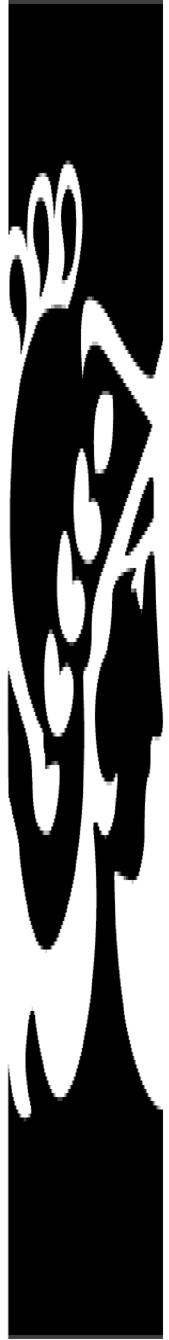
FACULTAD
DE INGENIERIA
Universidad de Buenos Aires



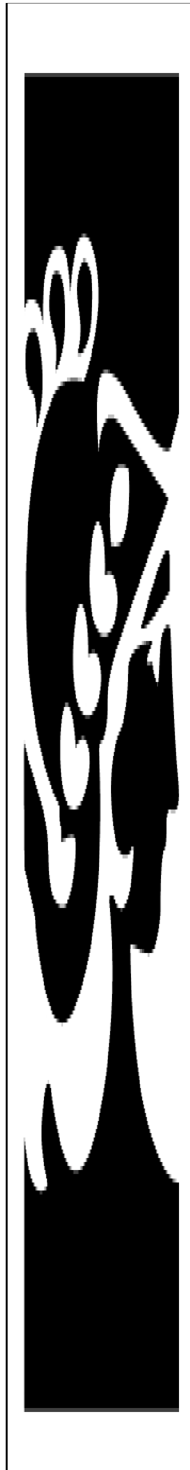
Reutilización con delegación, herencia y polimorfismo

Carlos Fontela
cfontela@fi.uba.ar

A3F

Temario

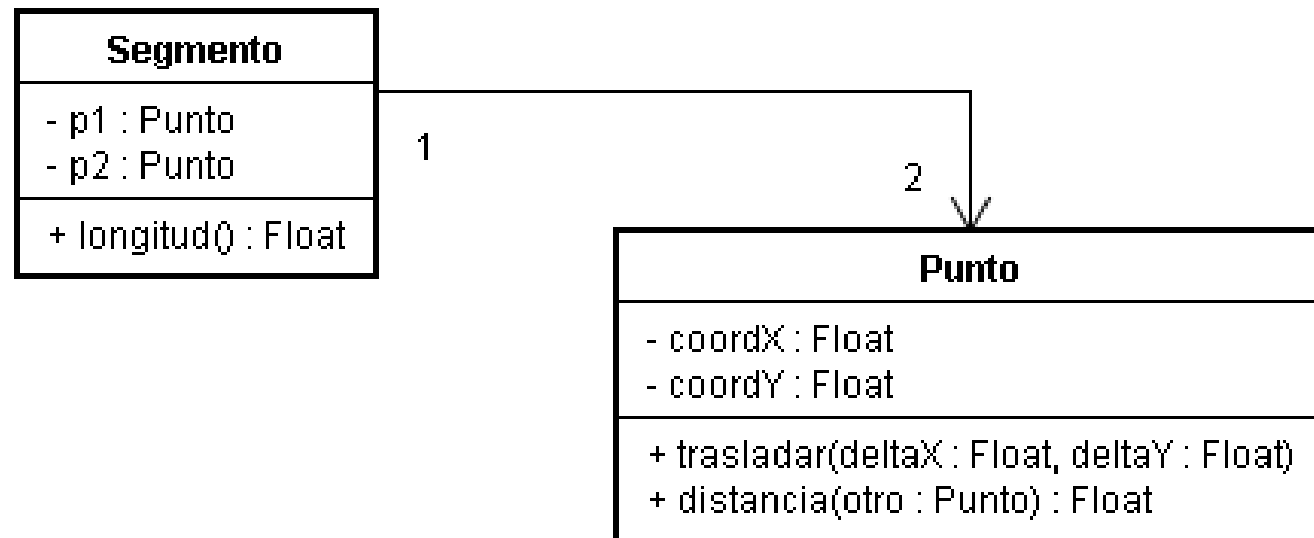
- Delegación
- Herencia
- UML: clases, paquetes, secuencias
- Cuándo usar herencia y cuándo delegación
- Redefinición
- Clases abstractas y métodos abstractos
- Polimorfismo

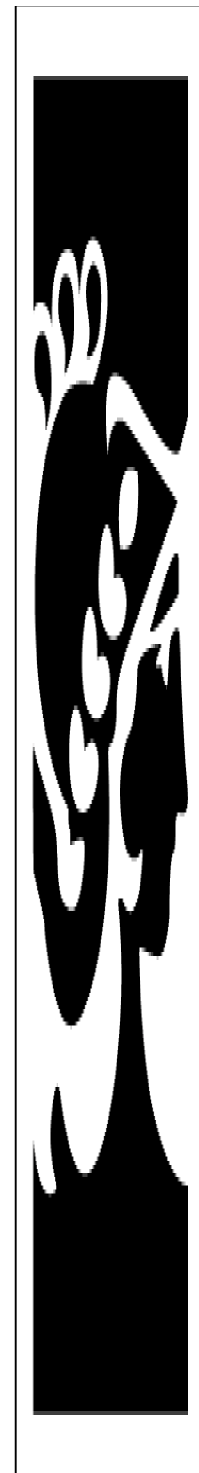


Delegación (1)

Un objeto contiene referencias a otros objetos y les delega comportamiento

```
Segmento >> longitud
    ^ (p1 distancia: p2).
```





Delegación (2)

- Es una forma de reutilización
- Mediante el envío de un mensaje a otro objeto
  - “cliente” pide un servicio a un “servidor”
- El otro objeto se preocupa de cómo implementa el método

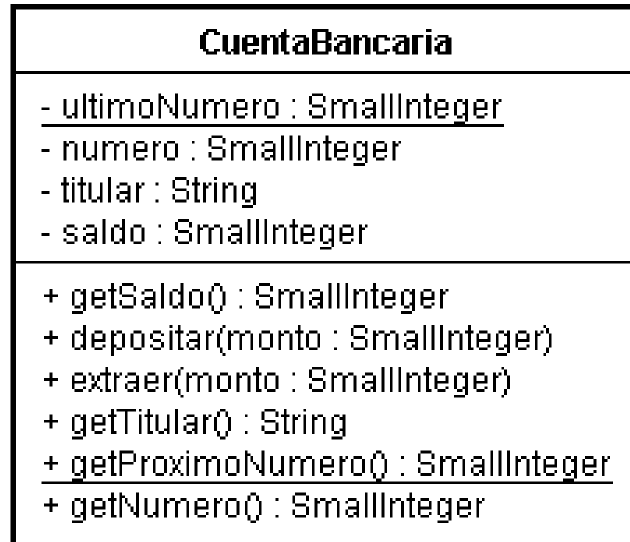
- Evitar los objetos omnipotentes
  - Con muchas responsabilidades
  - El comportamiento debe mantenerse junto con la información que utiliza

Agregación vs. Composición

Composición

- Las partes no son independientes del todo
- El objeto contenido no puede estar contenido en más de un contenedor
- Eliminación del todo implica la de las partes
- Rombo lleno en UML

