

fiuba

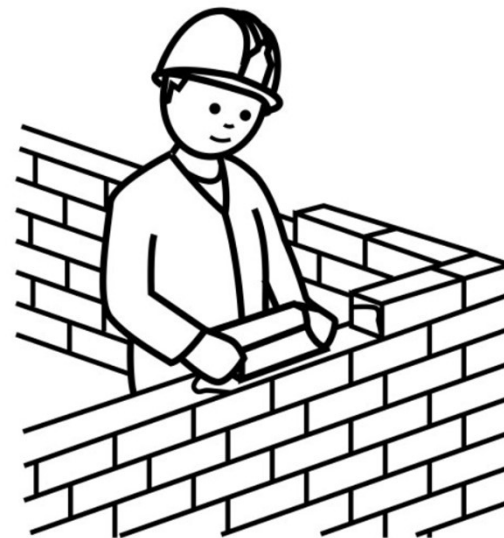
algo3

Colaboraciones de objetos (más delegación, herencia, etc.)

Carlos Fontela
cfontela@fi.uba.ar

Contexto

Vimos que un programa OO es un conjunto de objetos interactuando mediante mensajes
Pero hasta ahora sólo implementamos objetos aislados
=> Debemos ver cómo vincular objetos y hacerlos actuar en conjunto



Colaboraciones de objetos



Temario

Objetos en red y dependencias

Colaboración por delegación

Programar por diferencia

Herencia

Otras maneras

Redefinición

Clases y métodos abstractos

Visibilidad

Definiciones

Objeto receptor

Aquél que recibe un mensaje en un escenario de interacción entre objetos

Brinda un servicio => “servidor”

Objeto cliente

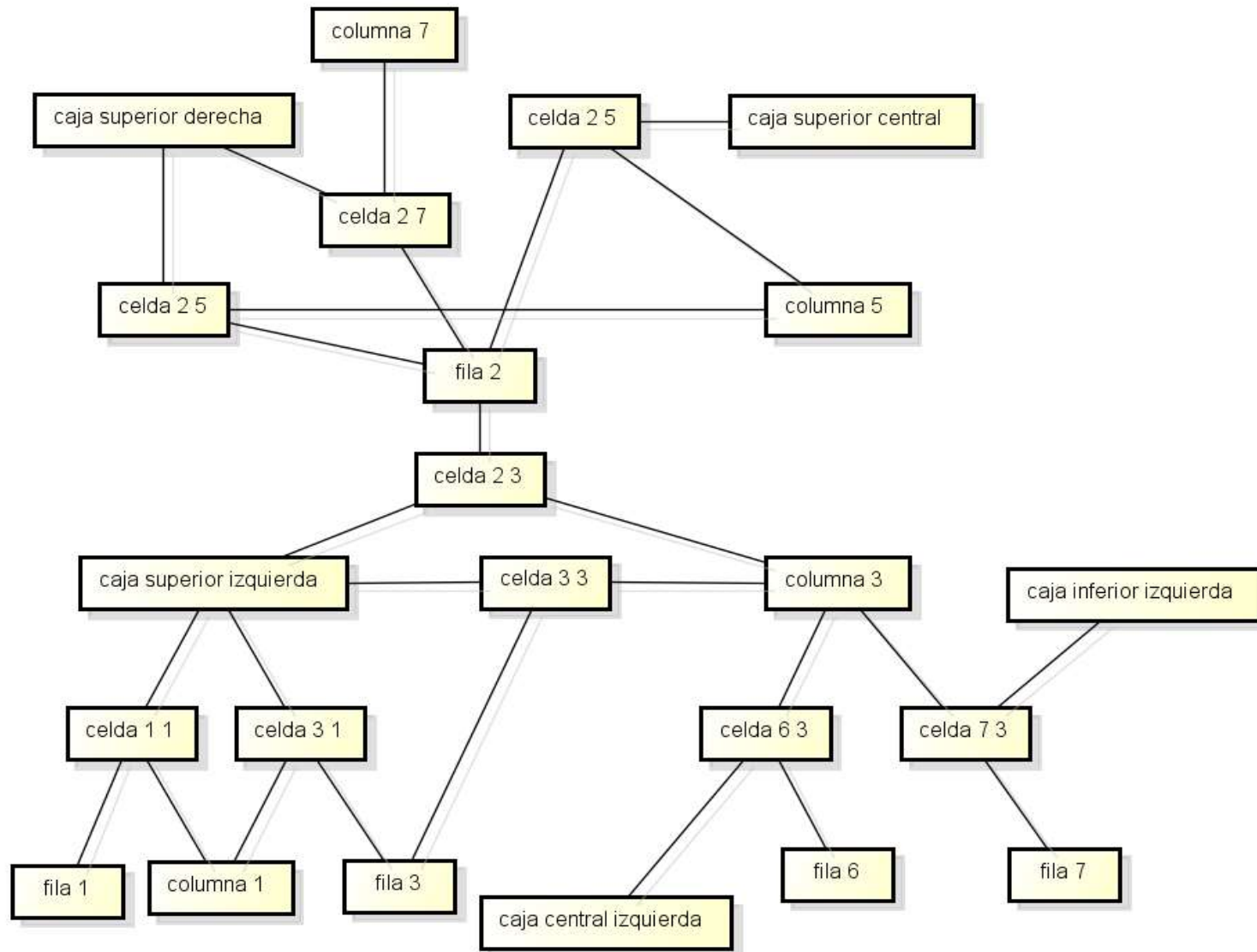
Aquél que envía un mensaje en un escenario de interacción entre objetos

