FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

Reutilización con Delegación y Herencia

Carlos Fontela
cfontela@fi.uba.ar

A3F

Temario

Delegación

Herencia

UML: clases, paquetes, secuencias

Cuándo usar herencia y cuándo delegación

Redefinición

Clases abstractas

Mecanismos de abstracción (1)

Clasificación (individuo-especie)

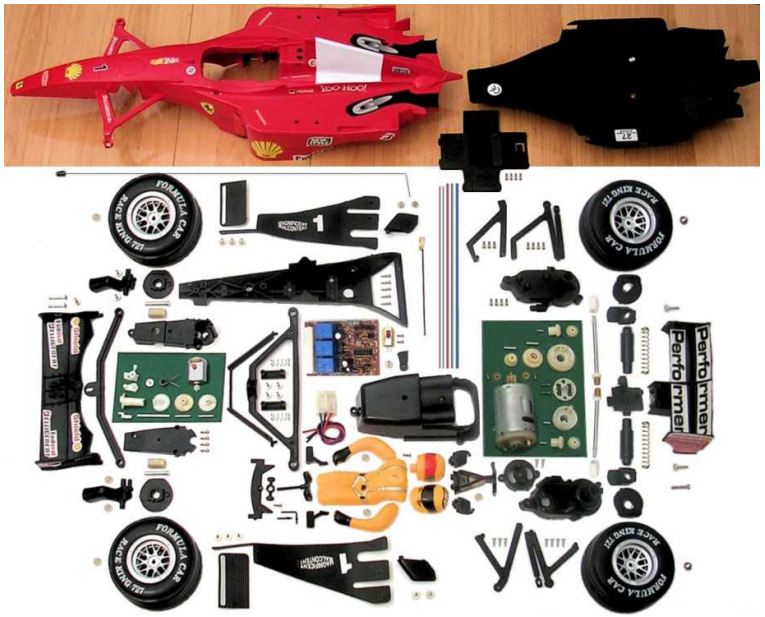
Lassie – perro / Juan Pérez – ser humano

En OO: instanciación (objeto-clase)

Mecanismos de abstracción (2)

Agrupación (entre individuos)

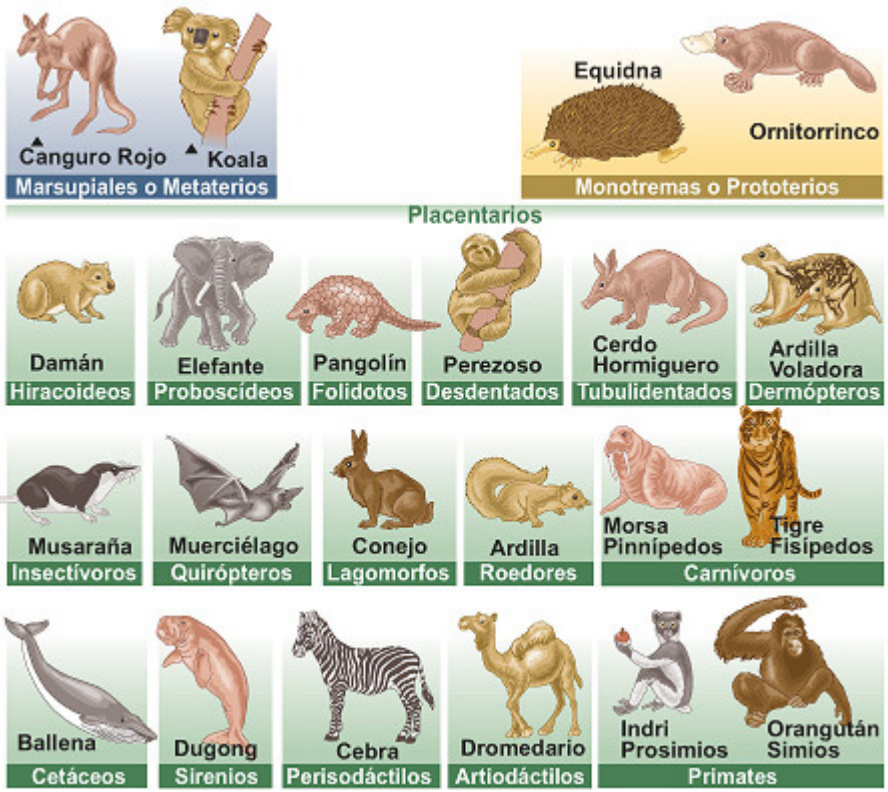
Auto - rueda



En OO: agregación (objeto-objeto)

Mecanismos de abstracción (3)

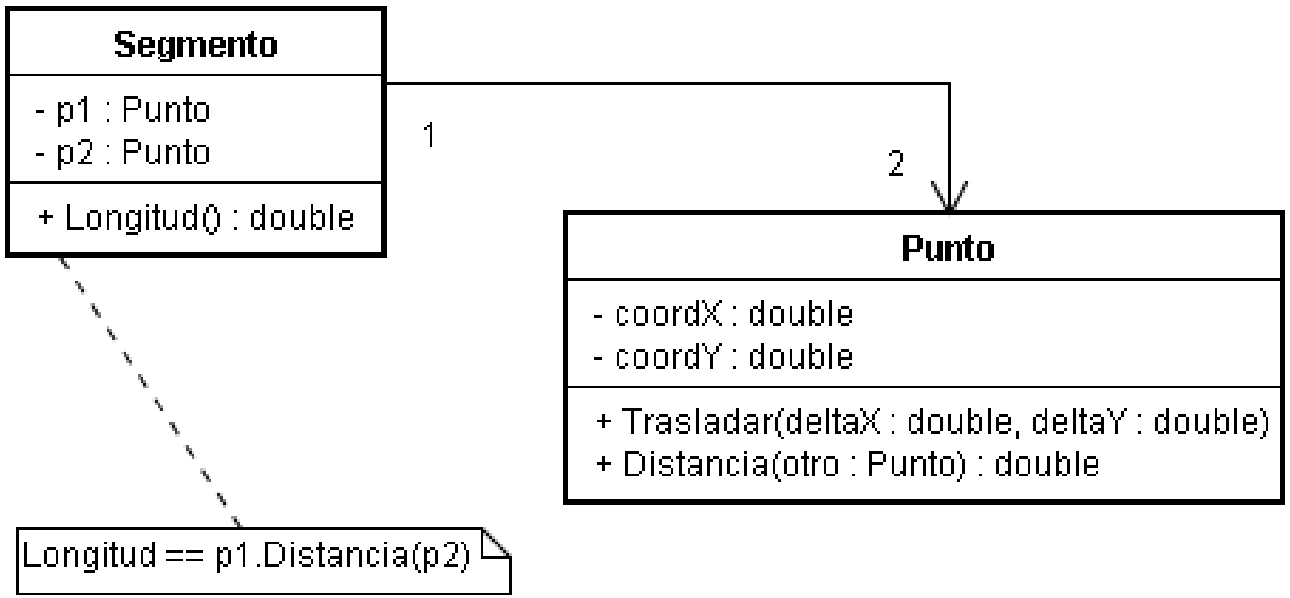
- Generalización (entre especies)
  - Lápiz – herramienta de escritura
  - Animal – ser vivo
- En OO: herencia (clase-clase)