Clases en UML



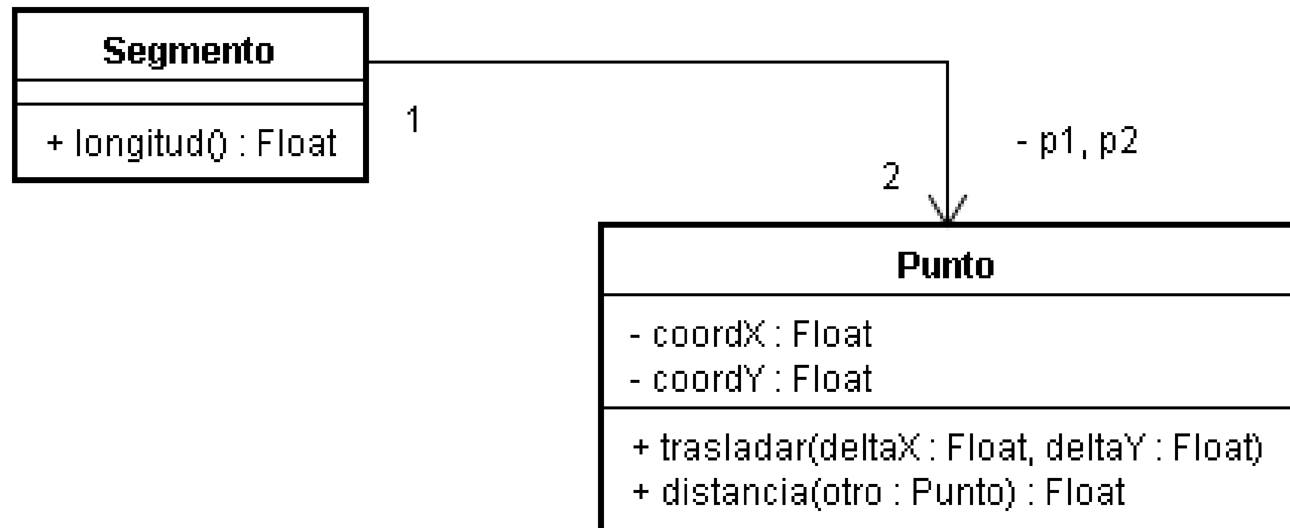
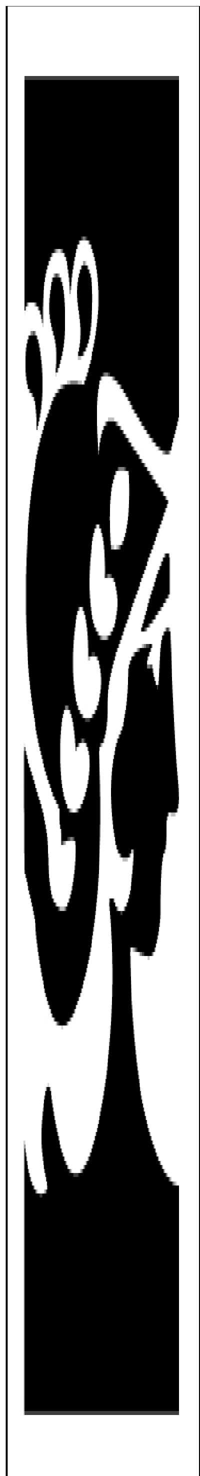


Diagrama de secuencias UML

:Cajero
SistemaCajeros
Base de datos Clientes
Base de datos Cuentas
1: ingresoTarjeta()
2: VerificacionTarjeta()
3: pedidoClave()
4: verificacionClave()
<<create>>
5: crearTransaccion()
:Transaccion
6: muestraMenu()
7: pedidoSaldo()
7.1: pedidoSaldo()
7.2: grabar()
<<destroy>>
8: terminar()
8.1: terminarGrabar()

2c2011



A3F

Ejercicio

En un banco existen varios mostradores.

Cada mostrador puede atender cierto tipo de trámites y tiene una cola de clientes, que no puede superar un número determinado para cada cola.

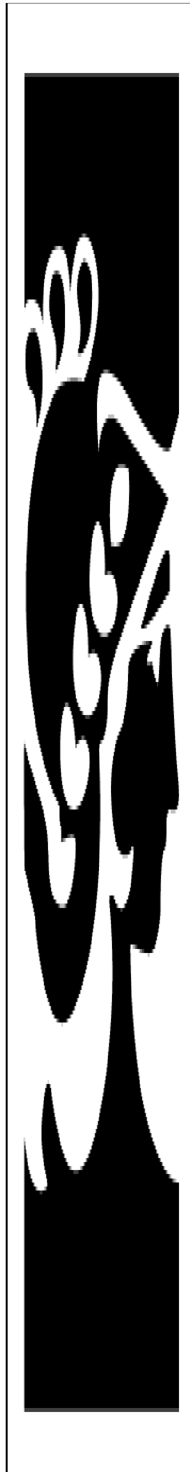
Además hay una cola general del banco en la cual se colocan todos los clientes cuando las colas de los mostradores están completas.

Cada cliente concurre al banco para realizar un solo trámite.

Un trámite tiene un horario de creación y un horario de resolución.

Se pide:

1) Implementar el método mostrador>>atiende:unTramite, que devuelve true o false indicando si el tramite se puede atender o no en el mostrador; note que el tipo de trámite correspondiente a unTramite tiene que coincidir con alguno de los tipos de trámite que atiende el mostrador.



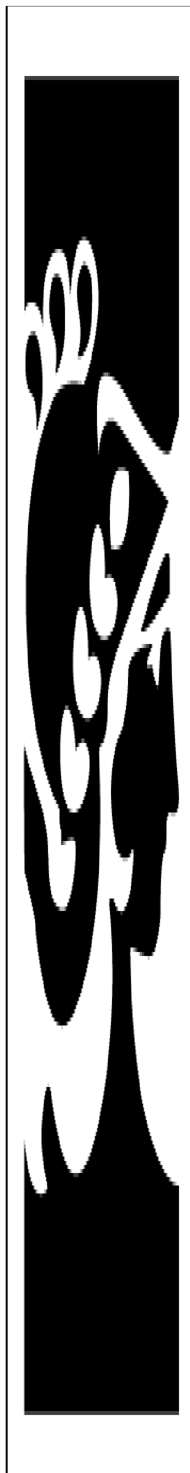
2) Implementar el método banco>>mostradoresQueAtienden:unTramite, que retorna la colección de todos los mostradores que atienden ese trámite.

3) Implementar el método banco>>mejorMostradorPara:unTramite, que retorna el mostrador con la cola más corta con espacio para al menos una persona más y que atienda ese trámite; si ningún mostrador tiene espacio, retorna nil.

4) Implementar el método banco>>atender:unCliente; cuando llega un cliente al banco se lo ubica en el mostrador que atienda el trámite que el cliente requiere, que tenga espacio y la menor cantidad de clientes esperando; si no hay lugar en ningún mostrador el cliente debe permanecer en la cola general de espera del banco.

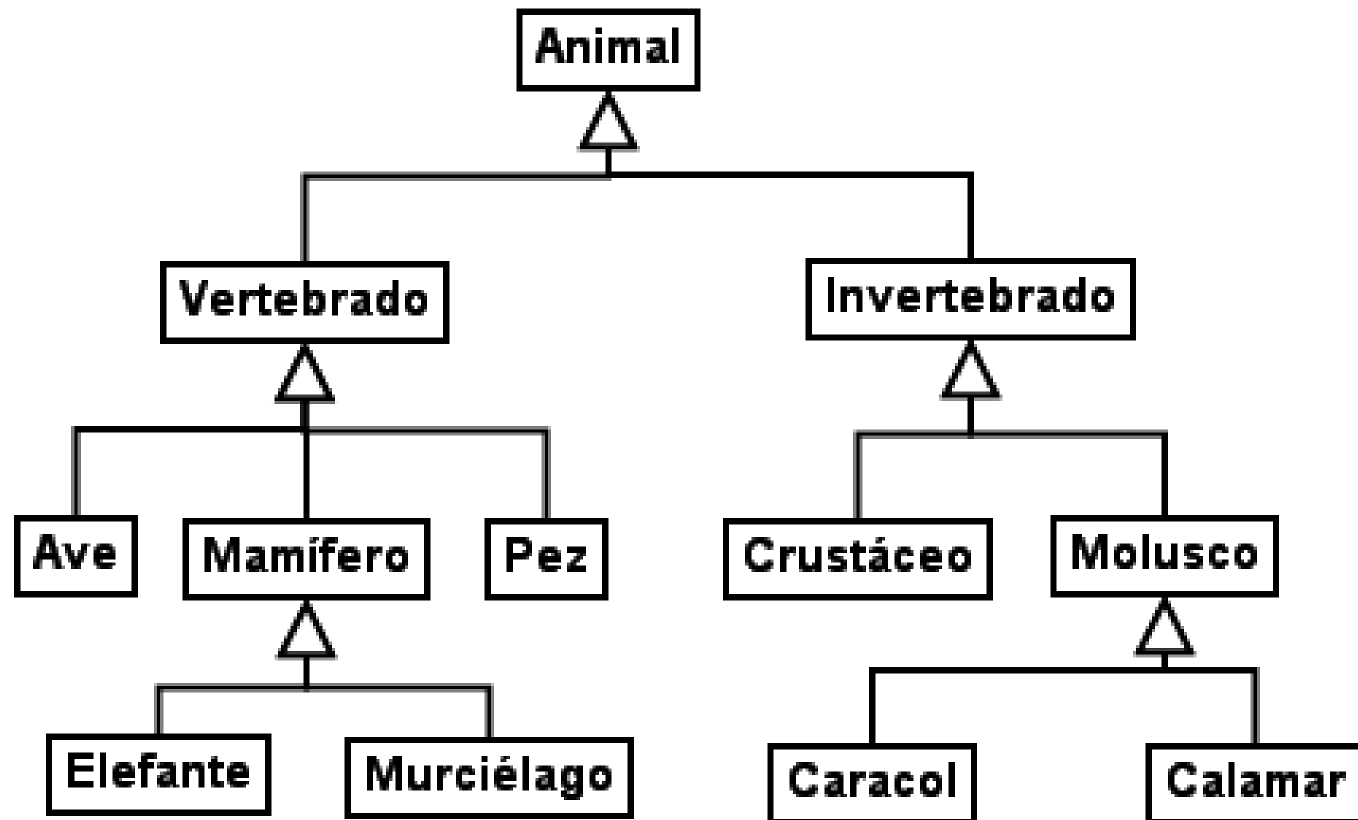
5) Implementar el método mostrador>>atenderPrimero; debe desencolar al primer cliente de la cola y atender su trámite, lo cual implica asignarle la hora de resolución al trámite del cliente.