Consume un servicio

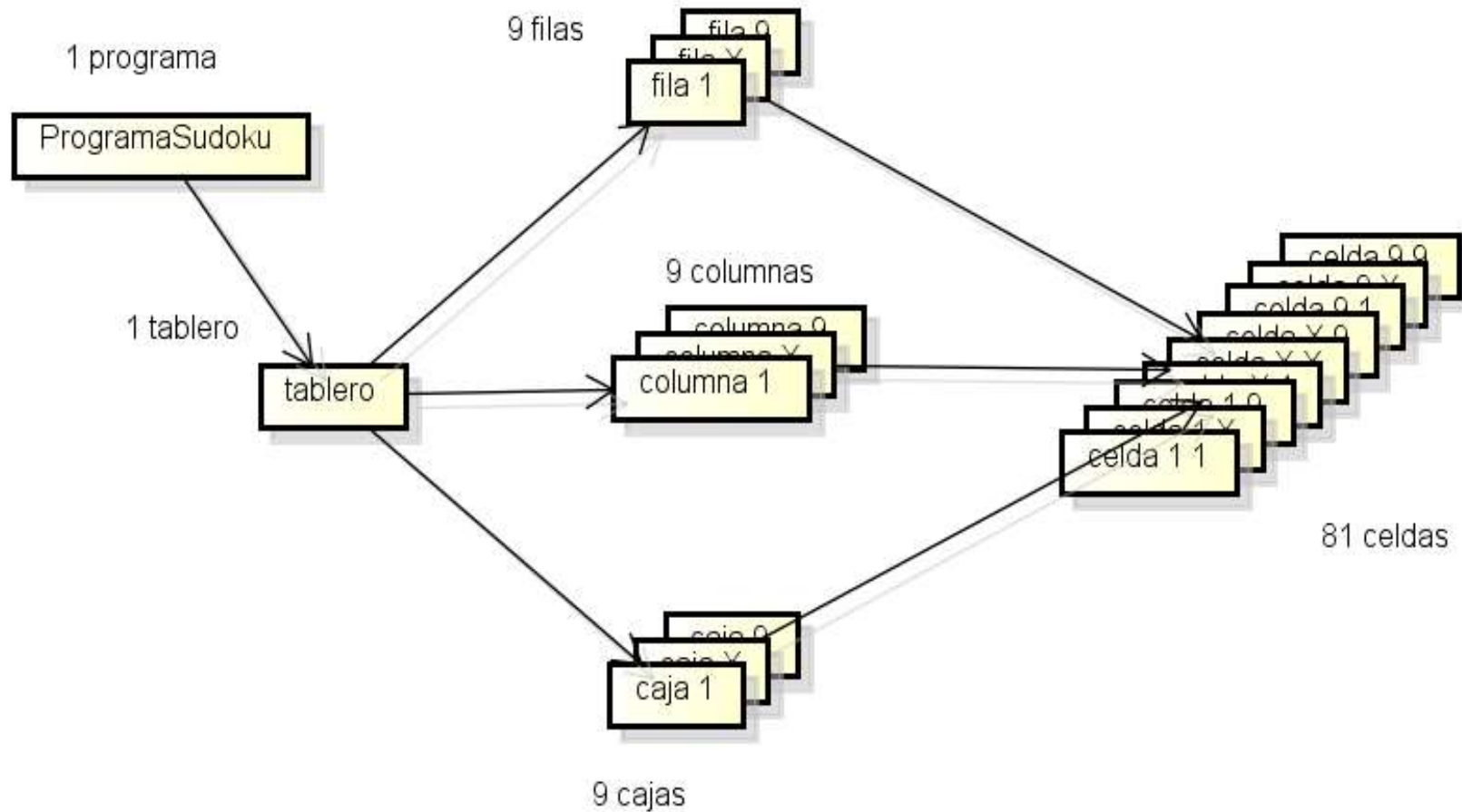
Atención

Todo mensaje tiene un objeto cliente y uno receptor
Su condición cambia según el mensaje

Web de objetos



Más fácil de ver...



Resolver fila >> contiene (numero)

contiene: numero

| encontrado |

((numero < 1) | (numero > 9))

ifTrue: [ValorInvalido new signal].

encontrado := false.

celdas do: [:celda | (celda contiene: numero)

ifTrue: [encontrado := true]].

^encontrado.

