

fiuba

algo3

Polimorfismo Refactorización

Carlos Fontela
cfontela@fi.uba.ar

Contexto

Definimos

Encapsulamiento: cada objeto es responsable de responder a los mensajes que recibe, sin que quien le envía el mensaje tenga que saber cómo lo hace

Polimorfismo: capacidad de respuesta que tienen distintos objetos de responder de maneras diferentes a un mismo mensaje

Hay consecuencias...

Polimorfismo y refactorización



Temario

Polimorfismo en general

Los lenguajes de comprobación estática:
vinculación tardía e interfaces

Refactorización

TDD completo

Polimorfismo

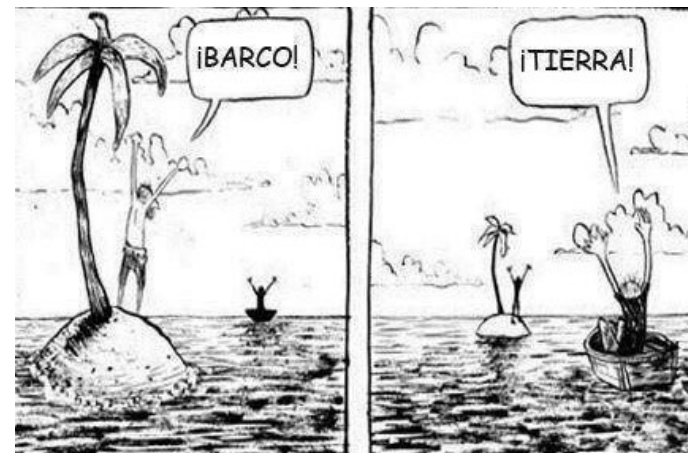
Capacidad que tienen distintos objetos de responder de maneras diferentes a un mismo mensaje

celda >> contiene(7)

fila >> contiene(7)

cajaAhorro >> extraer(100)

cuentaCorriente >> extraer(100)



Polimorfismo de objetos

```
var celdaLibre = {  
  contiene : function (valor) {  
    return false;  
  }  
}  
  
var celdaOcupada = {  
  contiene : function (valor) {  
    return (this.numero == valor);  
  }  
}
```

¿Y en lenguajes con clases?

Polimorfismo en lenguajes con clases

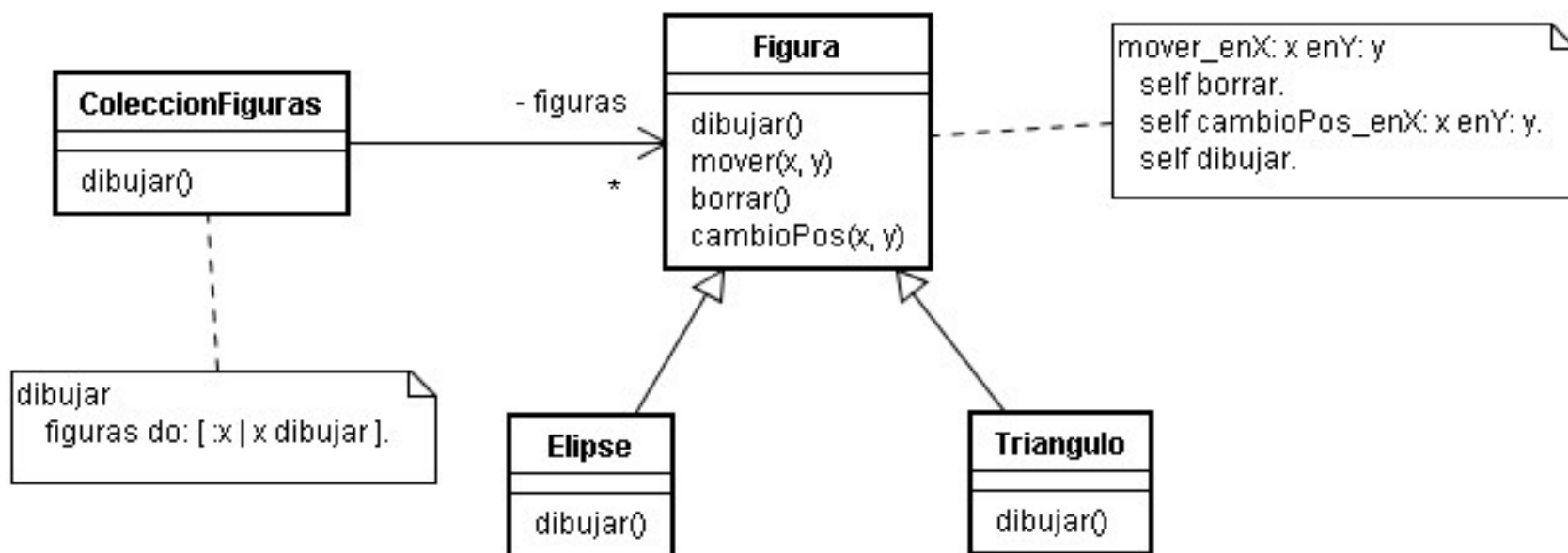
Distintos objetos responden de maneras diferentes a la llegada de un mensaje, basándose en la clase de la cual son instancias

cajaAhorro extraer: 100.