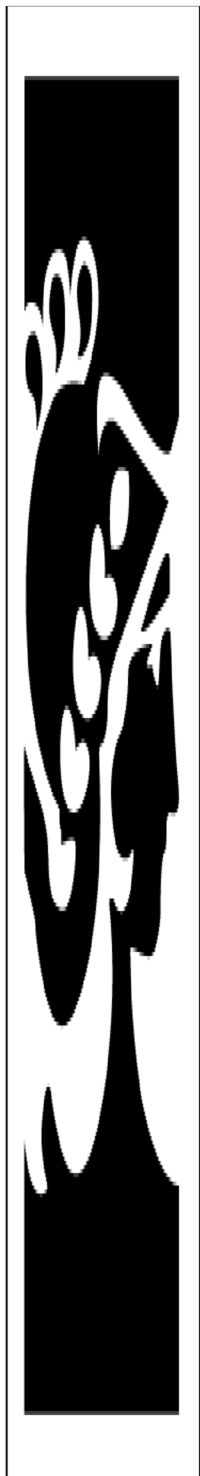
6) Implementar el método banco>>siguienteClientePara:unMostrador; debe elegir de la cola general del banco, el primer cliente que necesite realizar un trámite que unMostrador pueda atender; si no existe tal cliente el método retorna nil.



Taxonomías (1)

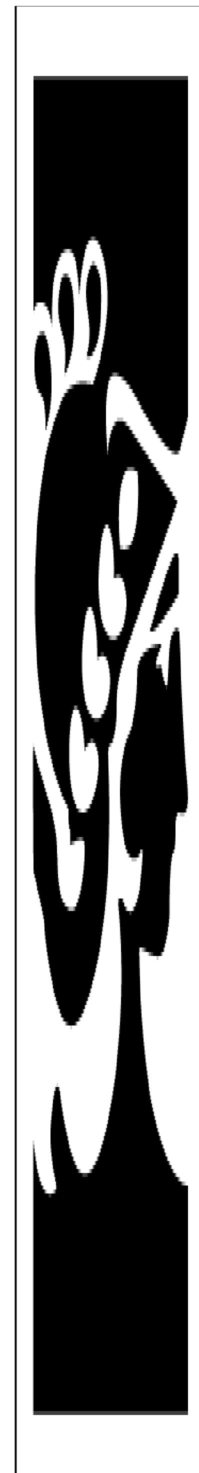
Relaciones "es un"





Taxonomías (2)

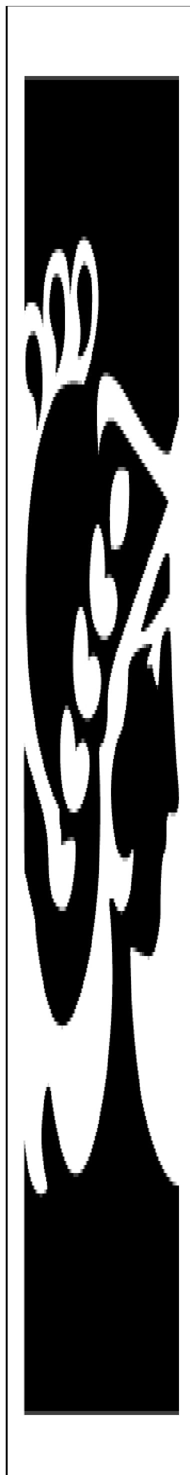
- FiguraPlana
  - Elipse
    - Circunferencia
  - Cuadrilátero
    - Paralelogramo
      - Rectángulo
        - Cuadrado
      - Rombo
    - Trapecio
  - Triángulo
    - TRectángulo



Taxonomías (3)

Observamos

- Las clasificaciones no tienen por qué ser completas, pero sí excluyentes
- Algunas de las clases del árbol pueden no tener instancias: clases abstractas
- La ubicación de una clase en la jerarquía se establece por la relación “es un”
- Cada clase hereda comportamiento y estructura de su ancestro

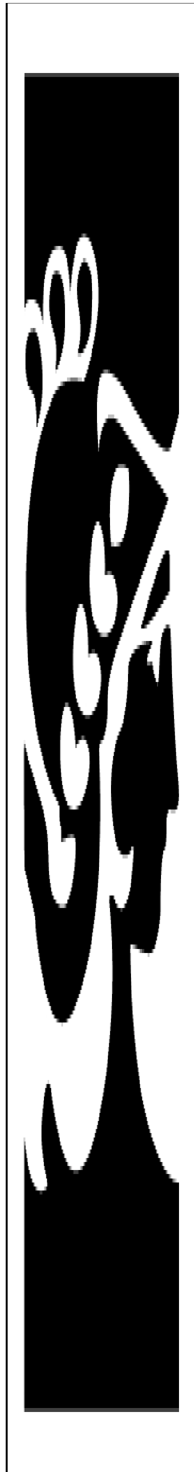


Herencia en Smalltalk

Figura subclass: #Elipse

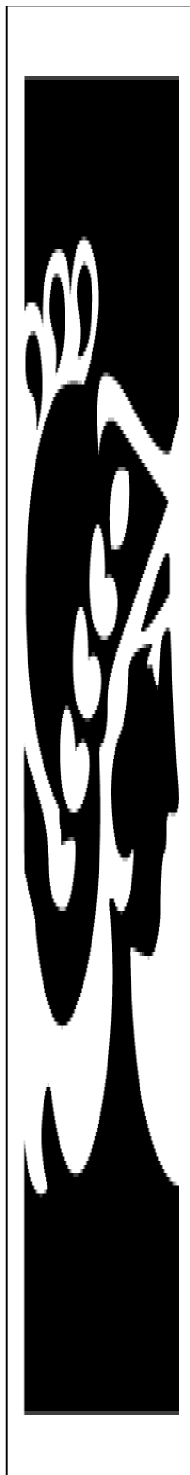
Elipse tiene, por lo menos:

- los mismos atributos de Figura
- los mismos métodos de Figura
- puede agregar atributos y métodos
- puede redefinir métodos



Jerarquía de raíz única

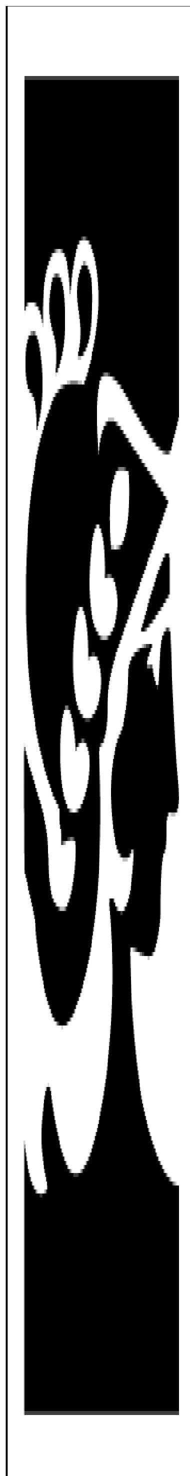
- Clase Object es madre de todas
  - Por eso hicimos
  - Object subclass: #CuentaBancaria
  - En Pharo, ProtoObject
- Atributos y métodos comunes a todas las clases
- Otras importantes consecuencias