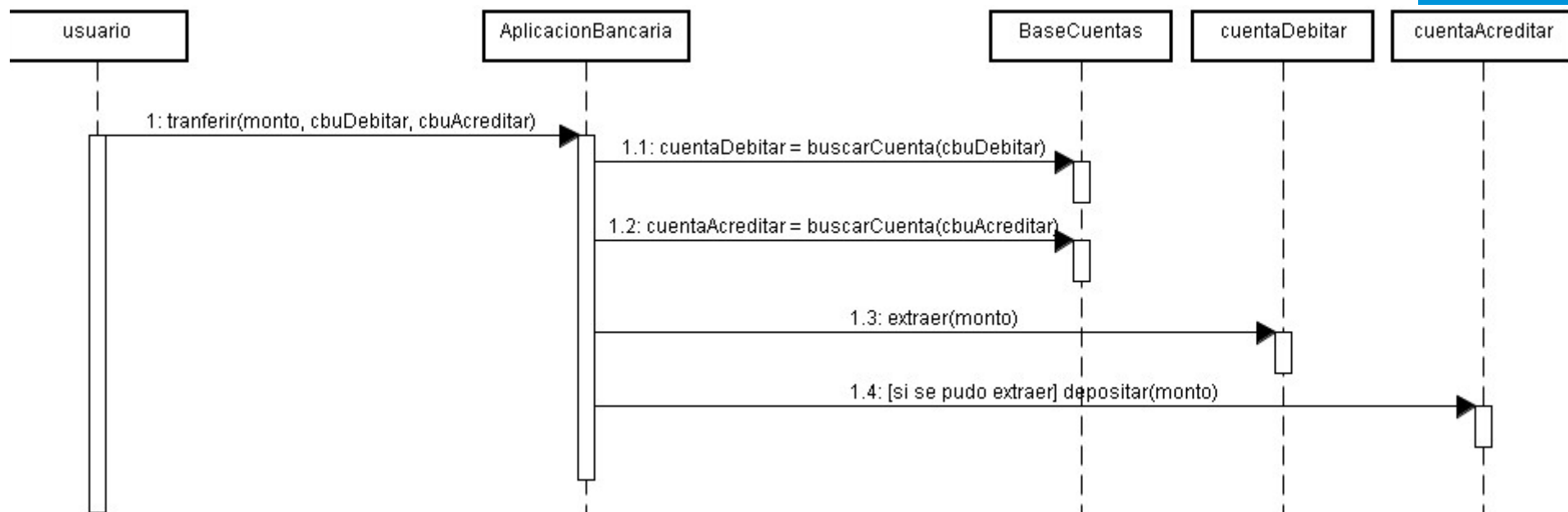
Colaboración por delegación

Se necesita alguna dependencia

Asociación

fila >> contiene delega en celda >> contiene

Hay otros tipos de dependencias

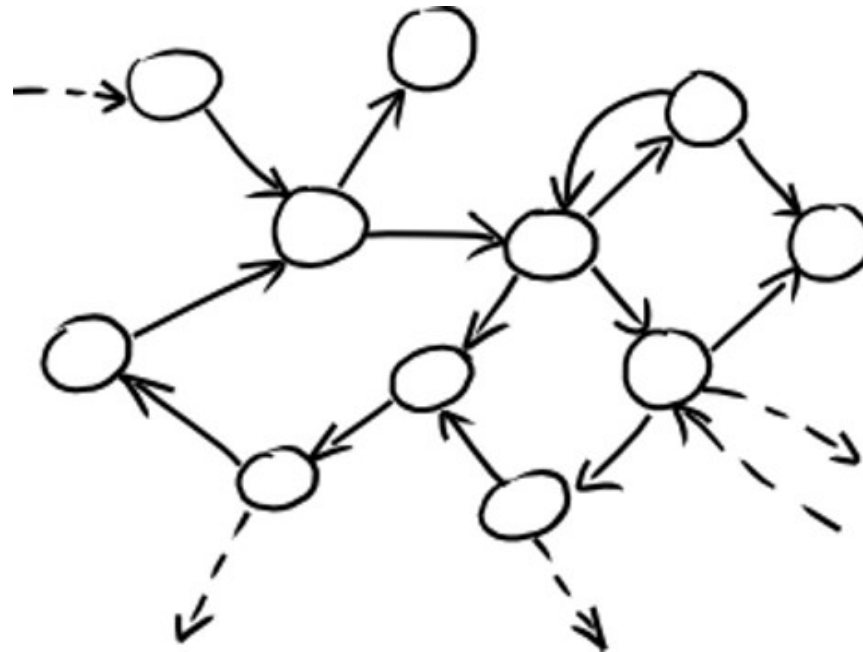


Dependencia y delegación

Un objeto depende de otro cuando debe conocerlo para poder enviarle un mensaje

Todo objeto cliente depende de su servidor

Decimos que el cliente delega en el receptor



Tipos de dependencia

Cliente debe conocer al receptor/servidor

- El objeto receptor se envía como argumento

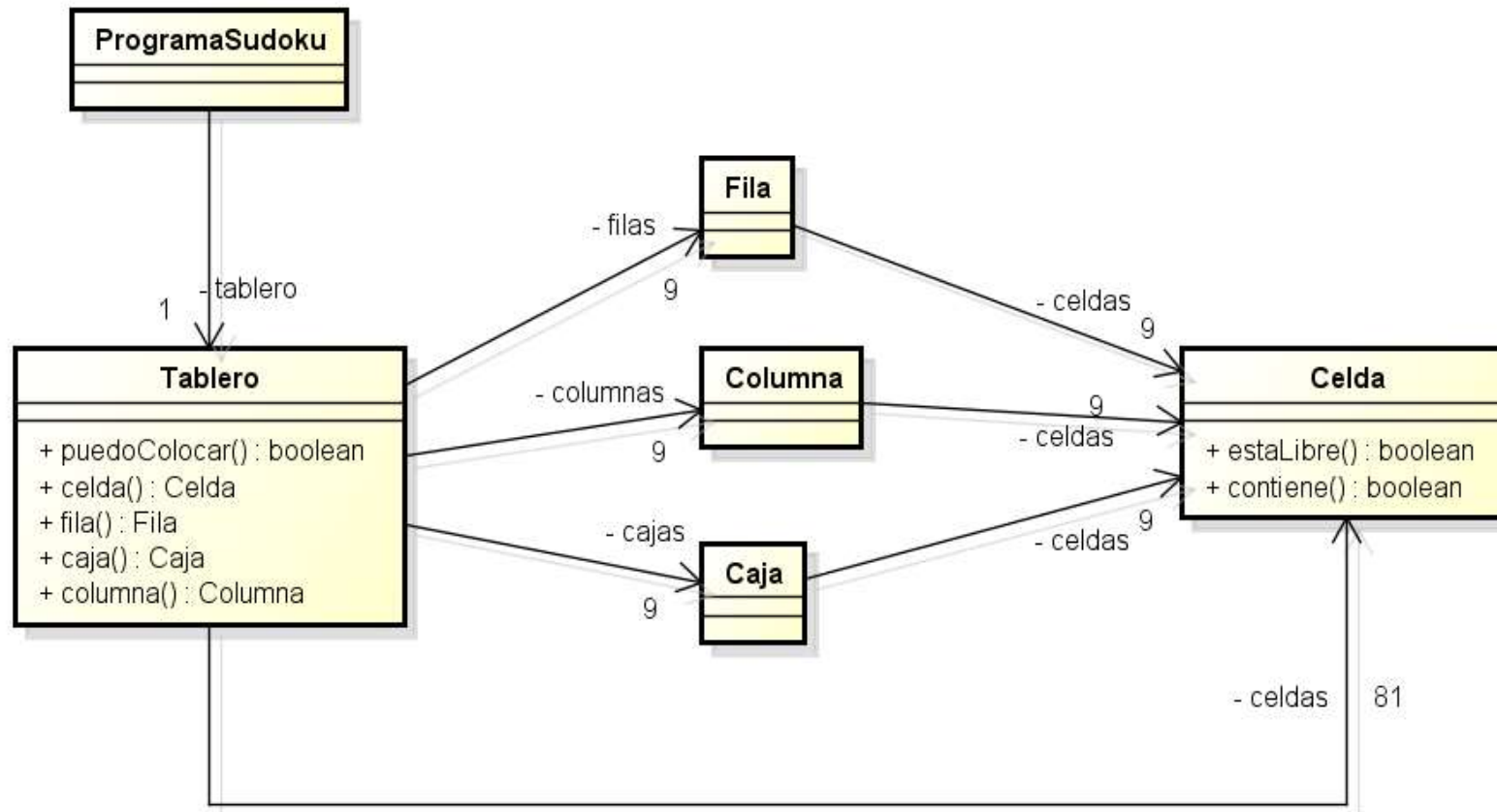