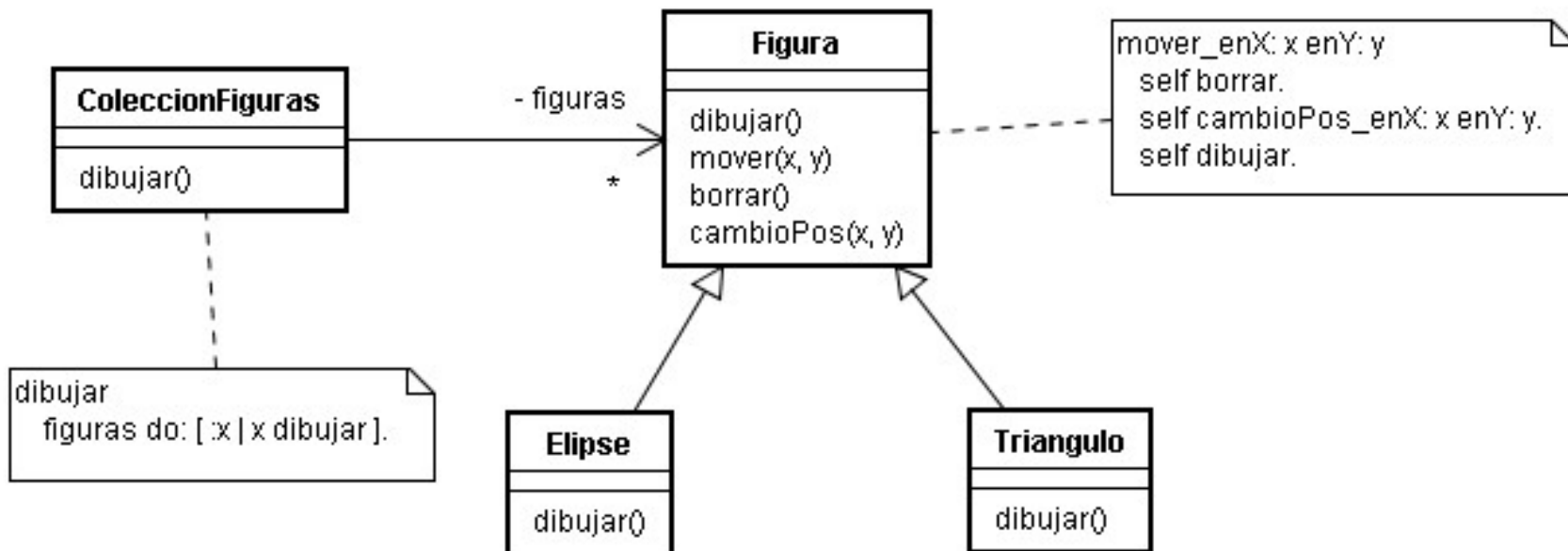
cuentaCorriente extraer: 100.