Delegación (1)

Un objeto contiene referencias a otros objetos y les delega comportamiento

```
public class Segmento {
    private Punto p1;
    private Punto p2;
    ...
}
```



Delegación (2)

Es una forma de reutilización

Mediante el envío de un mensaje a otro objeto

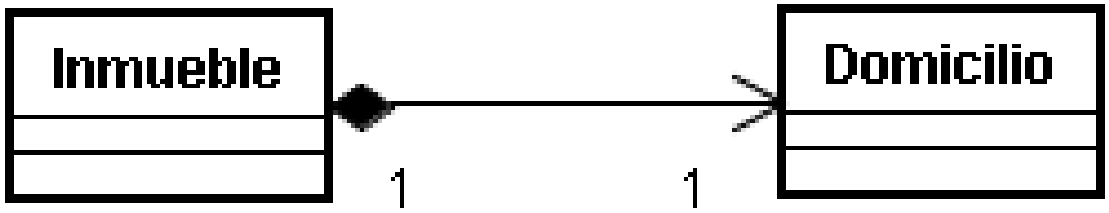
El otro objeto se preocupa de cómo implementa el método

Evitar los objetos omnipotentes

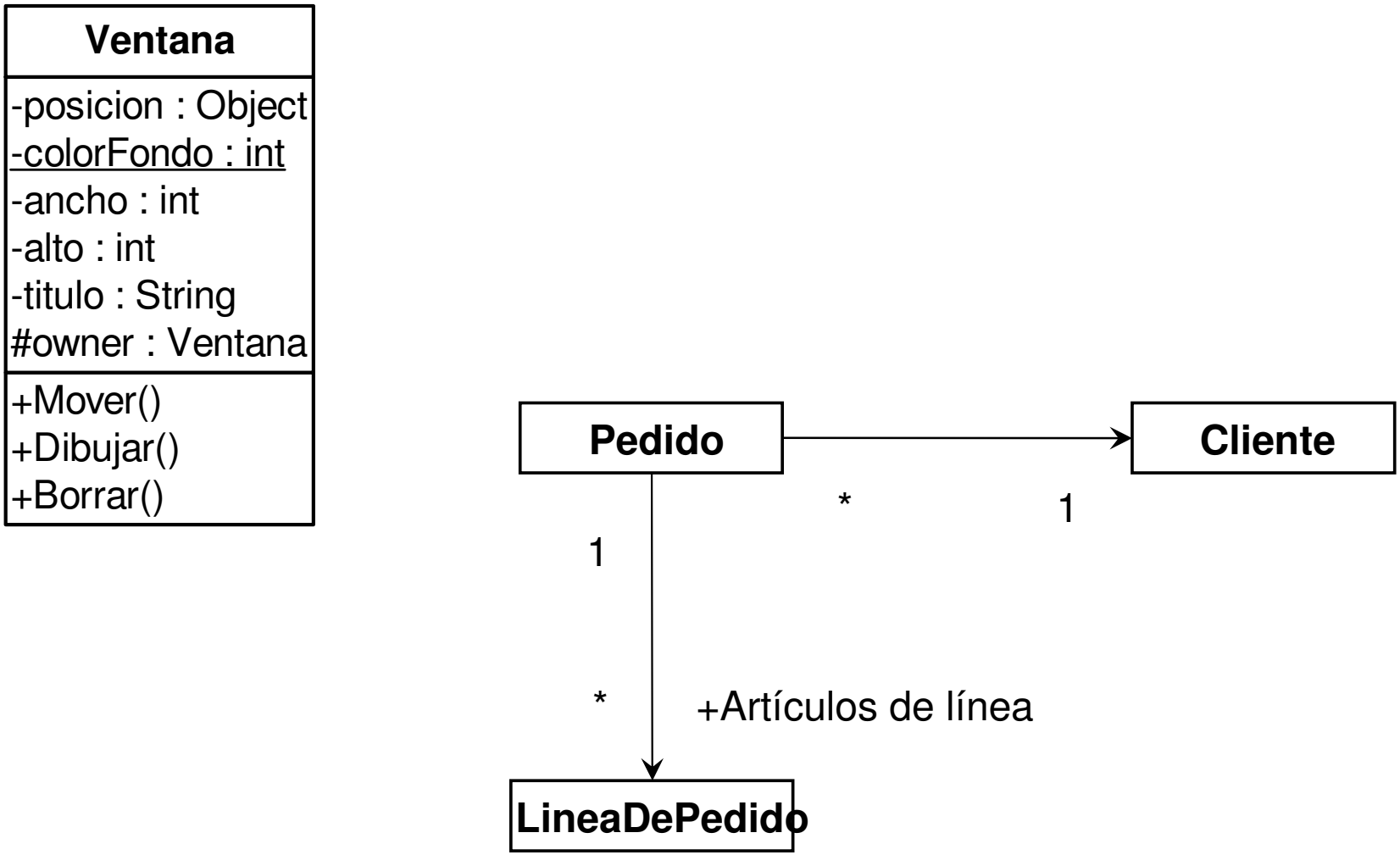
Agregación vs. Composición

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



Paquetes en UML

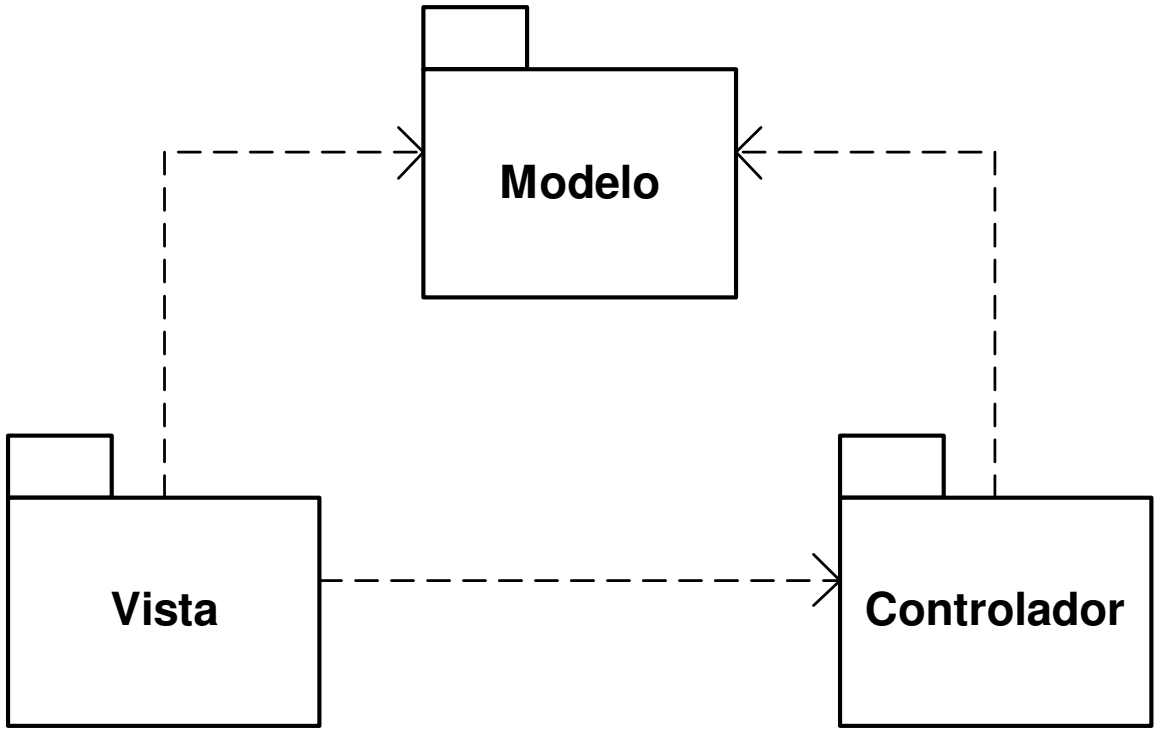
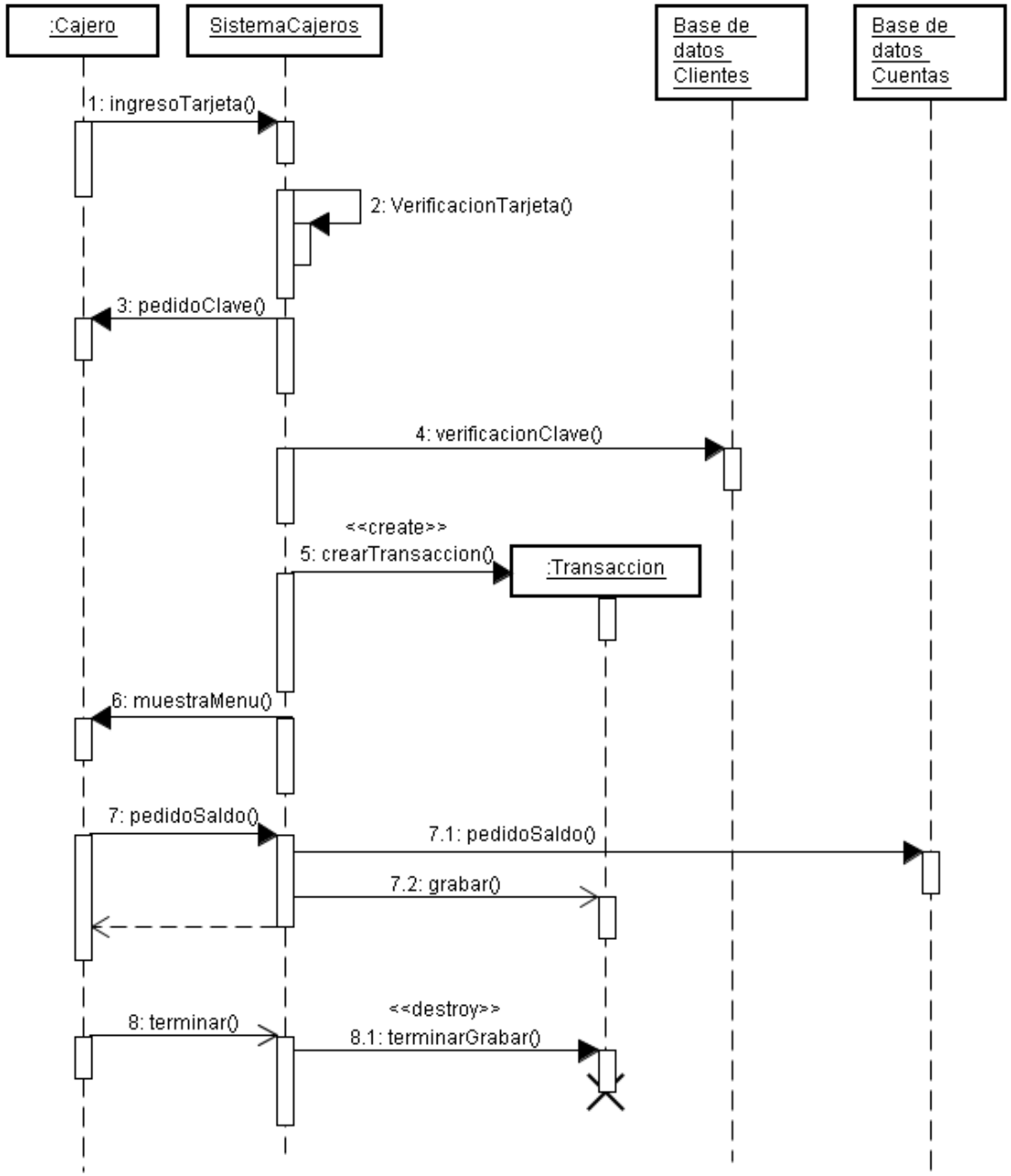