- El objeto receptor se obtiene como respuesta al envío de un mensaje a otro objeto

- El objeto cliente tiene una referencia al receptor: asociación

Asociación

- Forma de dependencia en la que el cliente tiene almacenada una referencia al servidor

Asociaciones entre objetos en diagramas de clases



Diagramas de clases derivan de los diagramas de secuencia

Recapitulación



Recapitulación: preguntas

¿Cómo se hace para que un objeto conozca al receptor de un mensaje a delegar?

(3 maneras)

¿Qué es una asociación?

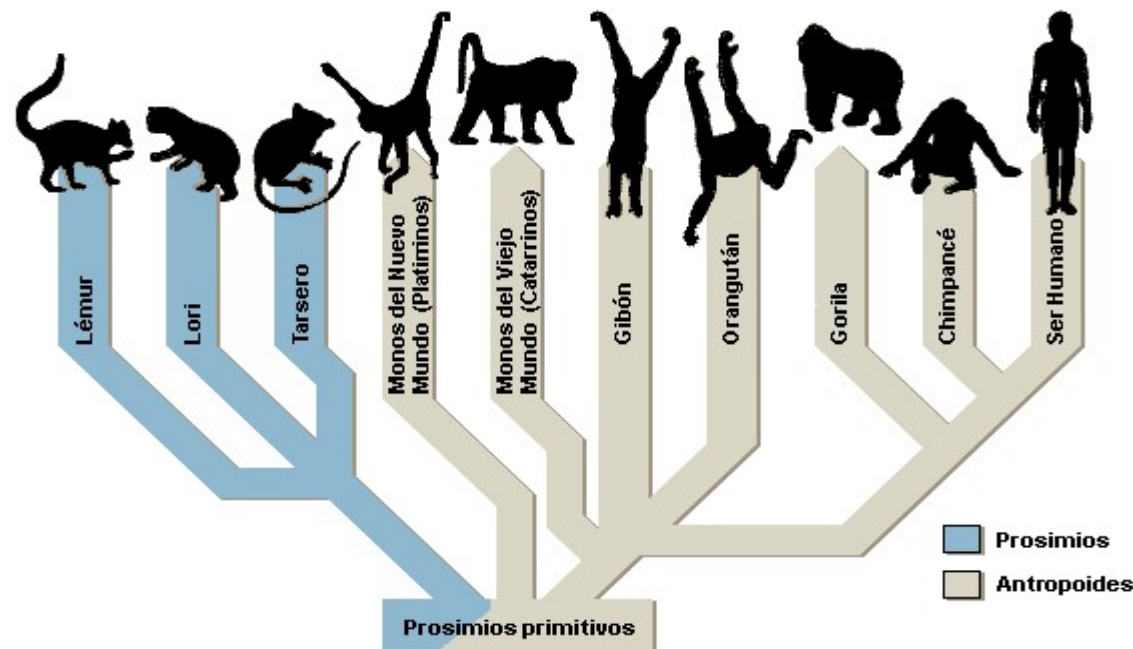


Pregunta complicada:
¿cómo aplicamos Test-First
cuando hay asociaciones?