Polimorfismo: analizar en Smalltalk

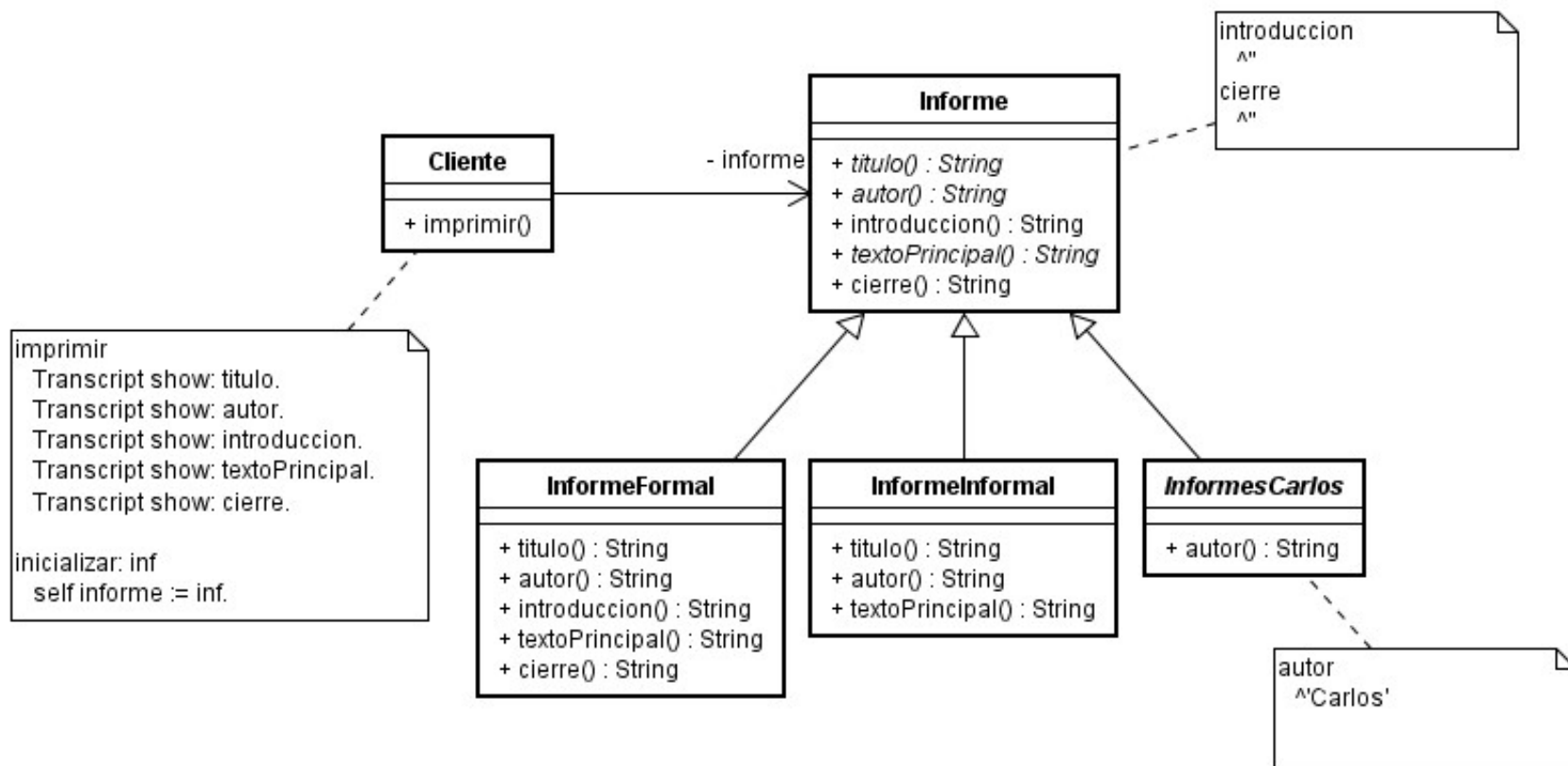


Polimorfismo: analizar en Java



¿Y si dibujar de Figura fuera abstracto? ¿Qué pasa con la verificación estática?

Polimorfismo: analizar



Algo un poco más complejo...

Ajedrez: cada tipo de pieza tiene un comportamiento distinto

=> Implementación obvia (?)

Coronación: ¿qué ocurre?

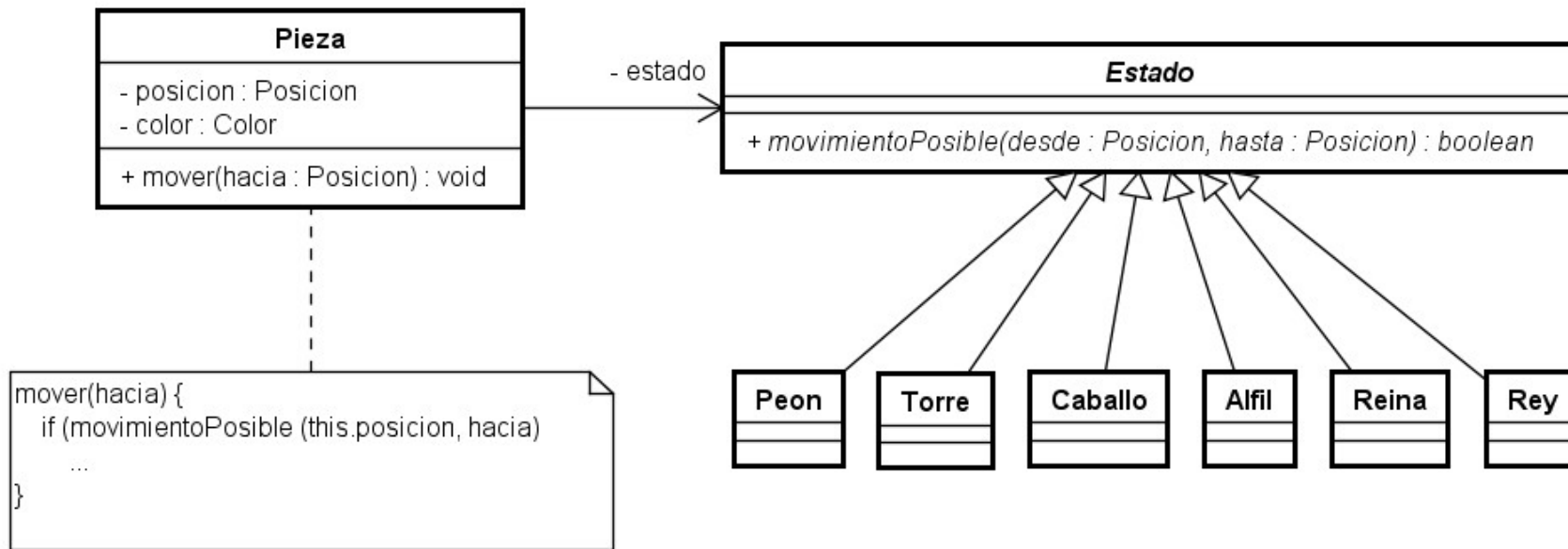
Posible interpretación: un objeto cambia de clase (su comportamiento varía luego de un cambio de estado)

... hay otros planteos: no es el mismo objeto

Quedémonos con el inicial: ¿puede un objeto cambiar su clase?



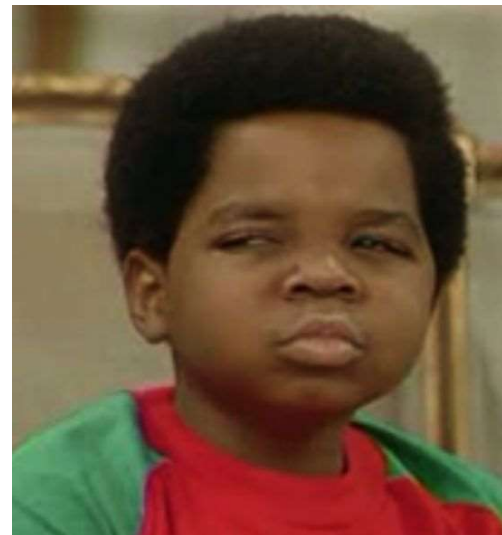
¿Y si vemos al tipo de pieza como un estado particular?