A3F

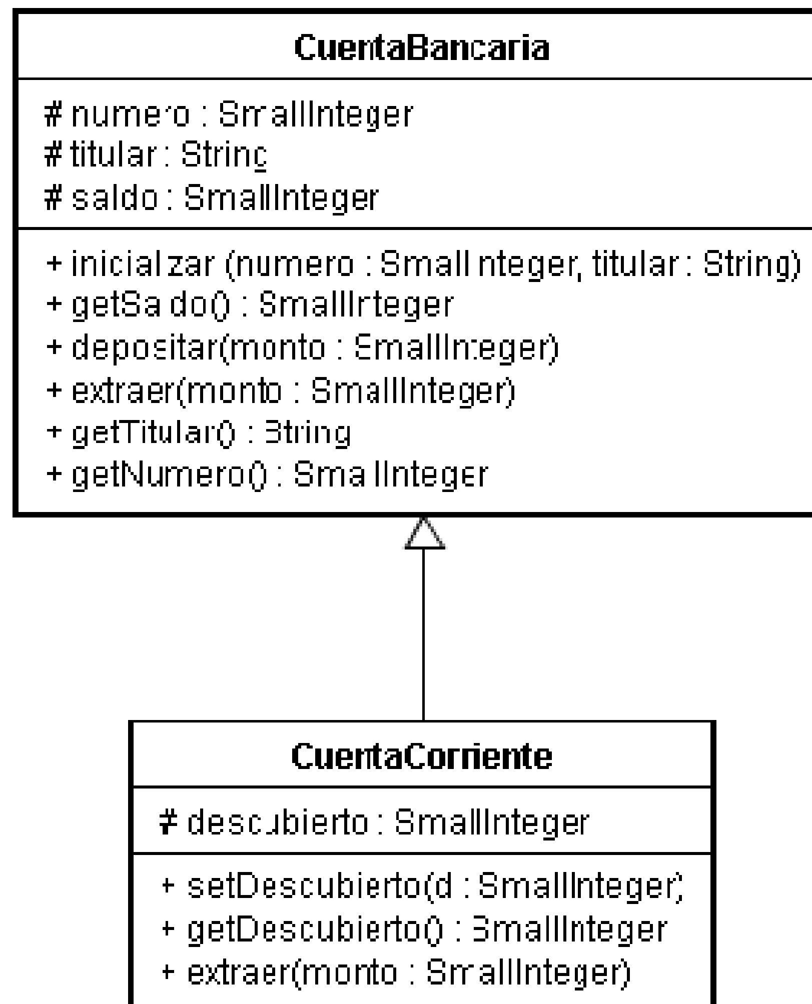
Inicializadores y herencia

- Receta
  - Llamar al inicializador del ancestro al principio del inicializador propio



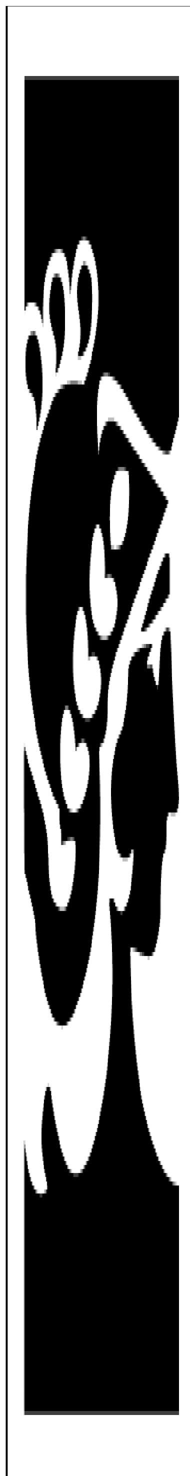
Redefinición (1)

Se puede volver a definir un método en una clase descendiente:



Redefinición (2)

- Debe preservar la semántica (significado)
- Obligatoria
  - Si la implementación debe ser diferente
  - Caso de extraer en CuentaBancaria
- Optativa
  - Razones, en general, de eficiencia
  - Caso de longitud de Elipse
- Los métodos deben tener la misma firma



Herencia y diseño por contrato

Invariantes de clase

Deben ser al menos los mismos de la clase ancestro

Precondiciones de métodos

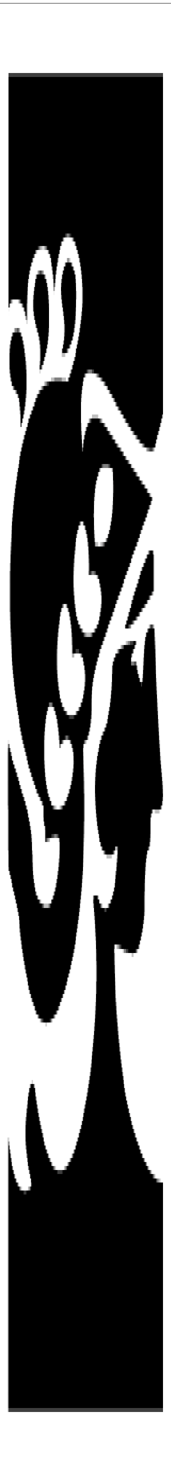
No pueden ser más estrictas en una subclase de lo que son en su ancestro

Postcondiciones de métodos

No pueden ser más laxas en una subclase de lo que son en su ancestro

Excepciones

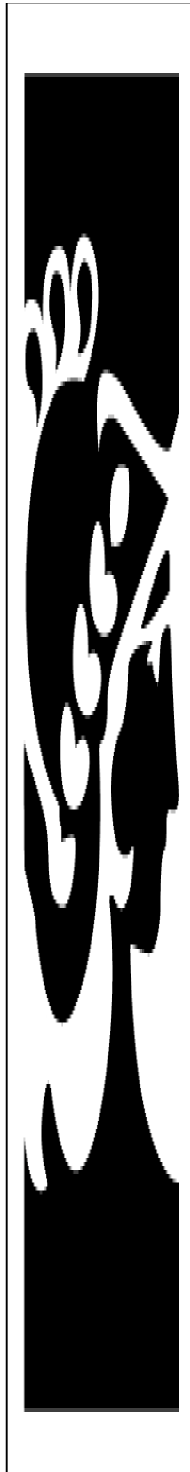
Un método debe lanzar los mismos tipos de excepciones que en la clase ancestro, o a lo sumo excepciones derivadas de aquéllas



A3F

Delegación vs. Herencia (1)

- Herencia: relación “es un”
- Composición/agregación:
  - “contiene”
  - “hace referencia”
  - “es parte de”
- Mito: en POO todo es herencia
- Mal ejemplo: Stack en Java 1.0/1.1
  - ¡una pila no es un vector!
- Herencia si se va a reutilizar la interfaz
  - Stack es un mal ejemplo



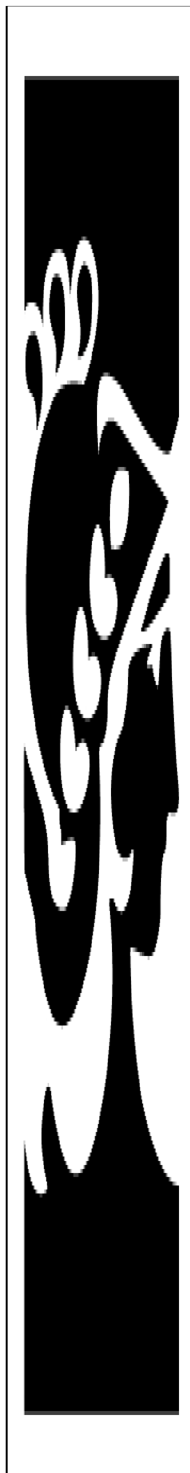
Delegación vs. Herencia (2)