Programación por diferencia

Programamos por diferencia cuando indicamos que parte de la implementación de un objeto está definida en otro objeto, y por lo tanto sólo implementamos las diferencias específicas



Programación por diferencia: herencia

Es una relación entre clases

=> Sólo en los lenguajes basados en clases

Relación generalización - especialización

Tipo más genérico: ancestro, madre, base

Tipo más específico: descendiente, hija, derivada

Ejemplos:

Perro deriva de Animal

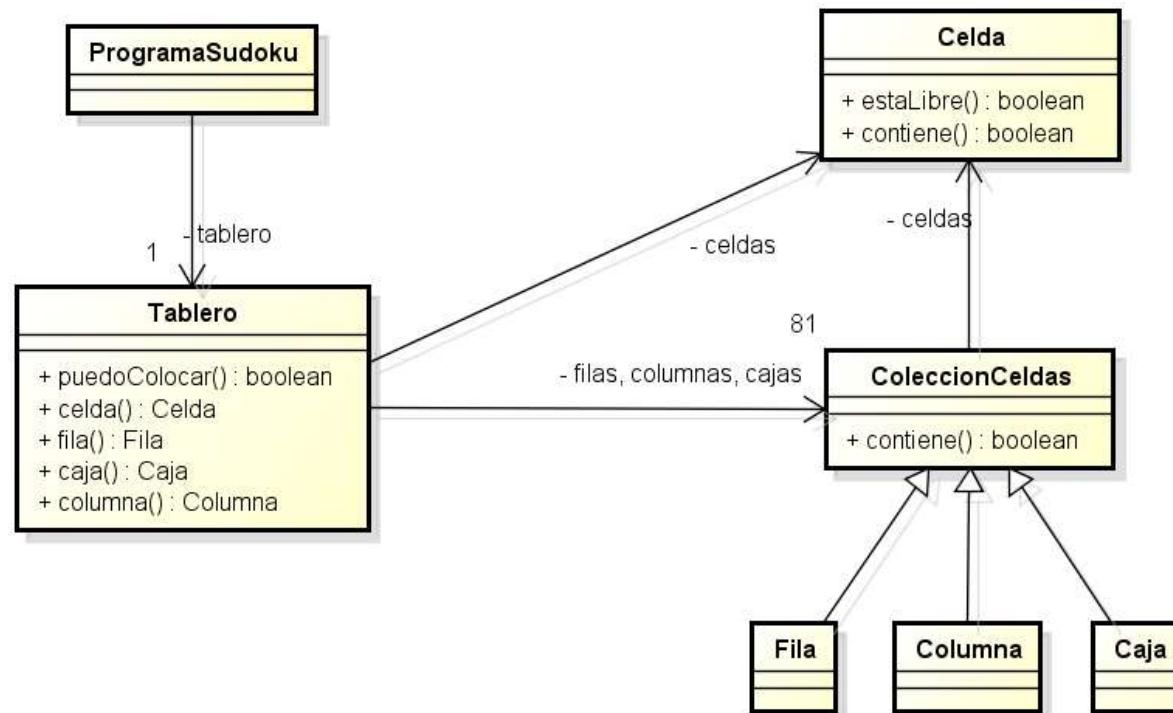
Lápiz desciende de ElementoDeEscritura

Número es madre de Complejo

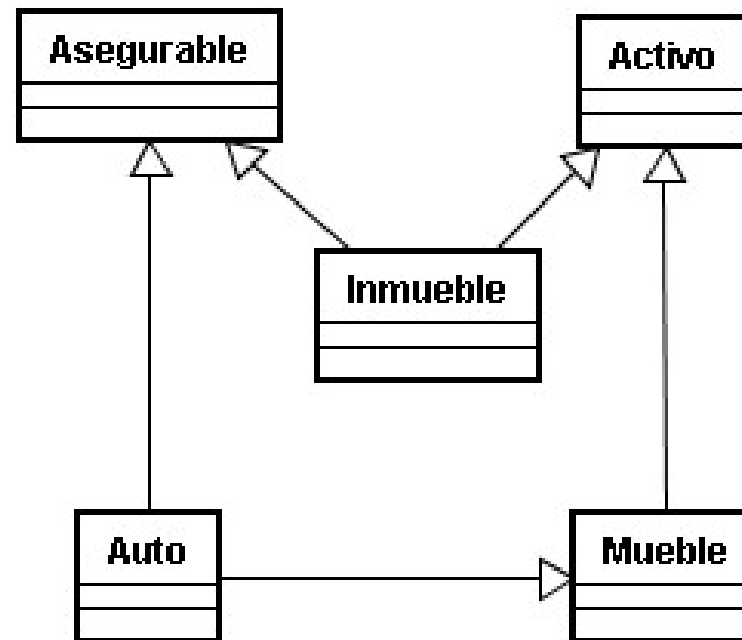
Herencia y POO

Si hay un comportamiento en la clase madre,
podemos usar ese comportamiento en cada
clase hija

¡Sin necesidad de escribir código nuevo!



Herencia múltiple (C++, Python, Eiffel)



Ojo: las clases no parecen excluyentes
Pero los objetos sólo pueden ser instancias de una
clase

Delegación vs. Herencia (1)

Herencia: relación “es un”

Toda instancia de una clase hija es instancia de la clase madre

Composición/agregación:

“contiene”, “hace referencia”, “es parte de”

Mito: en POO todo es herencia

Mal ejemplo: Stack en Smalltalk (y Java 1.0/1.1)

¡una pila no es una OrderedCollection (ni Vector)!