Coronación: `estado = new Reina ();`

¿Qué hicimos?

Hemos encapsulado un método
(*movimientoLegal*) en un objeto
Podemos tratar a los métodos como objetos
En Smalltalk los bloques pueden ser objetos
En C# hay “delegados”
En Java 8, expresiones lambda



Soluciones típicas

Se las llama “patrones”

Usamos el patrón State en el ejercicio del ajedrez
Y el patrón Template Method en el ejemplo de los informes

Y algo parecido a Command o Strategy
en el ejemplo de la colección de figuras

Ya volveremos



Para pensar...



¿Vinculación tardía?

Se plantea que el polimorfismo funciona porque la decisión del método a invocar se toma en tiempo de ejecución y no antes

Tiene sentido en lenguajes de comprobación estática

```
cajaAhorro.extraer(100);
```

```
cuentaCorriente.extraer(100);
```

En Java se da por defecto

Si no lo queremos, poner *final* al método

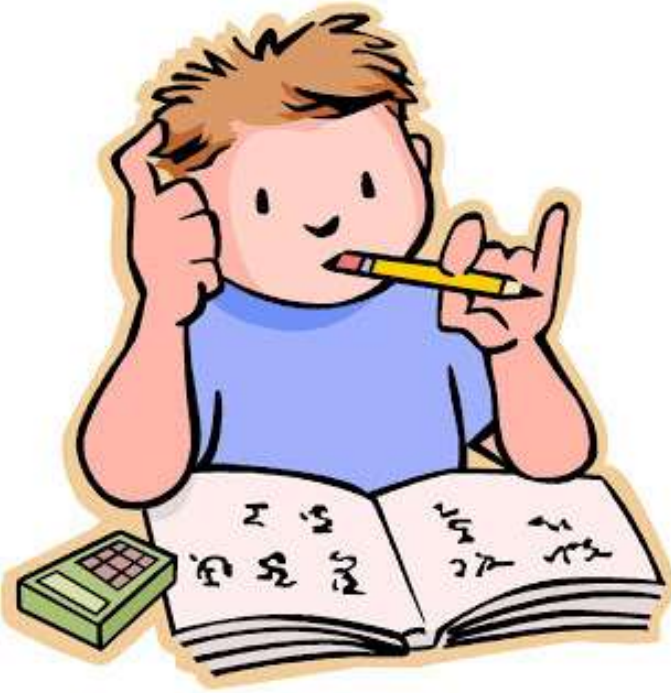
En C++ y C# se declaran métodos virtuales

Recapitulación



Recapitulación: preguntas

- ¿Por qué contraponemos el uso de “if” al polimorfismo?
- ¿Para qué querríamos polimorfismo sin herencia?
- ¿Polimorfismo es sinónimo de vinculación tardía?



Polimorfismo y herencia: ¿deben ir juntos?

```
Collection <Cuenta> cuentas = new ArrayList  
<Cuenta> ( );
```

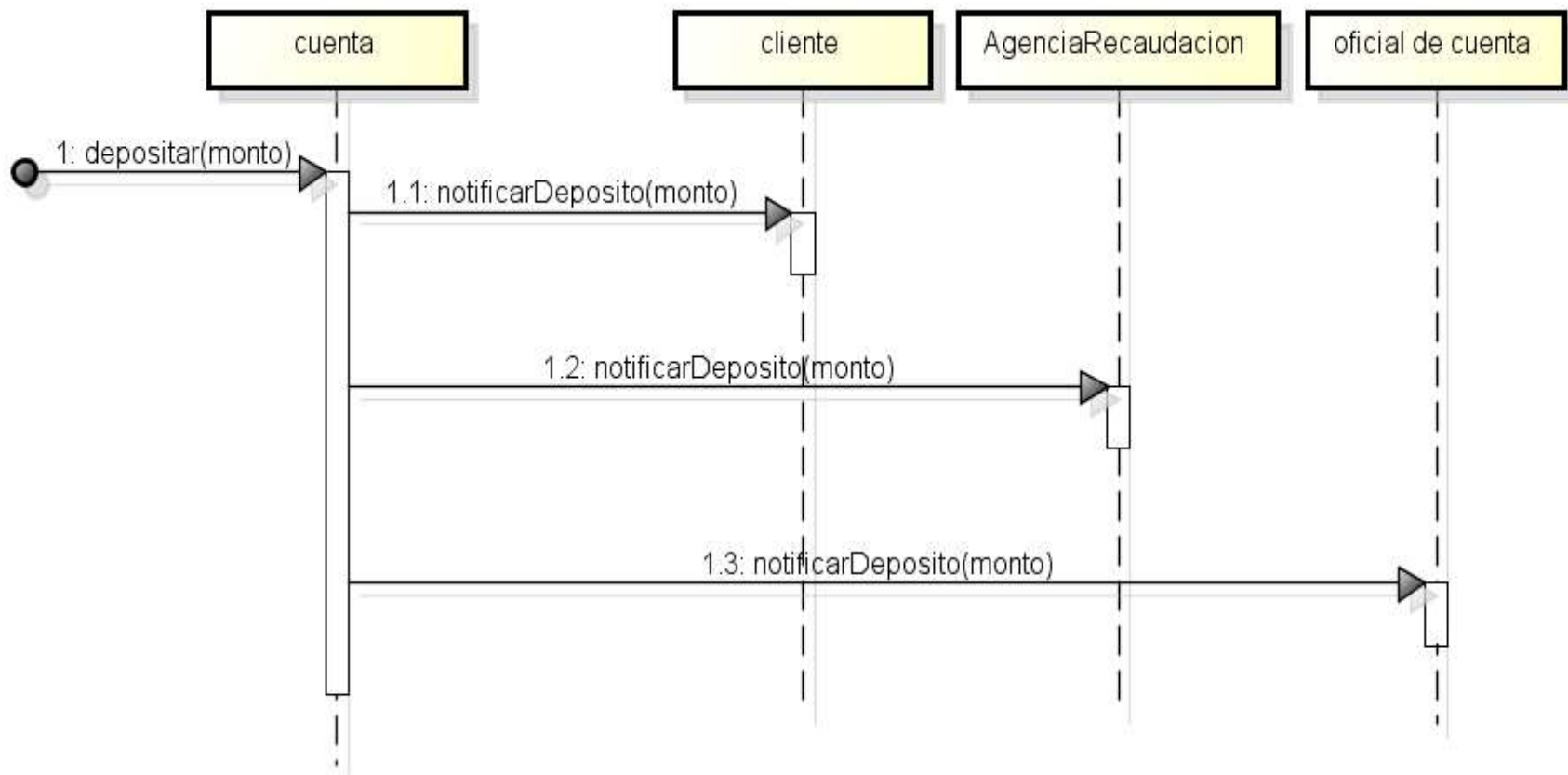
```
...
```

```
for (Cuenta c : cuentas)  
    c.extraer(100);
```