Herencia

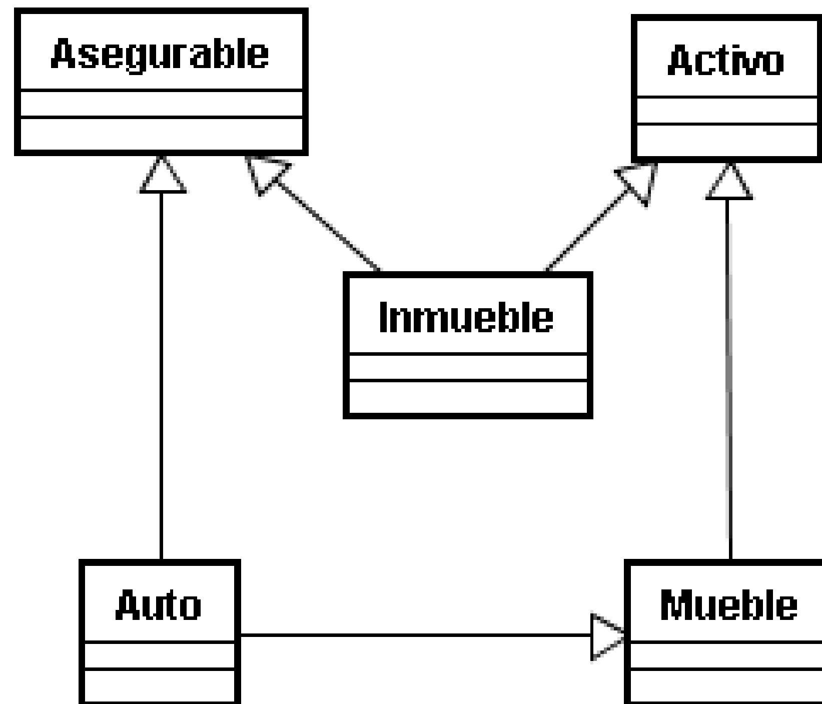
- Cuando se va a reutilizar la interfaz tal como está

Delegación

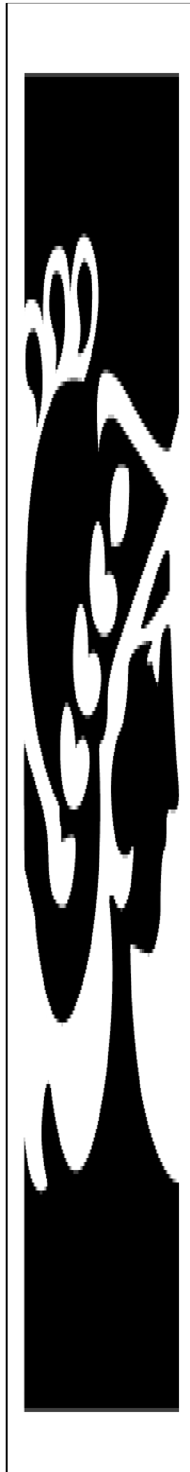
- Cuando se va a reutilizar sin mantener la interfaz



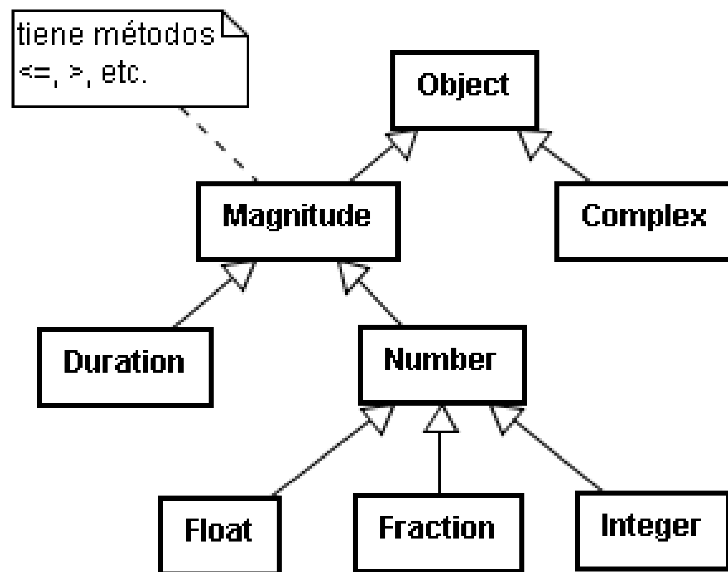
Herencia múltiple (C++, Python, Eiffel)