Diagrama de secuencias UML



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.

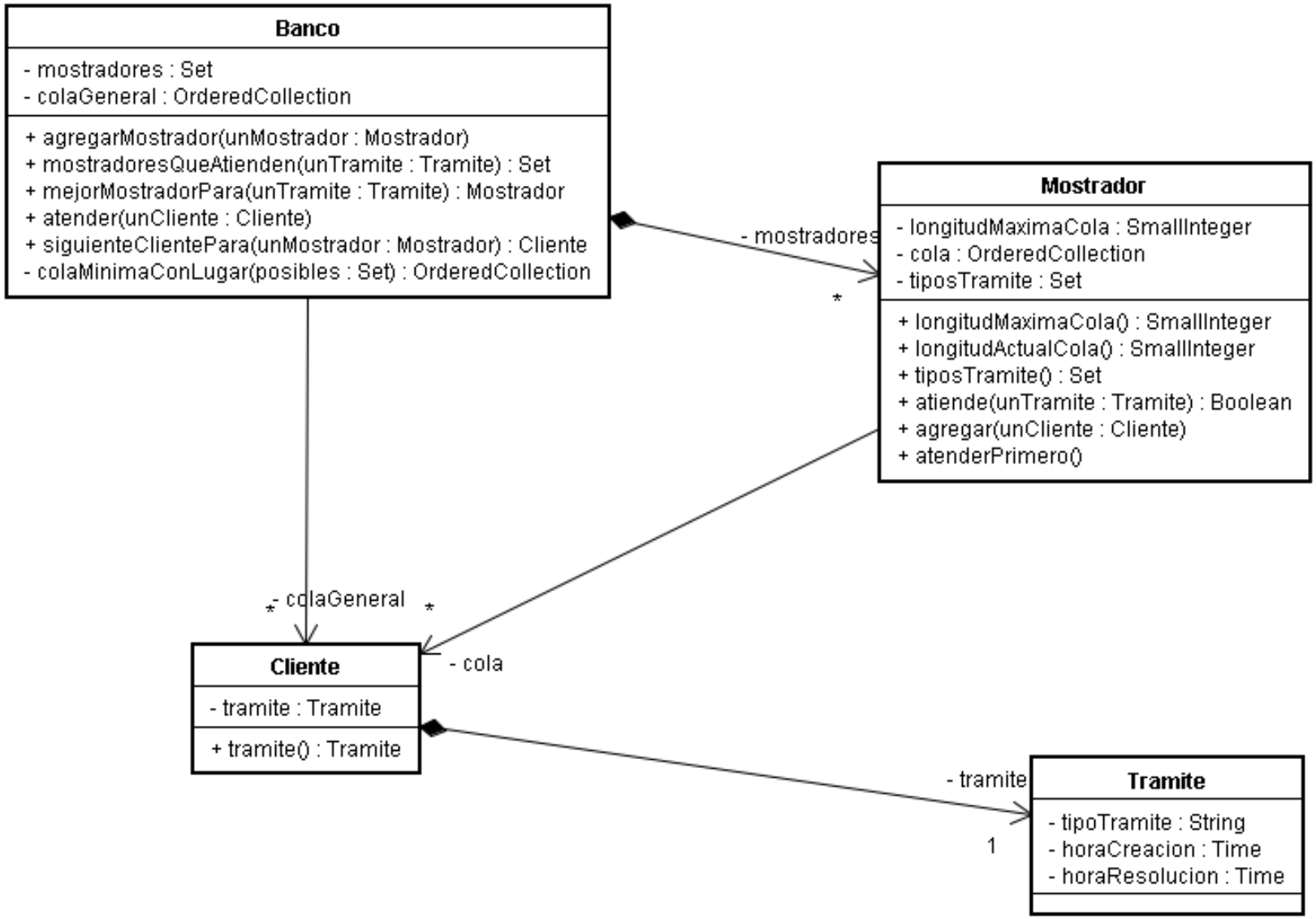
3) Implementar el método banco.mejorMostradorPara(unTramite), que retorna el mostrador con la cola más corta con espacio para al menos una persona más y que atienda ese trámite; si ningun mostrador tiene espacio, retorna null.