Compilador verifica que la clase de c tenga un
método *extraer*

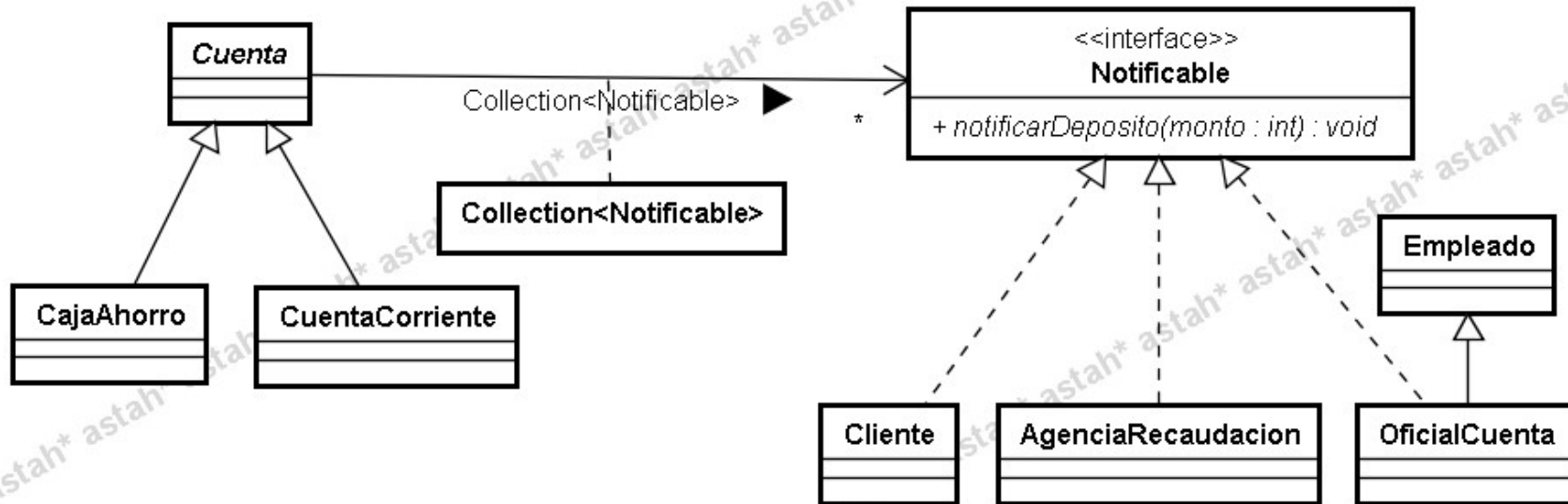
Tiene sentido en lenguajes de comprobación
estática

Polimorfismo sin herencia



Polimorfismo sin herencia ¿en Java?

```
public void depositar(int monto) {
    ...
    for (Notificable e : entidadesNotificar)
        e.notificarDeposito(monto);
}
```



¿Qué quiere decir <<interface>>?