Herencia si se va a reutilizar la interfaz

Stack es un mal ejemplo

Delegación vs. Herencia (2)

La herencia resulta muy seductora:

¿Qué más cómodo que programar por diferencia?

Herencia

Cuando se va a reutilizar la interfaz tal como está

Delegación

Cuando se va a reutilizar sin mantener la interfaz

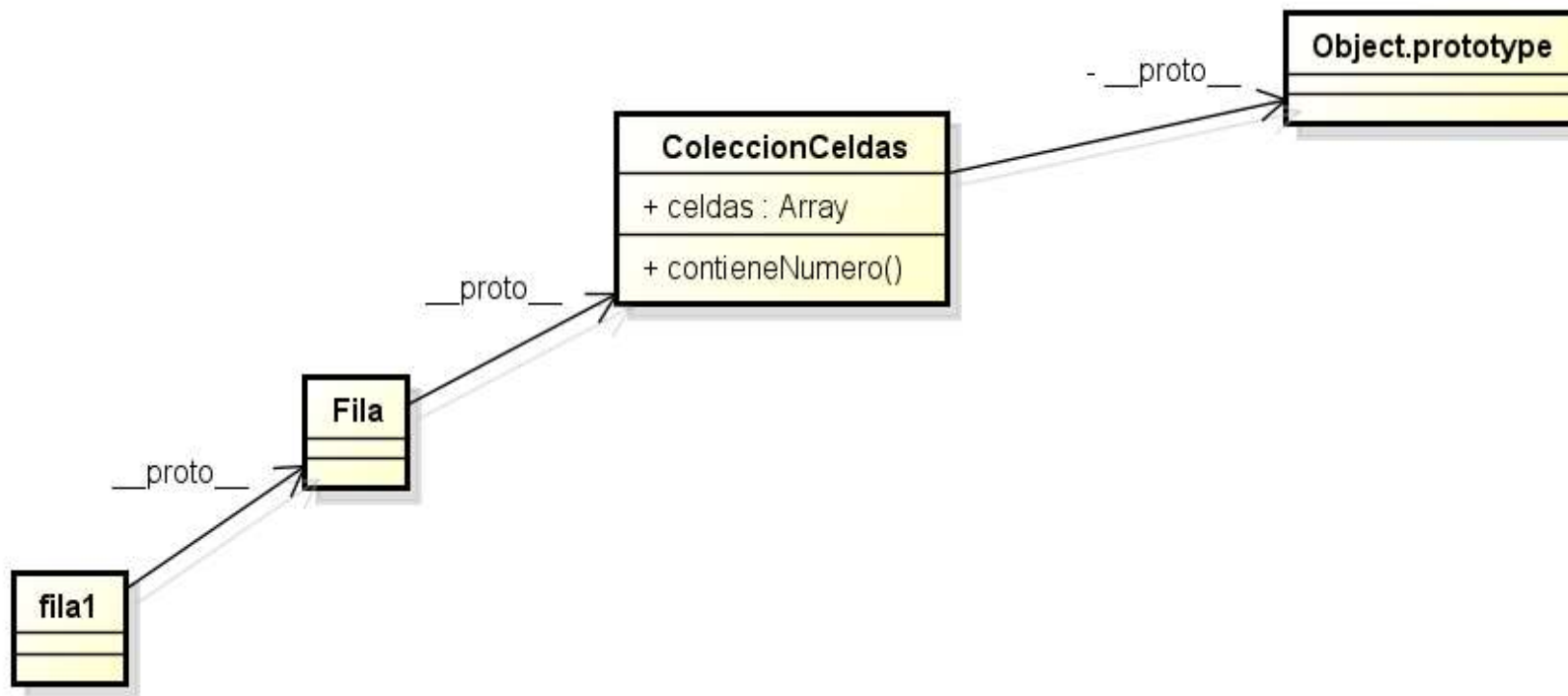


Programación por diferencia y lenguajes basados en prototipos

```
var ColeccionCeldas = {  
    celdas: [],  
    contieneNumero: function(numero) {  
        ...  
    }  
};  
var Fila =  
    Object.create(ColeccionCeldas);
```

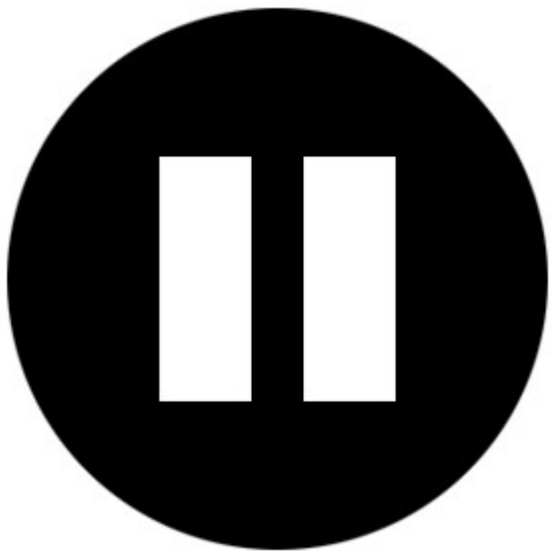
Fila puede usar los mensajes de ColeccionCeldas

Delegación de comportamiento en JavaScript



También es programar por diferencia

Recapitulación



Recapitulación: preguntas

¿Por qué no hay herencia en JavaScript?