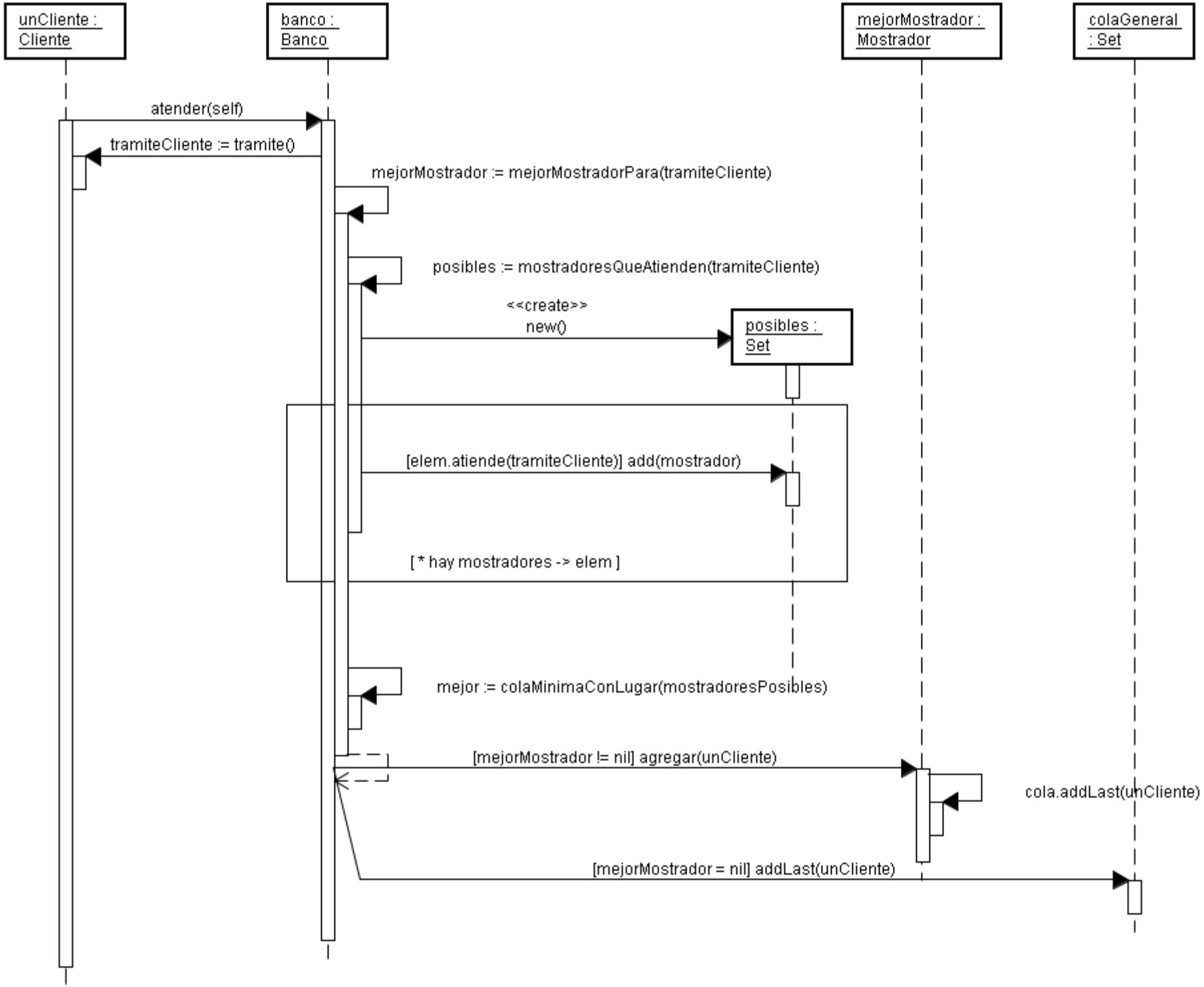
4) Implementar el método banco.atender(unCliente); cuando llega un cliente al banco se lo ubica en el mostrador que atienda el trámite que el cliente requiere, que tenga espacio y la menor cantidad de clientes esperando; si no hay lugar en ningún mostrador el cliente debe permanecer en la cola general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