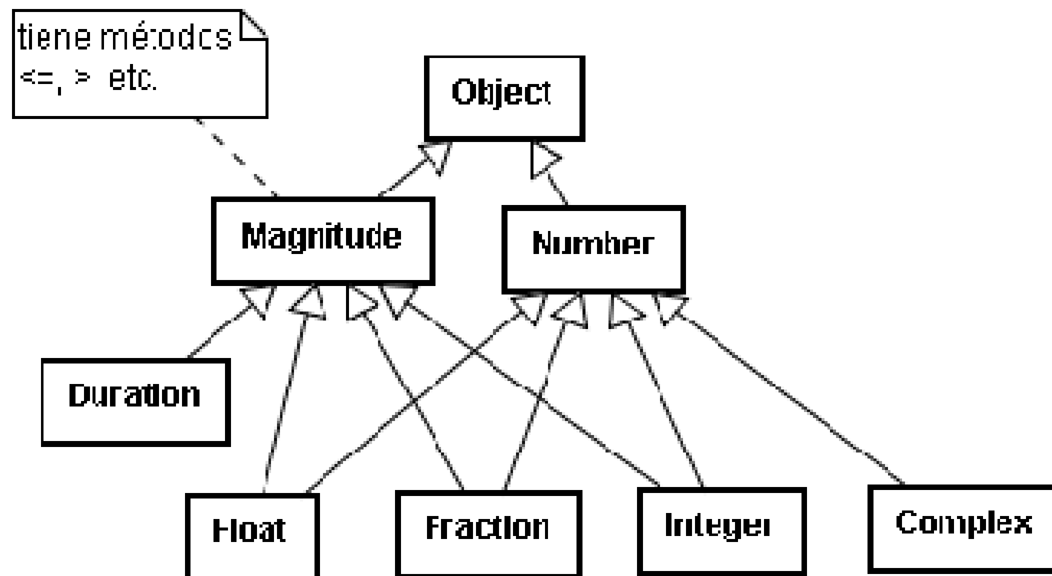
Las clases dejaron de ser excluyentes

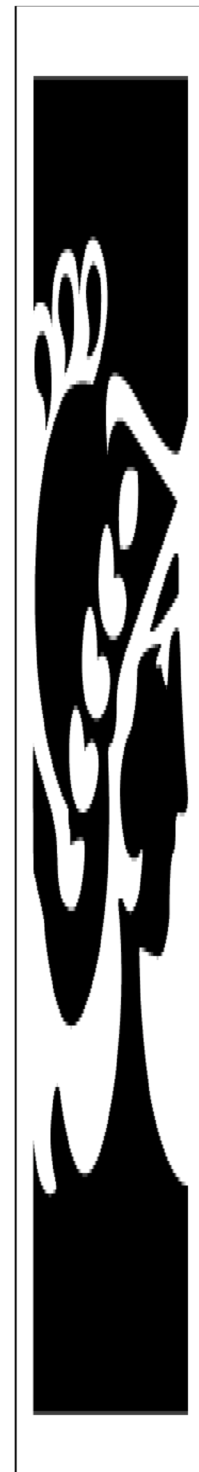


Ejemplo Smalltalk sin herencia múltiple



¿No sería mejor…

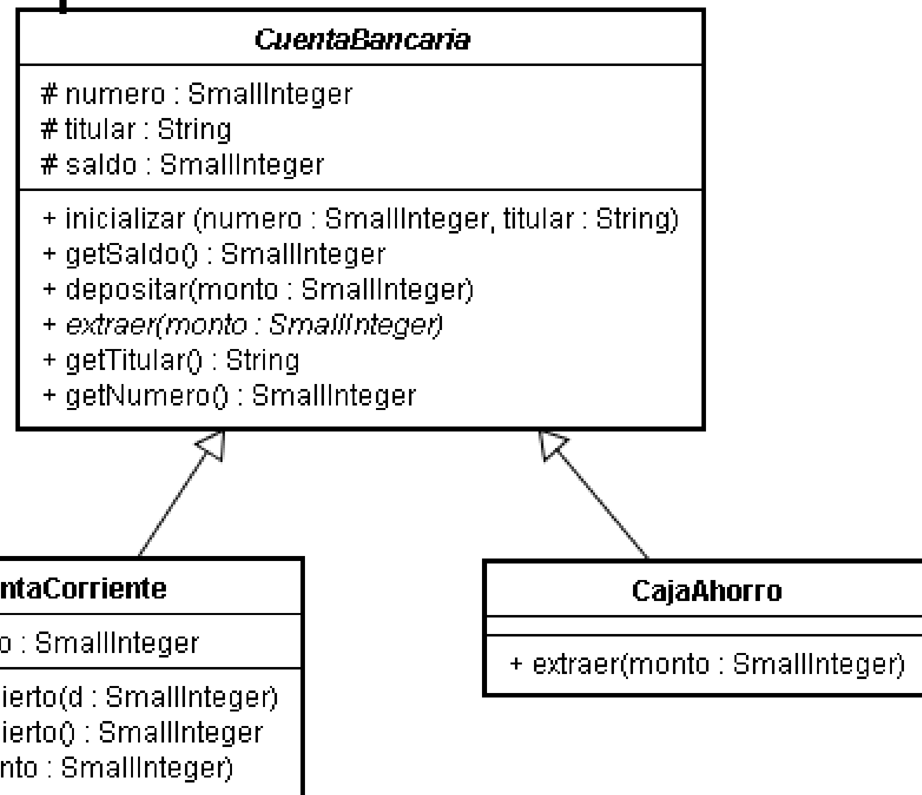


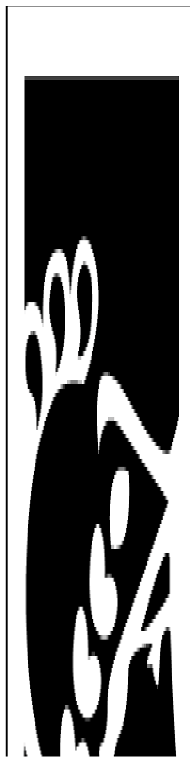


A3F

Clases abstractas (conceptual)

- No tienen instancias
  - Caso de CuentaBancaria si implemento CajaAhorro
- Generalizan estructura y comportamiento de varias clases
  - Caso del método depositar
  - O crean una familia



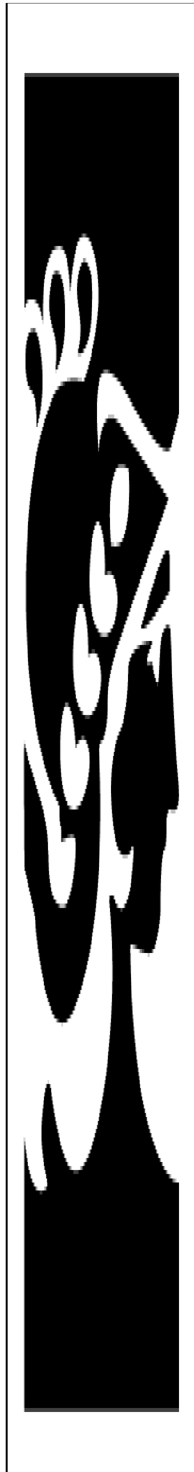


Clases abstractas en Smalltalk

No definidas en Smalltalk

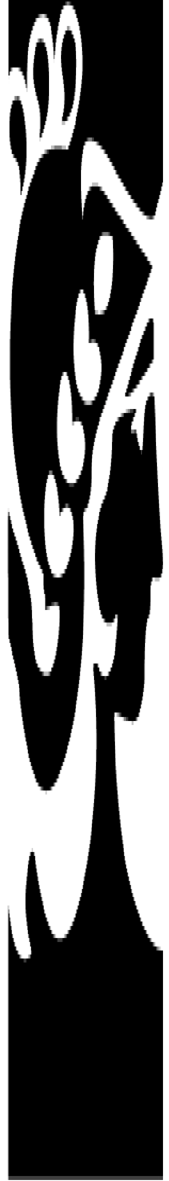
Sólo convencionalmente

Se supone que una clase con un método abstracto es abstracta

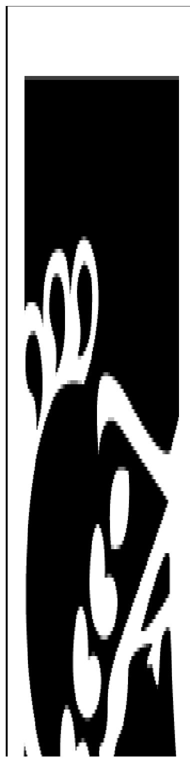


Métodos abstractos (conceptual)

- No se quiere que sean invocados: caso del extraer de CuentaBancaria
  - Las instancias de las subclases van a poder responder el mensaje
  - Pero sin definir comportamiento en la clase ancestro
  - A nivel de la clase madre sólo se puede prever la firma que tendrá
- Corolarios
  - No tienen implementación
  - Deben redefinirse



