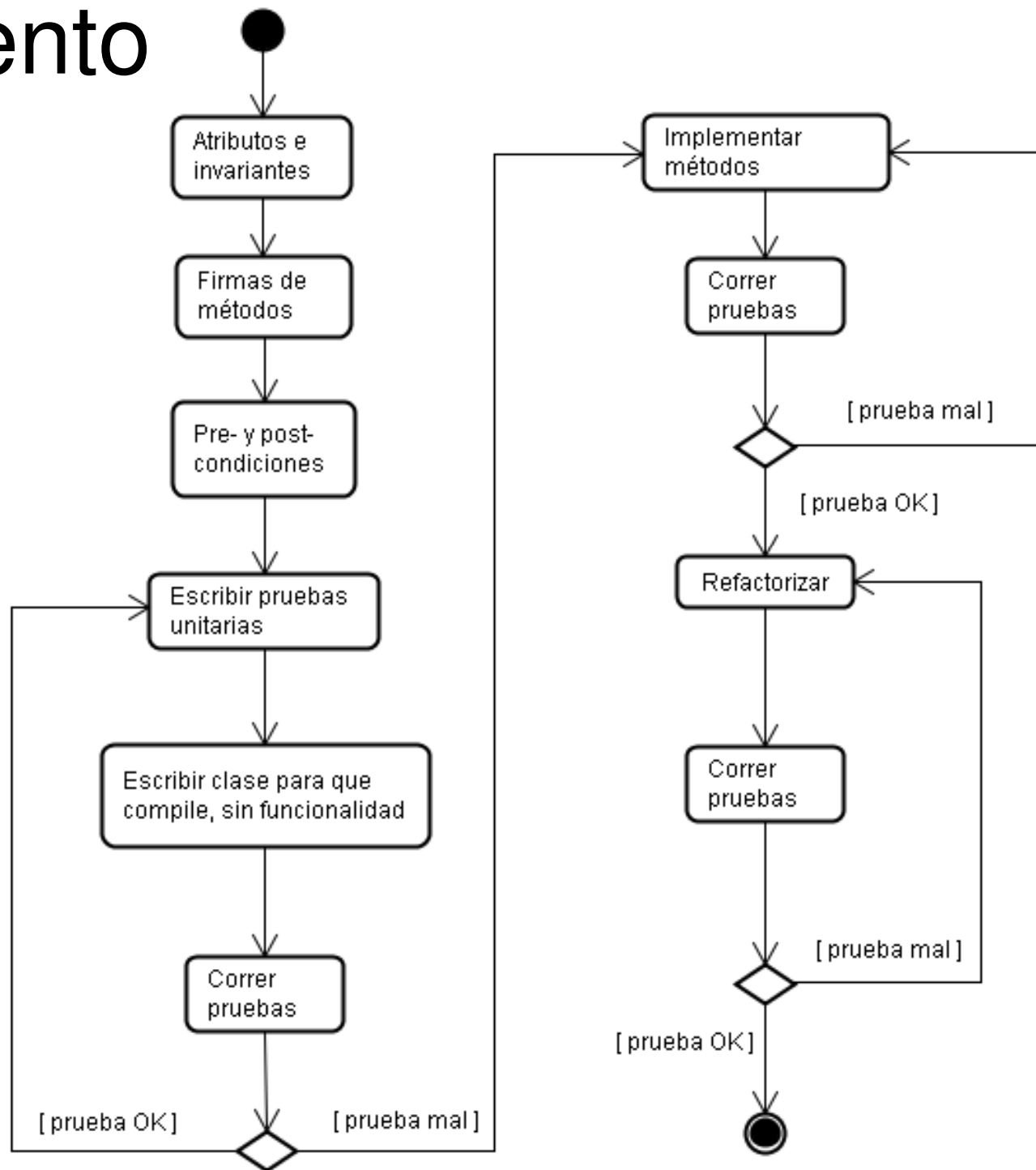
Invariantes y postcondiciones del modelo contractual

Otra herramienta: aserciones en el modelo contractual

Es redundante y preferimos las otras



Procedimiento



Pruebas unitarias (1)

Clase de la prueba: debe descender de
TestCase

Métodos de prueba: deben empezar con “test”
y no tener parámetros

TestCase subclass: #PruebaCuentaBancaria

instanceVariableNames: ‘cuenta’

classVariableNames: “

poolDictionaries: “

category: ‘MisPruebas’



Pruebas unitarias (2)

PruebaCuentaBancaria >> setUp

```
cuenta := CuentaBancaria new
  inicializarConNumero: 1234
  conTitular: 'Juan Pérez'
```