Interfaces

Como mecanismo necesario para el
polimorfismo sin herencia



Interfaces: clases muy abstractas

Son como clases

- Abstractas

- Con todos los métodos abstractos

- Sin atributos (sin estado)

Ejemplo

```
public interface Notificable {  
    /*public abstract*/ void notificarDeposito();  
}
```

Pueden heredar de otras interfaces

```
public interface RecibeMails extends Notificable {  
    void enviarMail();  
}
```


Herencia de interfaces

Uso

```
public class OficialCuenta extends Empleado  
    implements Notificable {  
    ...  
}
```

Corolario

Si una clase declara implementar una interfaz y no implementa (redefine) uno de sus métodos es abstracta

Interfaces: protocolos

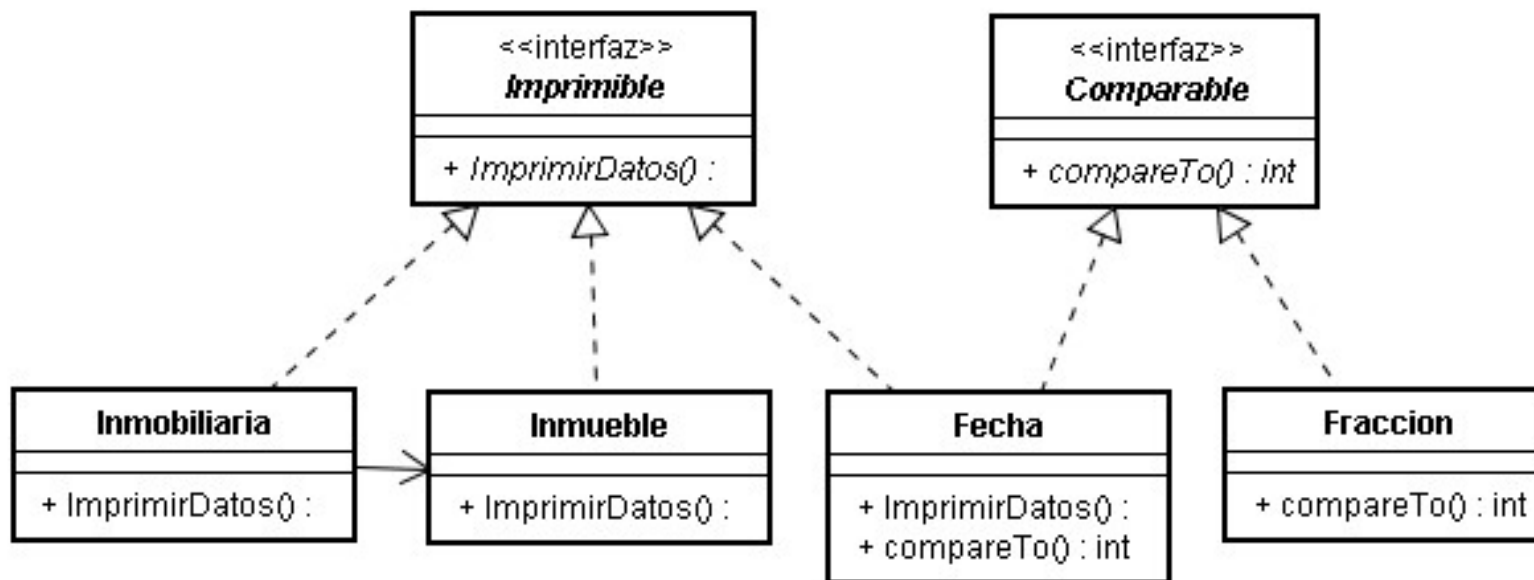
Son grupos de firmas de métodos

Sin implementar

Indican maneras de comunicarse con los objetos

Una clase puede implementar varias

Ojo con los conflictos de nombres



Interfaces y polimorfismo

Cada objeto puede tener distintas formas de uso, según el tipo con el que se lo accede

Fecha f = new Fecha(20,6,1964);

Imprimible i = f;

Comparable c = f;

Serializable s = f;

Todos se refieren al mismo objeto

Pero “lo ven” distinto

Cada variable sólo puede usar los métodos de su interfaz

Ojo: ¡sólo puedo instanciar clases!



¿Qué implica?

El tipo de la variable define la interfaz que puedo usar

```
Fecha f = new Fecha(20,6,1964);
```

```
Imprimible i = f;
```

```
Comparable c = f;
```

```
i.imprimir();
```

```
c.compareTo(c2);
```

```
f.imprimir();
```

```
f.compareTo(f2);
```

```
...
```

¿Qué es una interfaz?

Visión de lenguaje

Una clase “muy abstracta” que se puede usar para herencia múltiple

Visión desde el uso

Un tipo de datos que permite que ver a un mismo objeto con distintos tipos

=> Cada tipo implica un comportamiento



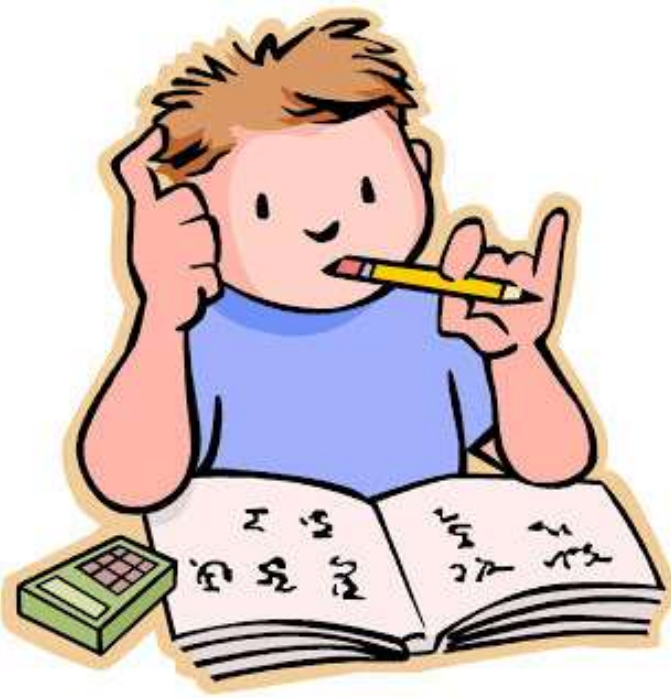
Recapitulación



Recapitulación: preguntas

¿Para qué sirven las interfaces? (por ahora...)