Solución

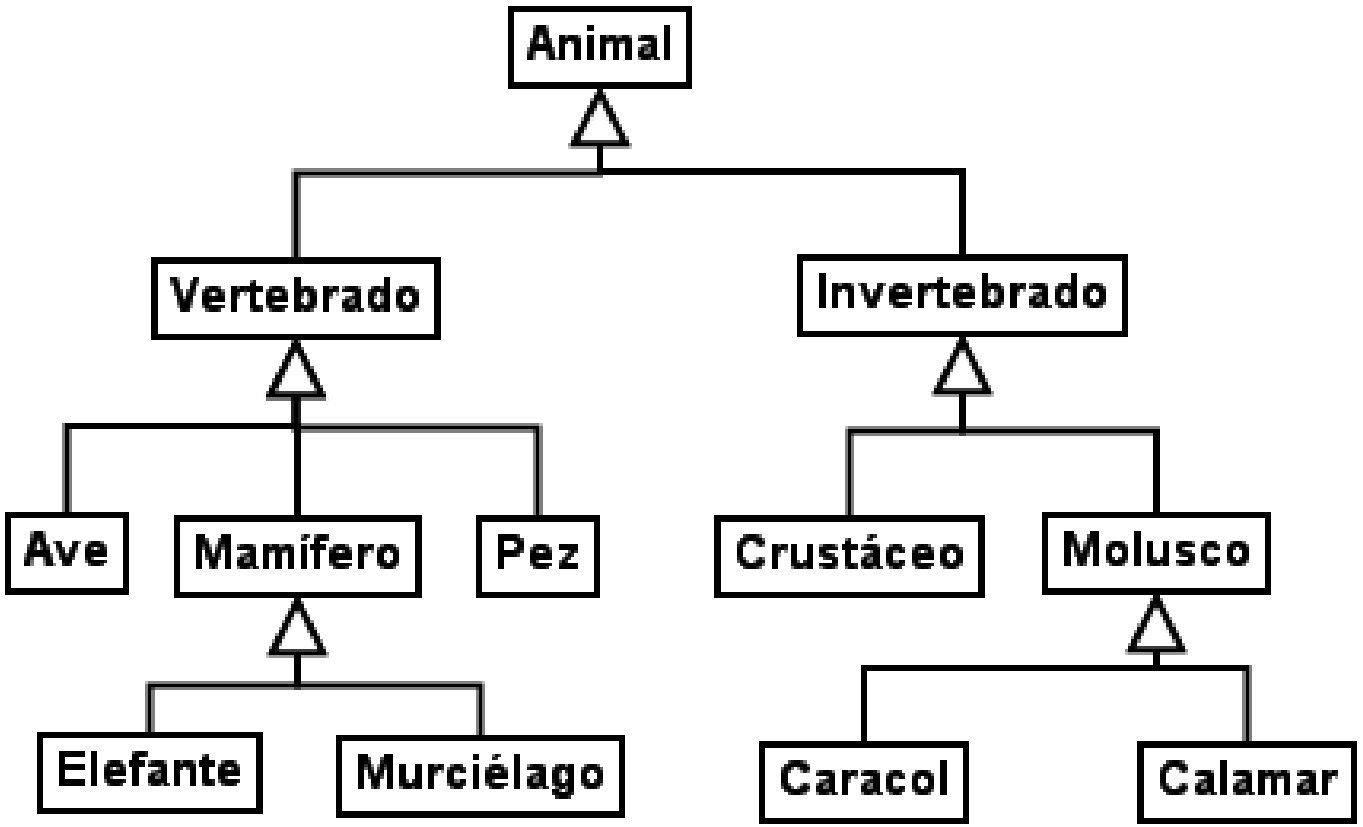

Banco
- mostradores : Set
- colaGeneral : OrderedCollection
+ agregarMostrador(unMostrador : Mostrador)
+ mostradoresQueAtienden(unTramite : Tramite) : Set
+ mejorMostradorPara(unTramite : Tramite) : Mostrador
+ atender(unCliente : Cliente)
+ siguienteClientePara(unMostrador : Mostrador) : Cliente
- colaMinimaConLugar(posibles : Set) : OrderedCollection
- mostradores
*
Mostrador
- longitudMaximaCola : SmallInteger
- cola : OrderedCollection
- tiposTramite : Set
+ longitudMaximaCola() : SmallInteger
+ longitudActualCola() : SmallInteger
+ tiposTramite() : Set
+ atiende(unTramite : Tramite) : Boolean
+ agregar(unCliente : Cliente)
+ atenderPrimero()
* - colaGeneral *
- cola
Cliente
- tramite : Tramite
+ tramite() : Tramite
- tramite
1
Tramite
- tipoTramite : String
- horaCreacion : Time
- horaResolucion : Time


unCliente : Cliente
banco : Banco
mejorMostrador : Mostrador
colaGeneral : Set
atender(self)
tramiteCliente := tramite()
mejorMostrador := mejorMostradorPara(tramiteCliente)
posibles := mostradoresQueAtienden(tramiteCliente)
<<create>>
new()
posibles : Set
[elem.atiende(tramiteCliente)] add(mostrador)
[* hay mostradores -> elem]
mejor := colaMinimaConLugar(mostradoresPosibles)
[mejorMostrador != nil] agregar(unCliente)
cola.addLast(unCliente)
[mejorMostrador = nil] addLast(unCliente)

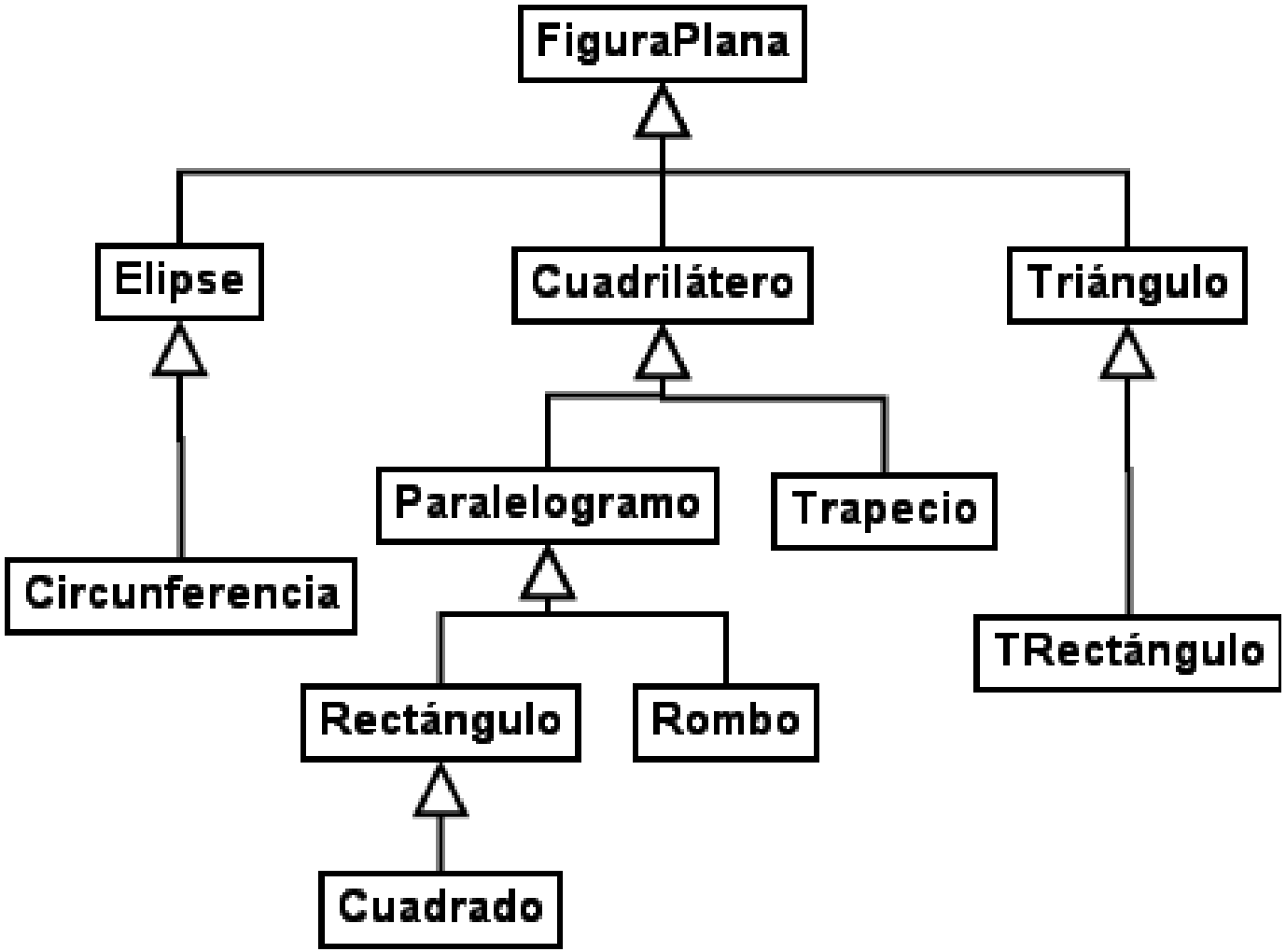
A3F

Taxonomías (I)

Relaciones “es un”



Taxonomías (II)



Taxonomías (III)

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

Reutilización

¿Usar clases en una aplicación diferente a que motivó su primer uso, tal como están?