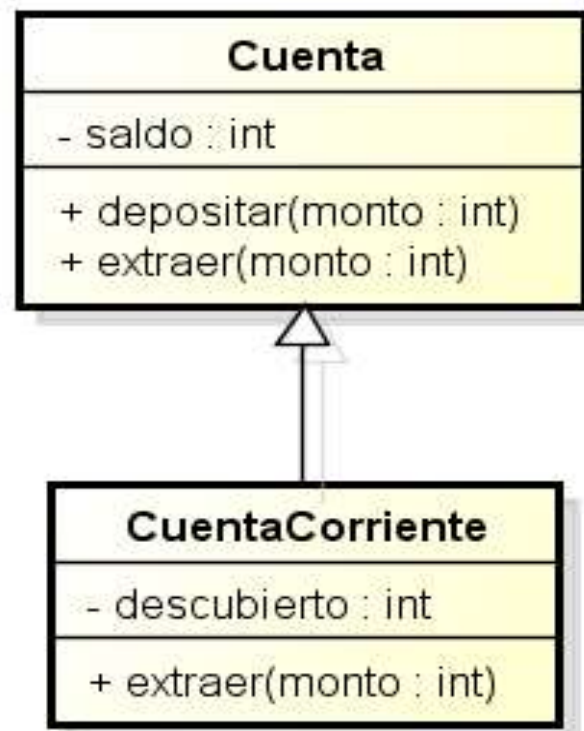
¿Qué se usa para reemplazarla?

¿Cuándo conviene heredar en vez de delegar?



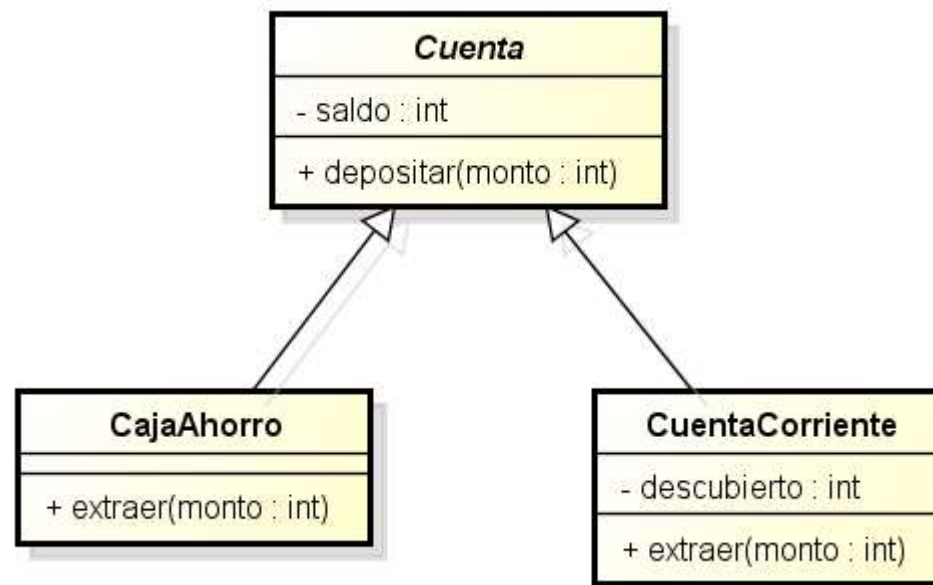
Redefinición

Para implementar de modo diferente la respuesta a un mismo mensaje



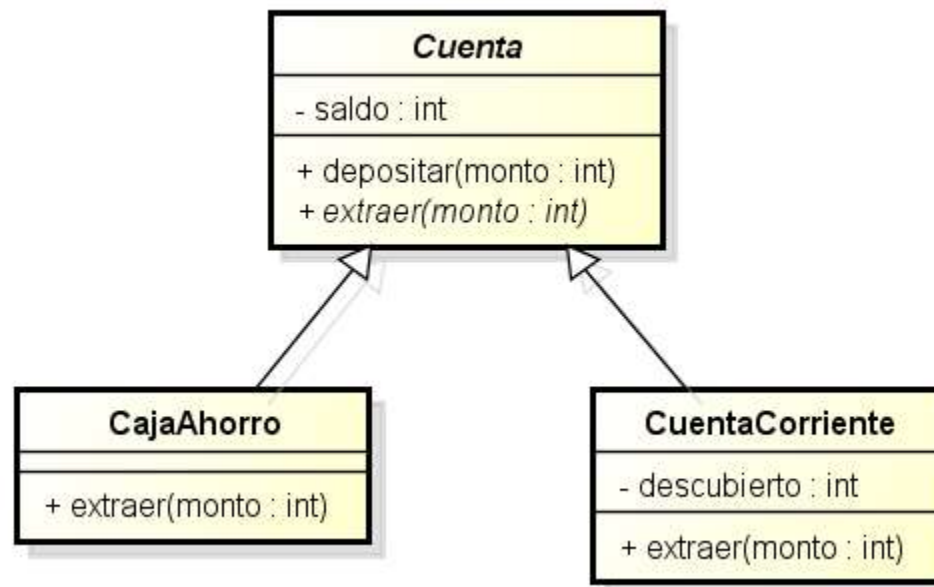
Clases abstractas

No pueden tener instancias en forma directa
Habitualmente porque sus descendientes cubren
todos los casos posibles



Métodos abstractos

No lo implementamos en una clase
Pero deseamos que todas las clases
descendientes puedan entender el mensaje