¿Por qué no hay interfaces en Smalltalk?



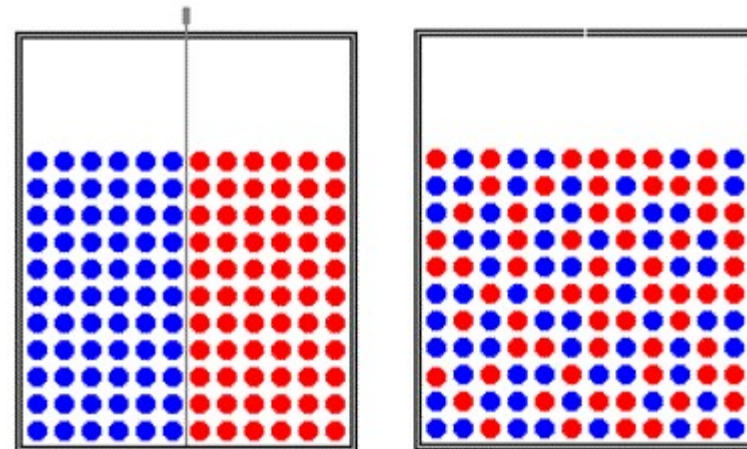
Refactorización: mejora del código



Entropía creciente

Todo código va empeorando su calidad con el tiempo

=> entropía, degradación



Refactorizaciones

Mejorar código, haciéndolo más comprensible

Sin cambiar funcionalidad

Refactorización

“Refactoring”

Mejorar el código ya escrito

¿Cómo?

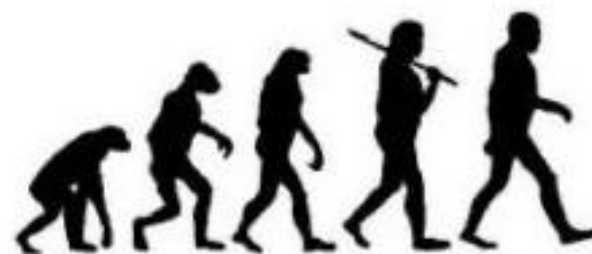
Modificar estructura interna

Sin modificar comportamiento observable

Ejemplos:

Eliminar código duplicado

Introducir polimorfismo



Refactoring
Improving the Design of Existing Code

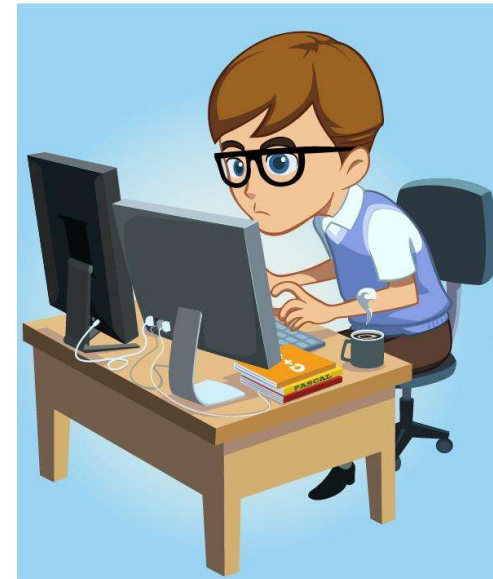
¿Hacemos un ejemplo?

CajaAhorro >> extraer(monto)

```
if (monto <= 0)
    throw new MontoInvalido();
if (monto > this.getSaldo())
    throw new SaldoInsuficiente();
this.saldo -= monto;
```

CuentaCorriente >> extraer(monto)

```
if (monto <= 0)
    throw new MontoInvalido();
if (monto > this.getSaldo() + this.descubierto)
    throw new SaldoInsuficiente();
this.saldo -= monto;
```