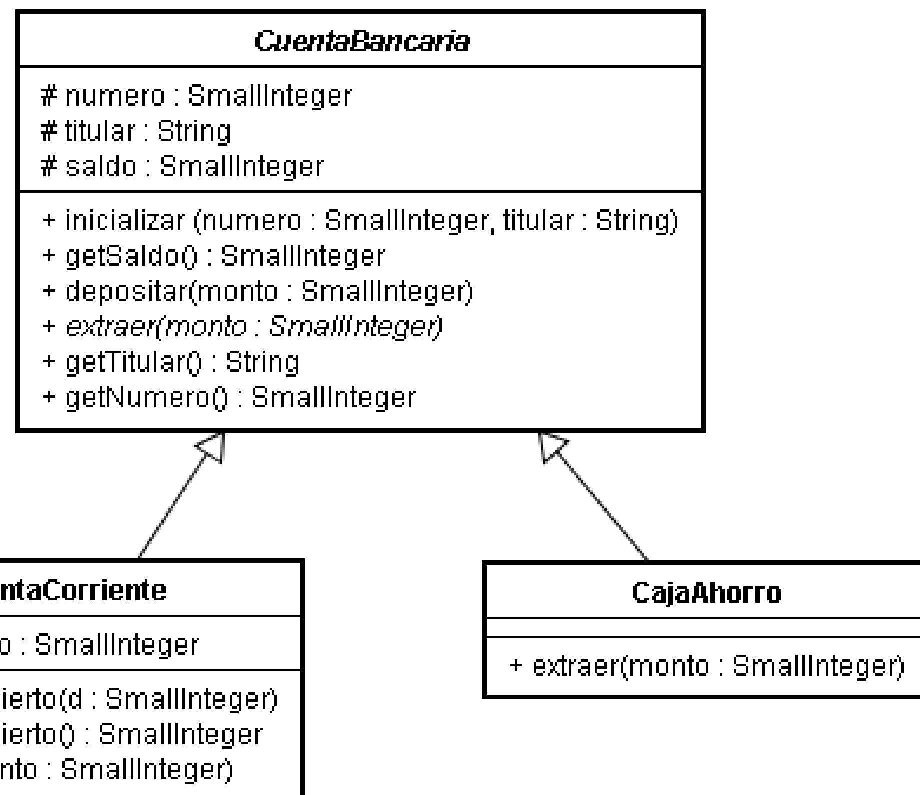
Métodos abstractos en Smalltalk



Convencional

Llamar a self subclassResponsibility

Pero no hay manera de forzar que no se lo llame



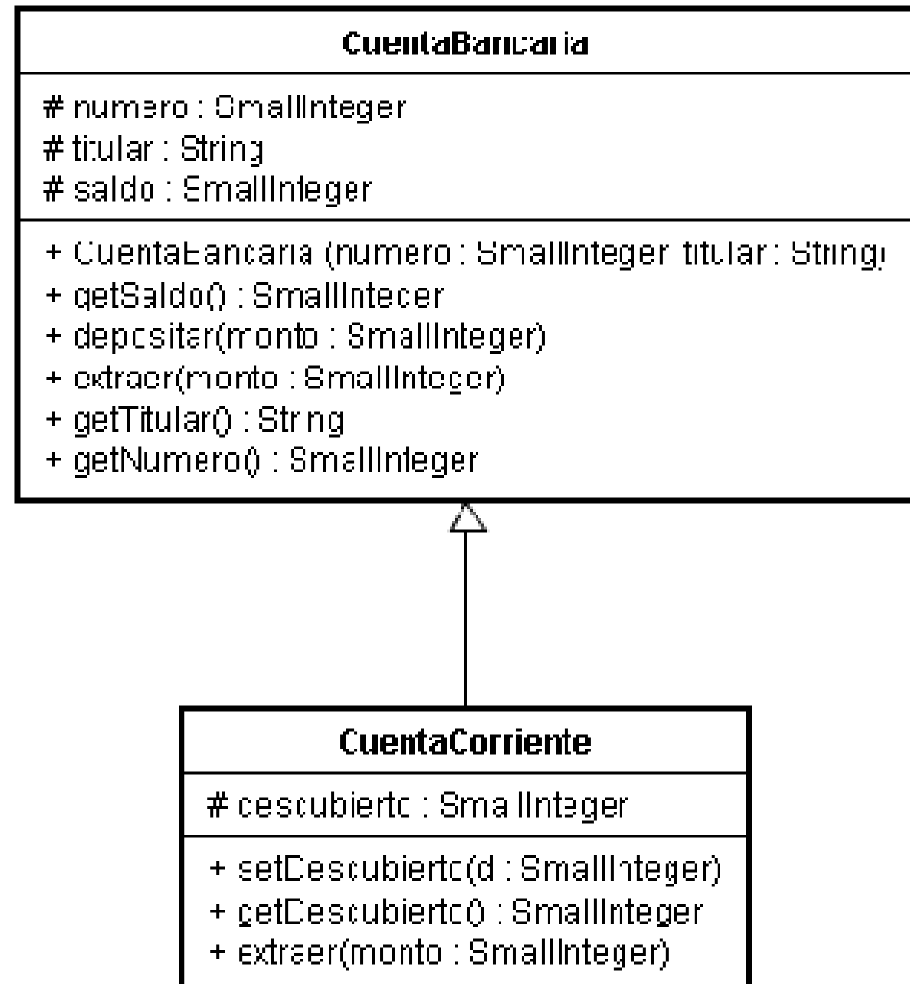
Analizar

cb := CuentaBancaria new.

cc := CuentaCorriente new.

cb extraer: 200.

cc extraer: 200.



A3F

Métodos virtuales (1)

ColeccionFiguras

dibujar()

\- figuras *

Figura

dibujar()
mover(x, y)
borrar()
cambioPos(x, y)

```
mover_enX: x enY: y
  self borrar.
  self cambioPos_enX: x enY: y.
  self dibujar.
```

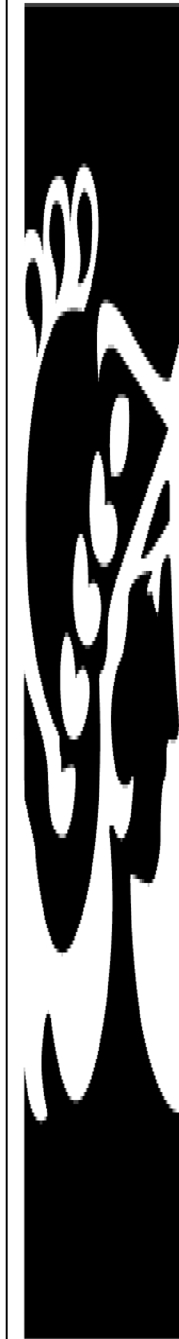
```
dibujar
  figuras do: [ :x | x dibujar ].
```

Elipse

dibujar()

Triangulo

dibujar()



Métodos virtuales (2)

unaElipse mover.

=> llama al dibujar de Elipse

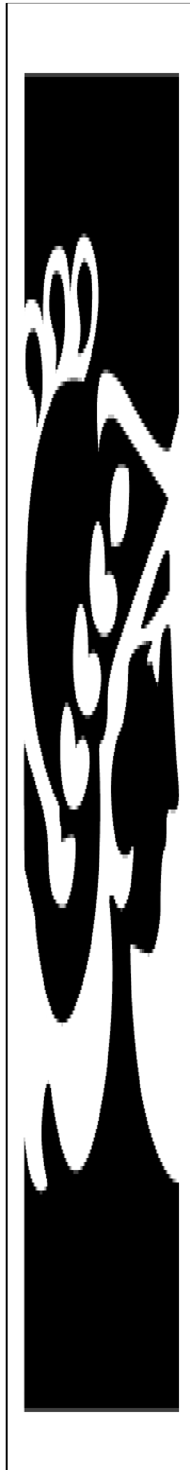
unTriangulo mover.

=> llama al dibujar de Triangulo

coleccion dibujar.

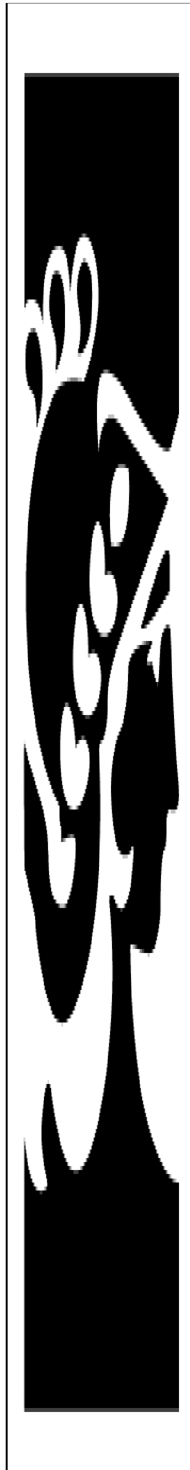
=> llama al dibujar de la clase de cada figura

En Smalltalk la “virtualidad” se da por defecto



Métodos virtuales (3)

- Los métodos virtuales agregan ineficiencias
  - Pero garantizan reutilización
  - Eliminar la “virtualidad” sólo si se demuestra que no se van a redefinir y la presunta ineficiencia
- Un método debe ser virtual sí o sí cuando se lo redefinirá y es llamado desde:
  - Un método en una clase ancestro
  - Un método que delegue en el método en cuestión de la clase ancestro
- En Smalltalk no hay opción: todo método de instancia es virtual