Esto sirve para inicialización repetida: no es una prueba

Nota: la sintaxis “NombreClase >> método” es convencional en Smalltalk



Pruebas unitarias (3)

```
PruebaCuentaBancaria >> testDeposito
```

```
    cuenta depositar: 500.
```

```
    self assert: (cuenta getSaldo = 500)
```

```
PruebaCuentaBancaria >> testExtraccionConSaldo
```

```
    cuenta depositar: 1200.
```

```
    cuenta extraer: 500.
```

```
    self assert: (cuenta getSaldo = 700)
```

```
PruebaCuentaBancaria >> testExtraccionSinSaldo
```

```
    self should: [ cuenta extraer: 1000 ] raise: Error
```



Pruebas unitarias (4)

PruebaCuentaBancaria >> testTitularVacio

self should:

```
[cuenta := CuentaBancaria new  
  inicializarConNumero: 1234 conTitular:]
```

raise: Error.

PruebaCuentaBancaria >> testNumeroNoPositivo

self should:

```
[cuenta := CuentaBancaria new  
  inicializarConNumero:-2  
  conTitular:'Juan Pérez']
```

raise: Error.



Pruebas unitarias (5)

PruebaCuentaBancaria >> testTitularNil

self should:

```
[cuenta := CuentaBancaria new  
  inicializarConNumero: 1234 conTitular:nil]
```

raise: Error.

PruebaCuentaBancaria >> testNumeroNil

self should:

```
[cuenta := CuentaBancaria new  
  inicializarConNumero:nil  
  conTitular:'Juan Pérez']
```

raise: Error.



Implementación de la clase (1)

Object subclass: #CuentaBancaria

instanceVariableNames: 'numero titular saldo'

classVariableNames: ''

poolDictionaries: ''

category: 'Cuentas'



Implementación de la clase (2)

CuentaBancaria >> getSaldo

^saldo

CuentaBancaria >> getTitular

^titular

CuentaBancaria >> getNumero

^numero



Implementación de la clase (3)

CuentaBancaria >> depositar: monto

(monto < 0) ifTrue: [Error new signal.] .

saldo := saldo + monto

CuentaBancaria >> extraer: monto

(monto > saldo) ifTrue: [Error new signal.] .

saldo := saldo - monto



Referencia self

Objeto “self”

depositar: monto

(monto < 0) ifTrue: [Error new signal.] .

self saldo := self saldo + monto

El mensaje se envía al “receptor”

cuenta depositar: 2000. “self referencia al receptor”

Lo mismo pasaba con los assert y los should
de las pruebas



Implementación de la clase (4)

CuentaBancaria >>

inicializarConNumero: num conTitular: tit

```
( (num isNil) | (tit isNil) | (tit = "") | (num <= 0) )  
  ifTrue: [Error new signal] .
```

```
numero := numero.
```

```
titular := titular.
```

```
saldo := 0
```



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

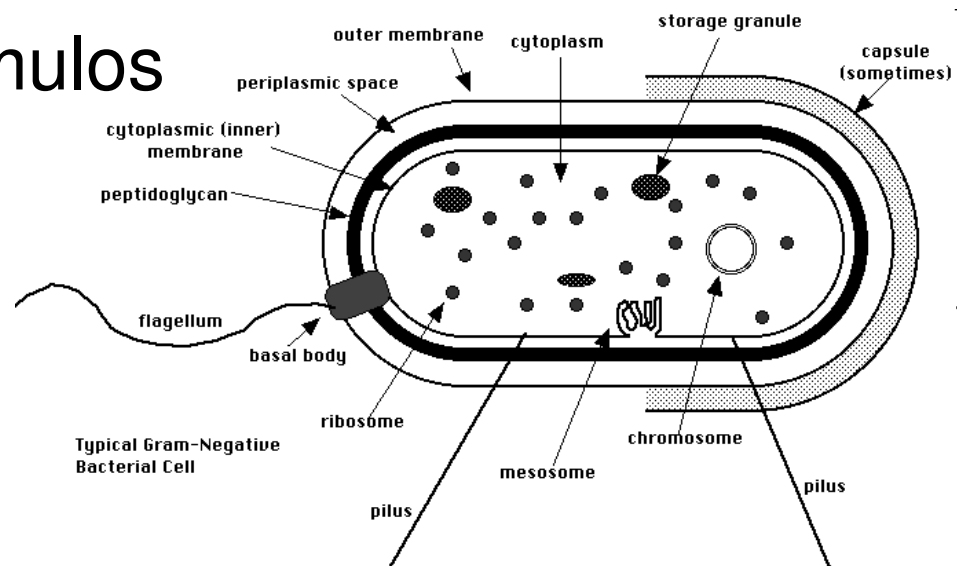
Alan Kay, creador de Smalltalk, y del término “Programación Orientada a Objetos” se basó en sus conocimientos de bacterias