Para qué

Mejorar código, haciéndolo más comprensible

- Para modificaciones

- Para depuraciones

- Para optimizaciones

Mantener alta la calidad del código

- Si no, se degrada

A la larga, aumenta la productividad



Cuándo refactorizamos

Actitud constante

Consecuencia de revisiones de código

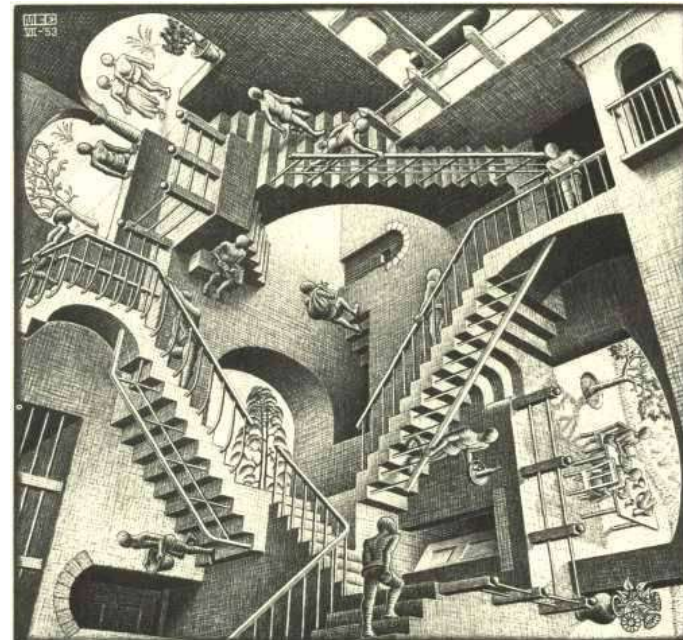
Antes de modificar código existente

Después de incorporar funcionalidad

Antes de optimizar

Ojo: optimizar \neq refactorizar

Durante depuraciones



Condiciones previas

Riesgo alto

Máxima: “Si funciona, no lo arregle”



Un paso por vez

Pruebas automatizadas

Escribirlas antes de refactorizar

Y correrlas luego de cada pequeño cambio

Problemas y refactorización

“Bad smells in code” (malos olores), los llama Fowler

Son indicadores de que algo está mal, y se solucionan con refactorizaciones

Hay catálogos

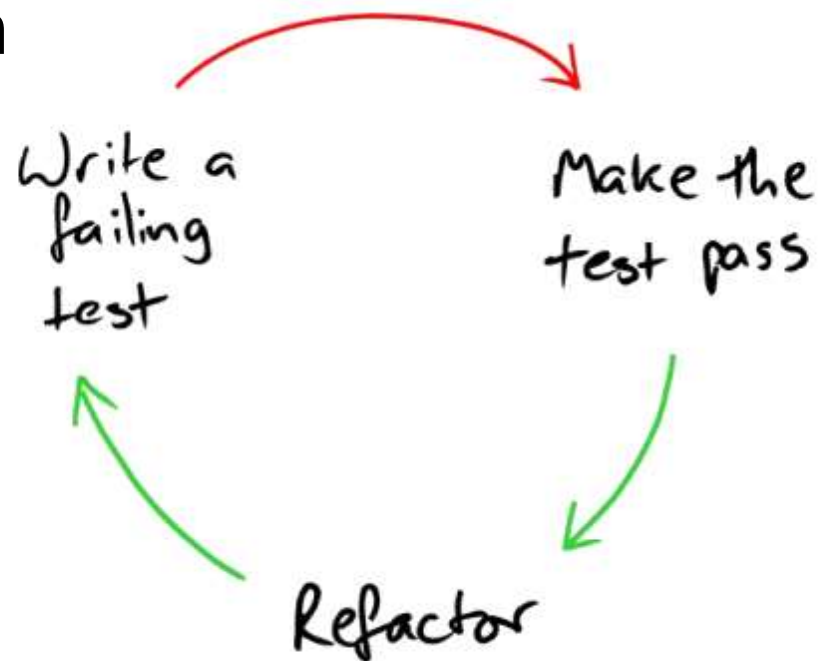
Smalltalk desde los inicios

Java: lenguaje habitual



TDD: recapitulación

Test-Driven Development =
Test-First +
Automatización +
Refactorización



Recapitulación

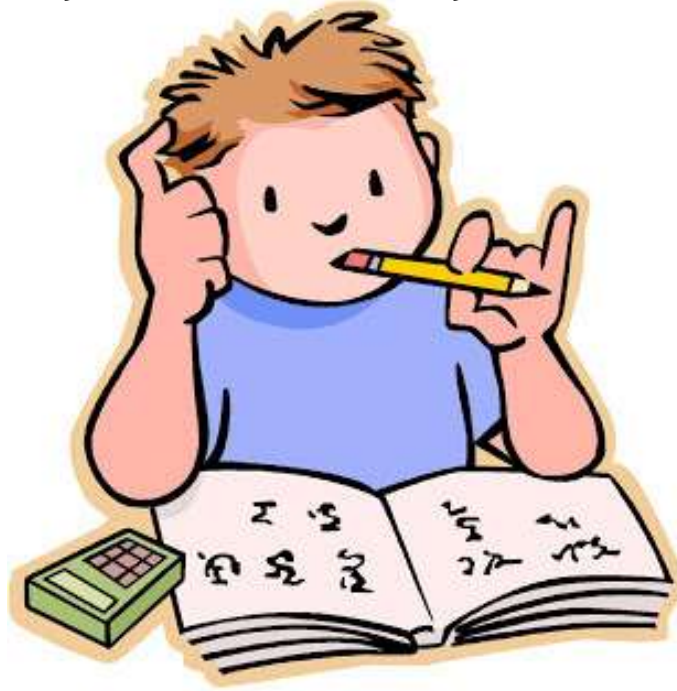


Recapitulación: preguntas

¿Para qué refactorizamos?

¿Por qué no hacemos varias refactorizaciones seguidas?

¿Qué relación hay entre TDD y refactorización?



Claves

Polimorfismo = distintos comportamientos
para un mismo mensaje

Polimorfismo seguro y sin herencia:
interfaces

Manejar la entropía => refactorización

Lectura obligatoria

“Replace Conditional with Polymorphism”,
<https://sourcemaking.com/refactoring/replace-conditional-with-polymorphism>



Qué sigue

Profundización

UML

Excepciones

Otros

Calidad de código