Ejemplo estándar en Smalltalk

Magnitude >> <= otroValor

  ^ self subclassResponsibility

Magnitude >> > otroValor

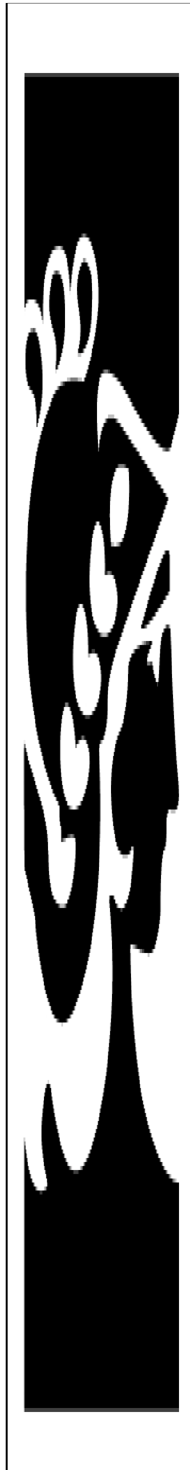
  ^ otroValor <= self

Luego, SortedCollection usa <= para insertar elementos

  => deberíamos redefinir <= en la clase correspondiente

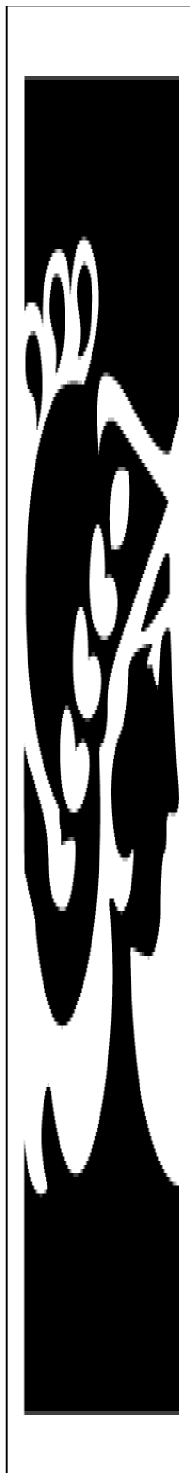
Otro ejemplo: Object >> =

Otro más: Object >> printString, invoca Object >> printOn

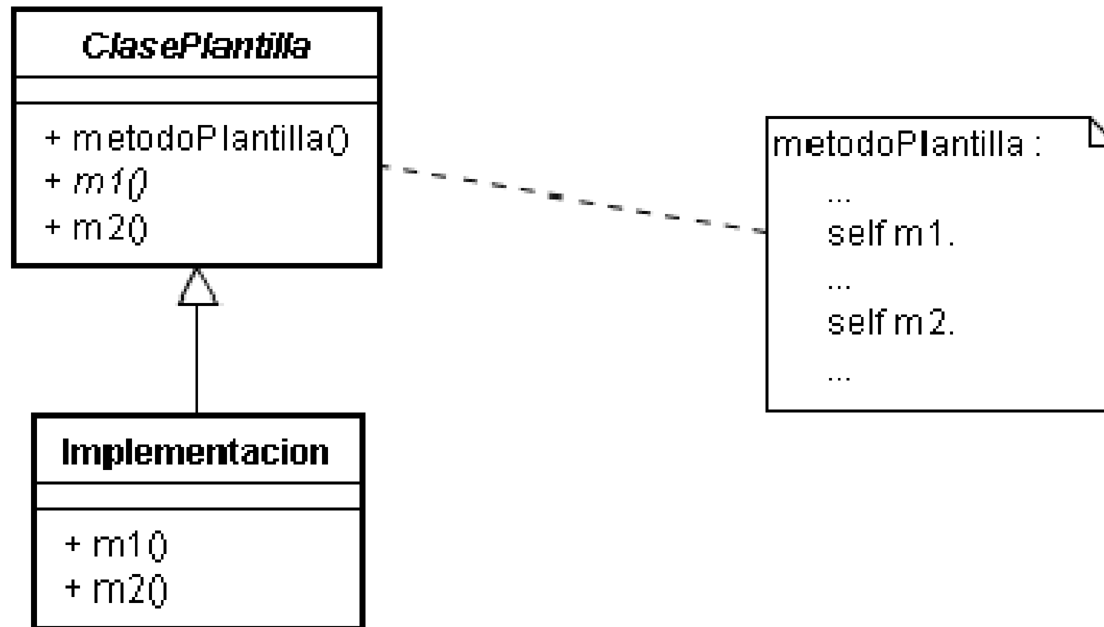


Polimorfismo

- Objetos de distintas clases de una misma familia entienden los mismos mensajes
  - Igual semántica (significado)
  - Implementaciones diferentes
  - Un mismo mensaje puede provocar la invocación de métodos distintos
- Vinculación tardía
  - Se retarda la decisión sobre el método a llamar hasta el momento en que vaya a ser utilizado



Aplicaciones del polimorfismo: Template Method



¿Recuerdan SUnit?

- TestCase es la "clase plantilla"
- setUp es un método a redefinir, con implementación vacía por defecto

Aplicaciones del polimorfismo: Strategy

montoFactura usa calculo para saber qué algoritmo aplicar

Factura

+ montoFactura() : SmallInteger
+ balanceada() : Boolean

1 — - calculo — 1

EstrategiaCalculo

+ calculoTotal(f : Factura)

EstrategiaSimple

+ calculoTotal(f : Factura)

EstrategiaJubilados

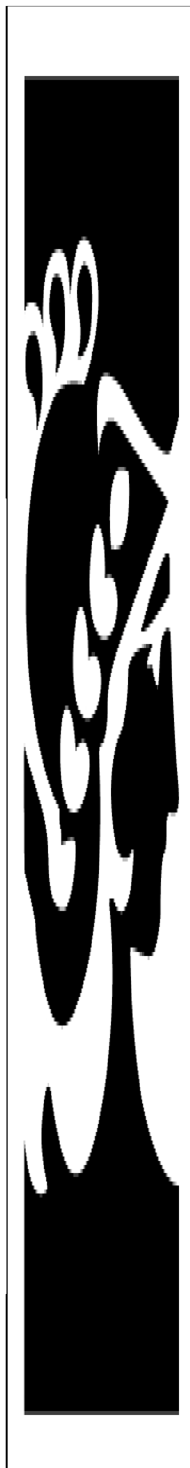
+ calculoTotal(f : Factura)

EstrategiaEmpleados

+ calculoTotal(f : Factura)

EstrategiaMartes

+ calculoTotal(f : Factura)



Claves

Herencia si se va a reutilizar la interfaz tal como está

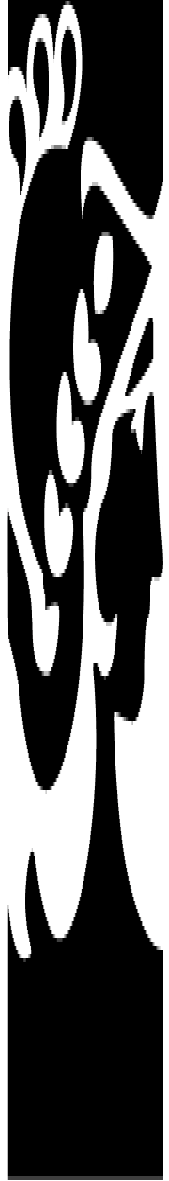
Relaciones “es un”

Delegación cuando se va a reutilizar cambiando la interfaz

Redefinición permite cambiar implementación manteniendo la semántica

Clases abstractas no tienen instancias

Polimorfismo = distintos comportamientos para un mismo mensaje



Lecturas obligatorias

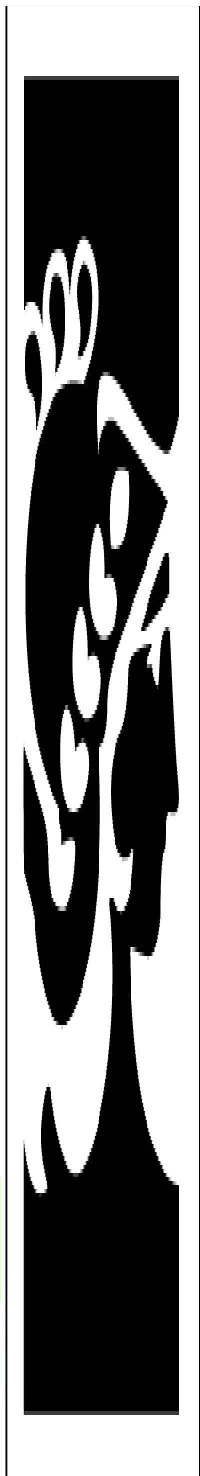
Principios de diseño de Smalltalk, de Daniel H. H. Ingalls.

Lo tienen en:

http://www.smalltalking.net/Papers/stDesign/stDesign.htm

Domain Driven Design, de Eric Evans, capítulo 1

Lo tienen en: http://domaindrivendesign.org/sites/default/files/books/chapter01.pdf



Lecturas optativas

Object-oriented analysis and design : with applications, Grady Booch

- Capítulo 3: “Classes and Objects”

Análisis y diseño orientado a objetos, James Martin y James Odell

- Capítulo 16: “Administración de la complejidad de un objeto”

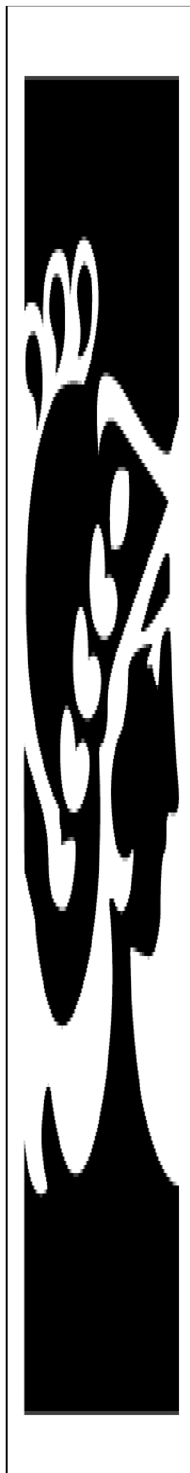
Ambos libros están en biblioteca

- El de Booch tiene una versión en castellano, agotada
- Son libros antiguos

Orientación a objetos, diseño y programación, Carlos Fontela 2008, capítulos 4 y 5 “Delegación” y “Herencia de implementación”

Orientación a objetos, diseño y programación, Carlos Fontela 2008, capítulos 7 y 8 “Polimorfismo basado en herencia” y “Polimorfismo basado en interfaces”

UML gota a gota, Martin Fowler, capítulos 4, 5, 6 y 7



Qué sigue

Excepciones y cierre conceptual

Temas de desarrollo de software

Calidad de código y buenas prácticas de desarrollo