- Es el uso simple de clases
- No se lo llama reutilización

Extensión: Usar clases como base para la creación de otras nuevas con comportamiento parecido

- Delegación
- Herencia
- No necesitamos el código fuente

Herencia y lenguajes

En Java, para indicar herencia

```
public class Elipse extends Figura { ... }
```

En C# reemplazamos “extends” por “:”

Elipse tiene, por lo menos:

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

Herencia, variables y objetos

"Casteo" hacia arriba automático

```
Elipse e = new Elipse();
Figura f = e;          // válido y seguro
// e = f;          inválido e inseguro
```

Otro caso

```
public void p (Figura x) { … }
// … luego será invocado:
p(e);// e es de tipo Elipse
```

Los llamamos "objetos polimorfos"

Parecen una cosa pero son otra

f es una Elipse aunque se la trate como Figura

```
Figura f = new Elipse ( );        // tipos distintos en la variable y el objeto !!
```

Herencia y colecciones

O colecciones polimorfas

```
public Figura[ ] v = new Figura [3];

// luego…

v[0] = new Elipse ( );

v[1] = new Circulo ( );

v[2] = new TrianguloRectangulo ( );
```

Pero

- Estamos perdiendo información del objeto
- Cada v[i] sólo va a tener disponibles los atributos y métodos públicos de Figura
- Ya volveremos sobre esto

Particularidades de los lenguajes

- Jerarquía de raíz única (Java, C#, Smalltalk)
  - Clase Object
  - Todo se puede transformar en un Object
  - Atributos y métodos comunes a todas las clases
  - Otras importantes consecuencias
- Posibilidad de evitar la herencia (Java, C#)
  - Declaramos la clase como “final”
    - Ejemplo: String
      - public final class String {...}
  - “sealed” en C#

Más sobre visibilidad

- Atributos y métodos protegidos (protected)
  - Son visibles sólo para las clases descendientes y las del propio paquete
  - Poco uso; riesgos
- Clases con visibilidad “de paquete”
  - Sólo se pueden usar dentro de su paquete
- ¿Clases privadas?
  - Sólo cuando son clases internas
- No hay clases protegidas

Constructores

Los constructores no se heredan

- Cada clase debe tener el suyo

Receta

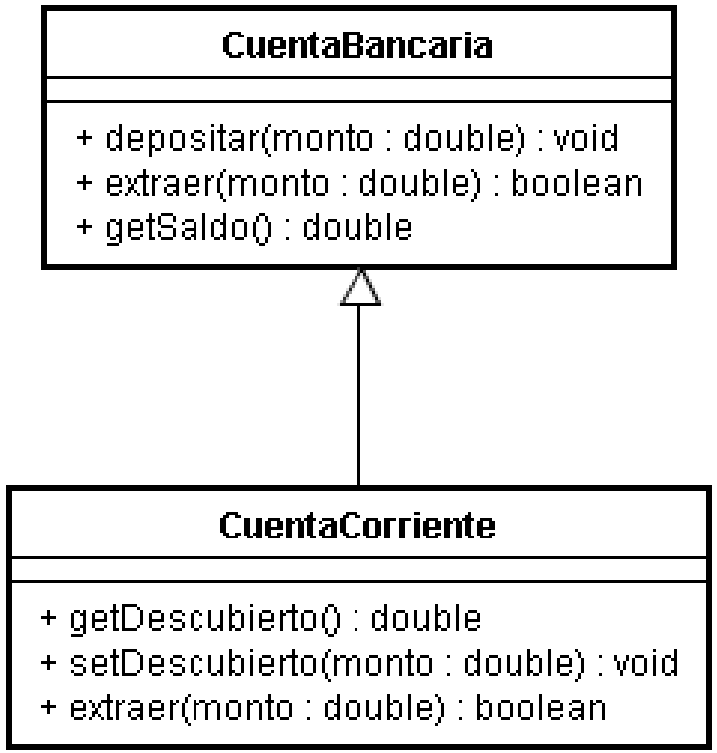
- Llamar al constructor del ancestro al principio del constructor propio

```
// Java:
public Derivada( ) {
   super();     ... }
// C#:
public Derivada( ) : base ( ) {
   ... }
```

- Automático con constructores por defecto

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 signatura

Redefinición en Java y C#

Los métodos privados no pueden ser redefinidos

- Se estaría definiendo uno nuevo

Posibilidad de evitar la redefinición

- Métodos “final”

```
public final void m() {...}
```

- “sealed” en C#

No se puede redefinir un método haciéndolo más privado

Sobrecarga y redefinición

- Redefinición se hace con la misma firma
- Si no, es sobrecarga

Herencia y diseño por contrato

- 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
- Invariantes de clase
  - Deben ser al menos los mismos de la clase ancestro
- Excepciones
  - Un método debe lanzar los mismos tipos de excepciones que en la clase ancestro, o a lo sumo excepciones derivadas de aquéllas

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)