Herencia en Smalltalk y Java

Smalltalk:

```
ColeccionCeldas subclass: #Fila
```

Java:

```
class Fila extends ColeccionCeldas  
{...}
```

En ambos casos hay una clase madre por defecto, llamada Object

En Smalltalk hay que declararla:

```
Object subclass: #ColeccionCeldas
```


Visibilidad

Importante para garantizar encapsulamiento

Atributos y métodos privados

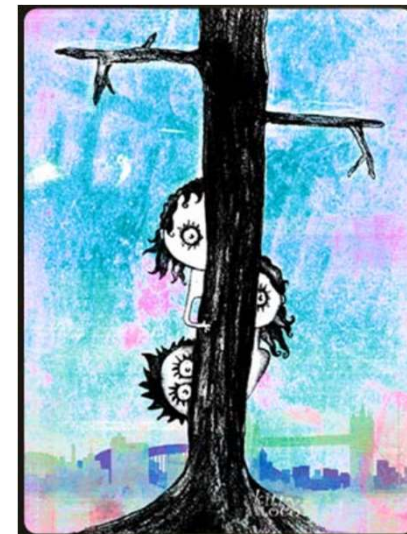
Sólo los puede usar el objeto receptor en su clase

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 y Java

Smalltalk

Todos los métodos son públicos

Todos los atributos son protegidos

Aunque se recomienda considerarlos privados

Hay convenciones para hacer métodos y atributos privados

¡Ojo que es una solución frágil!: depende de la memoria y la buena voluntad

Java

Visibilidad de atributos, métodos, clases

Directivas *public*, *private* y *protected*

Además hay visibilidad *de paquete*



Clases y métodos abstractos en Java

Directiva *abstract*

```
public abstract void extraer (int  
    monto) ;
```

```
public abstract class Cuenta {  
    ...  
}
```

Toda clase con un método abstracto debe ser abstracta

Todo se chequea en tiempo de compilación

Clases y métodos abstractos en Smalltalk

Método abstracto

```
extraer: monto  
    self subclassResponsibility.
```

En caso de invocar este método, se lanza una excepción *SubclassResponsibility*

Clase abstracta = clase con un método abstracto

Todo se chequea en tiempo de ejecución

Inicialización y herencia

```
public CuentaCorriente (String titular, int descubierto) {  
    this.titular = titular;  
    this.saldo := 0;  
    this.descubierto := descubierto;  
}
```

```
public CuentaCorriente (String titular, int descubierto)  
{  
    // "super" invoca al constructor de la clase madre  
    super(titular);  
    this.descubierto := descubierto;  
}
```

=> Invocar a los inicializadores en cascada

Inicialización y asociación

¿Es lo mismo? ¿Debemos inicializar en cascada?

Si por cada objeto que creamos, inicializamos en cascada los objetos referenciados: quedarían atados al objeto de arranque de la asociación

Eso se llama composición

No es lo que deseamos en todos los casos

Herencia y comprobación estática

```
Cuenta c = new Cuenta ( );
```

```
CuentaCorriente cc = new  
    CuentaCorriente ( );
```

```
Cuenta c = cc;
```

```
Cuenta ccc = new CuentaCorriente ( );
```

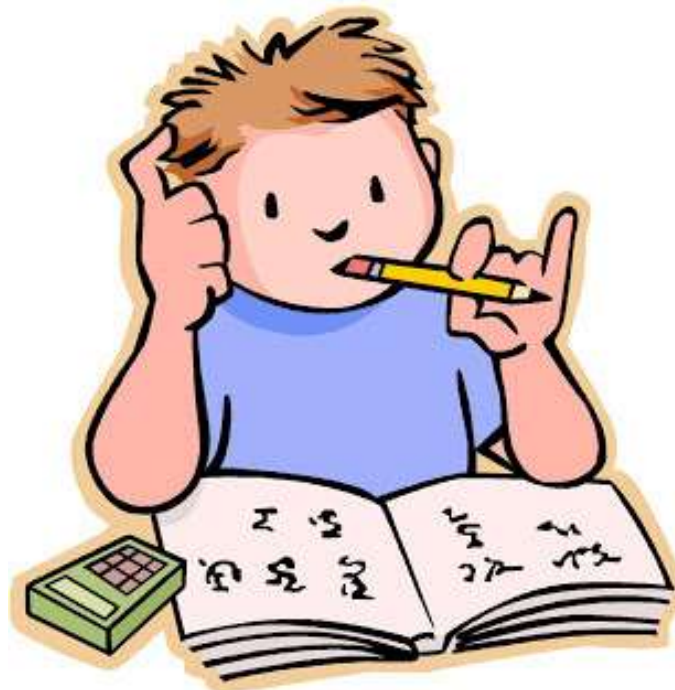
¿Qué comprobaciones hace el compilador?

Recapitulación



Recapitulación: preguntas

- ¿Para qué sirve un método abstracto?
- ¿Qué hay que hacer al inicializar en contextos de herencia?
- ¿Es lo mismo para las asociaciones?



Claves

Hay varias maneras de delegar

Y al menos dos maneras de programar por diferencia

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

Lecturas obligatorias

"The Art of Enbugging", Andy Hunt y Dave Thomas:

https://media.pragprog.com/articles/jan_03_enbug.pdf

"GetterEradiator", Martin Fowler:

<https://martinfowler.com/bliki/GetterEradiator.html>



fiuba

algoritmos

Qué sigue

Polimorfismo

Refactorización

Profundización

UML

Excepciones

Otros



fiuba
algoritmos