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



Constructores o inicializadores

No existen en Smalltalk: sólo está el método de clase new

Debería dejar al objeto en un estado válido

=> debe cumplir con los invariantes

El “new” no es seguro

Hay un método “initialize”, que se puede redefinir

De todas maneras, no tiene parámetros

⇒ Tampoco es seguro

Por eso definimos el método:

```
CuentaBancaria >> inicializarConNumero: numero conTitular: titular
```

Se invoca

```
cuenta := CuentaBancaria new
```

```
inicializarConNumero: 1234 conTitular:'Juan'
```



Visibilidad (conceptos)

Importante para garantizar ocultamiento de implementación

Atributos y métodos privados

Sólo los puede usar el objeto receptor en su clase

Atributos y métodos públicos

Se los puede usar desde cualquier lado

Atributos y métodos protegidos

Sólo los puede usar el objeto receptor en su clase y en las clases derivadas



Visibilidad en Smalltalk

Todos los métodos son públicos

Todos los atributos son protegidos

No privados, se pueden acceder desde una subclase

Aunque se recomienda considerarlos privados

Hay convenciones para hacer métodos y atributos privados

Pero no se puede forzar

Sólo como aviso



Atributos de clase

Supongamos que necesitamos que el número de cuenta fuera incremental

Solución:

Agregar un atributo “ultimoNumero” que mantenga un único valor para la clase y todas sus instancias

En Smalltalk se declaran en “classVariableNames”

```
classVariableNames: 'ultimoNumero'
```

OJO: no confundir con los atributos del objeto clase

Algo propio de Smalltalk, porque las clases son objetos



Ejercicio: número auto-incremental

Hay que cambiar las pruebas de los inicializadores con número, eliminando dos de ellas

Y hay que agregar al menos una:

PruebaCuentaBancaria >> pruebaAutoIncremental

```
cuenta1 := CuentaBancaria new
```

```
  inicializarConTitular:'Juan Pérez'.
```

```
cuenta2 := CuentaBancaria new
```

```
  inicializarConTitular:'Ana García'.
```

```
assert: ( (cuenta2 getNumero) – (cuenta1 getNumero) = 1)
```



Caso de atributo de clase

Escribimos prueba

Nos aseguramos de que no pase

Implementamos la solución

Nos aseguramos de que corra

CuentaBancaria
- ultimoNumero : <u>SmallInteger</u> - numero : SmallInteger - titular : String - saldo : SmallInteger
+ inicializar(titular : String) + getSaldo() : SmallInteger + depositar(monto : SmallInteger) + extraer(monto : SmallInteger) + getTitular() : String + <u>getProximoNumero()</u> : SmallInteger + getNumero() : SmallInteger



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

unString size => ¿tiene que haber un atributo?

numeroComplejo getModulo => propiedad calculable

Noción: (propiedad = “atributo conceptual”)

=> Los atributos conceptuales deberían estar implementados como propiedades



Smalltalk: todo son objetos vivos y mensajes

No hay variables que no referencien objetos

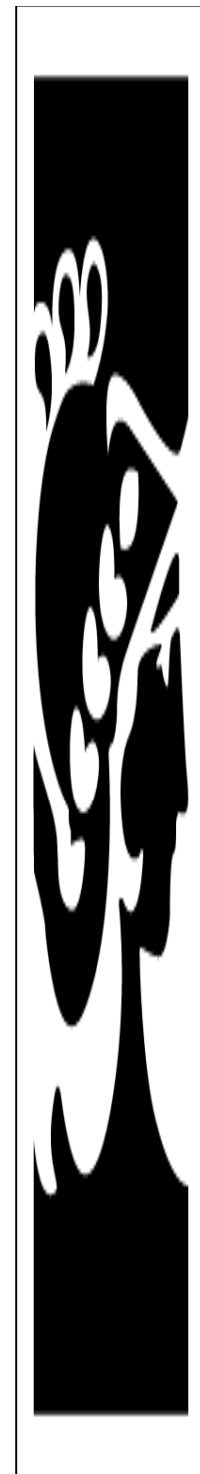
Las clases son objetos

El IDE es un objeto, y sus partes también

No hay estructuras de control: son mensajes

=> 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

Implementación con dos modelos

Contratos

TDD

Las pruebas deben ser automatizadas



Lectura obligatoria

Apunte de Pruebas (ver sitio de la materia)



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, herencia, polimorfismo

Excepciones y temas conceptuales

Temas varios