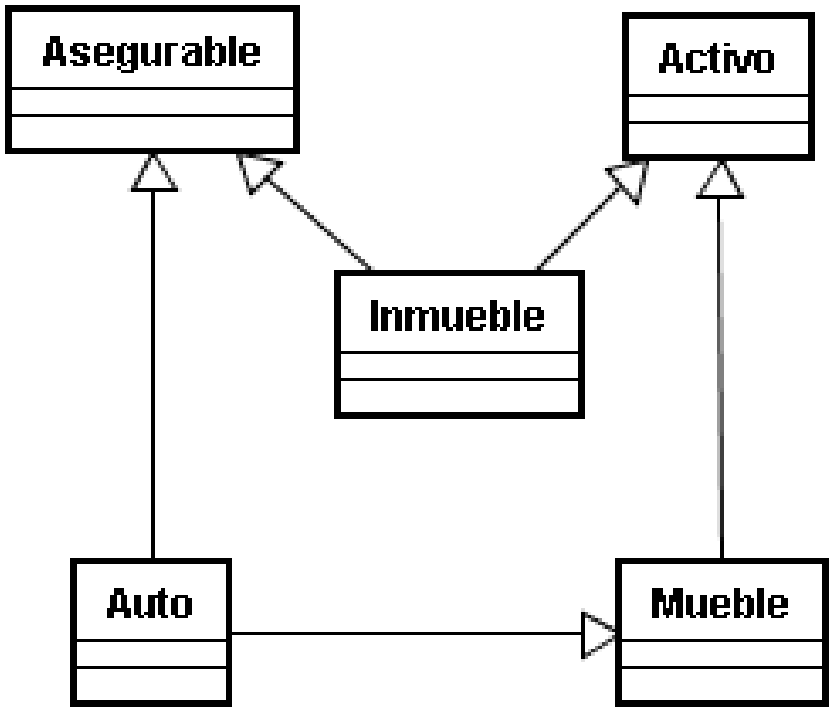
Herencia

- Cuando se va a reutilizar la interfaz

Delegación

- Cuando se va a reutilizar el estado

Herencia múltiple (C++, otros)



Las clases dejaron de ser excluyentes

Clases abstractas

No tienen instancias

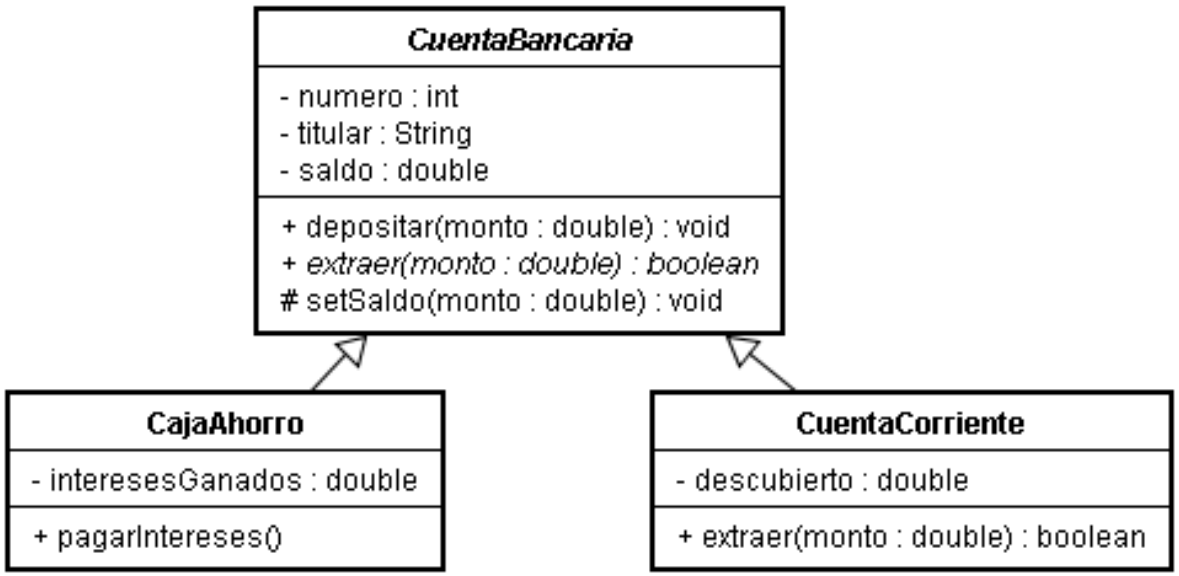
- Caso de CuentaBancaria si implemento CajaAhorro
- Pero pueden tener constructor
  - ¡que no debe ser llamado nunca!

Generalizan estructura y comportamiento de varias clases

- Caso del método mover
- O crean una familia

En Java y C#:

- Se declara “abstract”



Ejercicio: círculos y elipses

Suponga que tiene una clase Elipse, que implementa la siguiente interfaz para una elipse "no girada":

```
public interface ElipseI {
    Punto getCentro ( );
    double getRadioX ( );
    double getRadioY ( );
    void cambiarEscala (double factorX, double factorY);
    void moverCentro (Punto nuevoCentro);
    }
```

Implemente una clase Circulo:

a. Usted sabe que todo círculo es una elipse, por lo que desearía utilizar herencia para implementar Circulo. ¿Ve algún problema? Suponga que puede modificar la interfaz ElipseI, y la clase Elipse, ¿le permite esto solucionar el problema? Muéstrelo en código y en diagrama de clases.

b. Suponga que no tiene manera de cambiar la clase Elipse. ¿Se le ocurre otra idea, aunque sea sin usar herencia? Muéstrelo en código y en diagrama de clases.

¿Solución?

Ver
http://cysingsoft.wordpress.com/2009/04/27/sobre-el-circulo-y-la-elipse-carlos-fontela/

Claves

Herencia para relaciones “es un”

Composición para relaciones “es parte de”

Redefinición permite cambiar implementación manteniendo la semántica

Clases abstractas no tienen instancias

Lectura obligatoria

Apunte de Pruebas (ver sitio de la materia)

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

- Más de 15 años
- No existía Java ni C#, sí Smalltalk

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

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

Qué sigue

Polimorfismo

Smalltalk

Excepciones

Colecciones e iteradores