

Reuso de Código

Herencia vs. Composición

Abstracción de comportamiento en método de la superclase

Reuso de Código

Herencia vs. Composición

Herencia de Clases

- Herencia total: debo conocer todo el código que se hereda -> *Reutilización de Caja Blanca*
 - Usualmente debemos redefinir
 - Los cambios en la superclase se propagan automáticamente a las subclases
 - Herencia de Estructura vs. Herencia de comportamiento
 - Es útil para extender la funcionalidad del dominio de aplicación

Composición de Objetos

- Los objetos se componen en forma Dinámica -> Reutilización de **Caja Negra**
 - Existe una relación de conocimiento entre el objeto compuesto y su parte: *relación de composición*
- Los objetos pueden reutilizarse a través de su **interfaz** (sin conocer el código)
- A través de las relaciones de composición se pueden delegar responsabilidades entre los objetos

Un simple ejemplo: Clase Cola

- Cola es una estructura de datos con **comportamiento específico**.
 - Pop, push, top, isEmpty
- ¿Cómo implementamos la clase Cola ?
 - ¿cómo subclase de OrderedCollection?
 - ¿Tiene sentido heredar todo el comportamiento de OrderedCollection?
 - Habría que anular o redefinir demasiado comportamiento?
 - Cola se compone de una OrderedCollection para mantener sus elementos?



Un simple ejemplo: Clase Cola

Cola como subclase de Object

```
Cola>>initialize
```

```
    elementos := OrderedCollection new.
```

```
Cola>> push: unObjeto
```

```
    elementos addLast: unObjeto
```

```
Cola>> pop
```

```
    ^elementos removeFirst
```

```
Cola>> top
```

```
    ^elementos first
```

```
Cola>> isEmpty
```

```
    ^elementos isEmpty
```

Cola
- elementos
+ push :() + pop() + top() + isEmpty()



Un simple ejemplo: ColaDoble y Pila

- Una ColaDoble es una secuencia de elementos a la que se puede agregar y sacar elementos por ambos extremos.
- Como implementaría la clase ColaDoble usando la clase Cola?
- Como implementaría la clase Pila usando ColaDoble?
- Ejercicio para la casa....

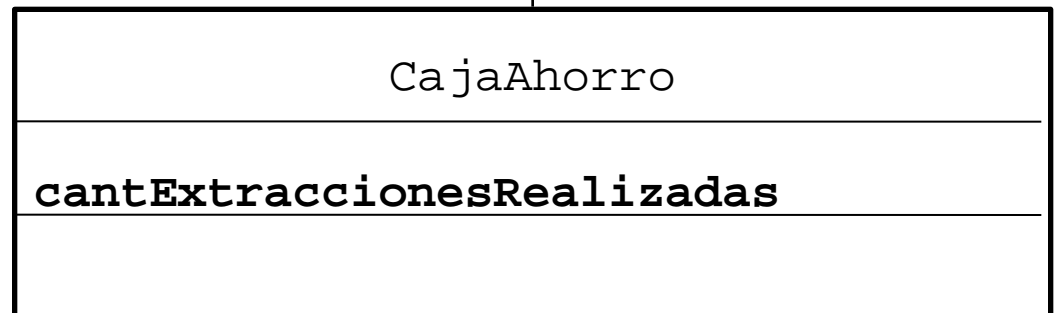
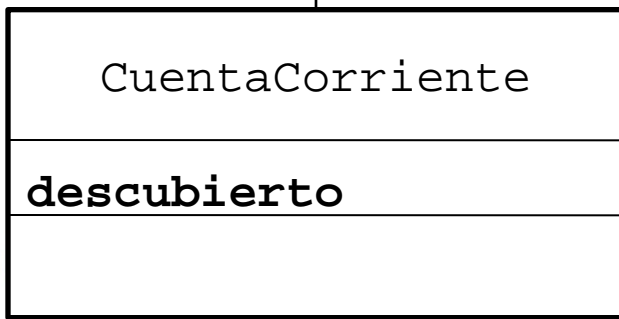
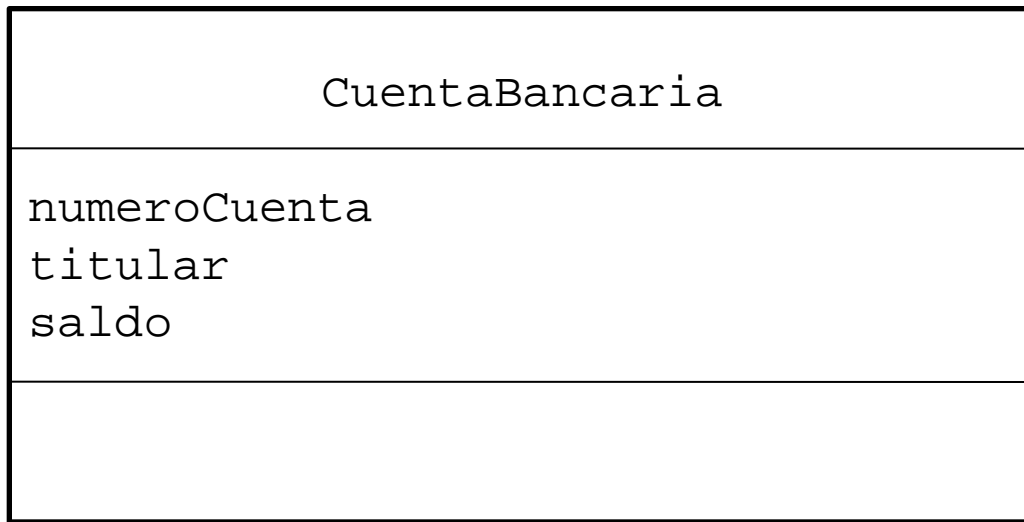


Reuso de Código

Abstracción de comportamiento
en método de la superclase

#extraer: en la clase Cuenta Bancaria

- Tener en cuenta:
 - chequear que haya saldo suficiente
 - Las cajas de ahorro cuentan las cantidades de extracciones
 - Las cuentas corrientes tienen un monto en descubierto permitido.

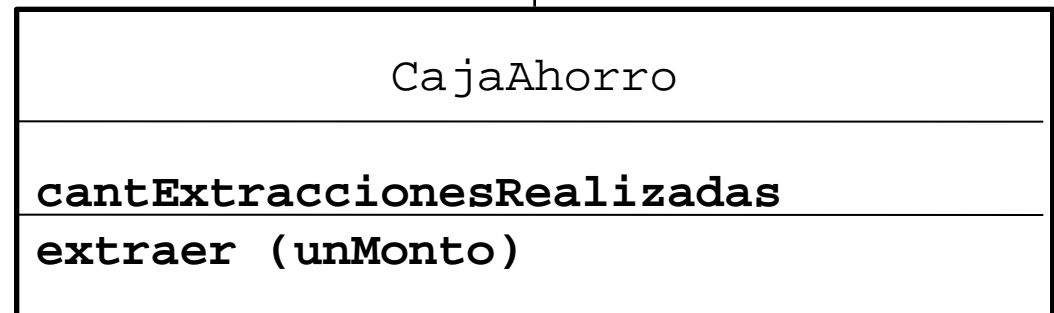
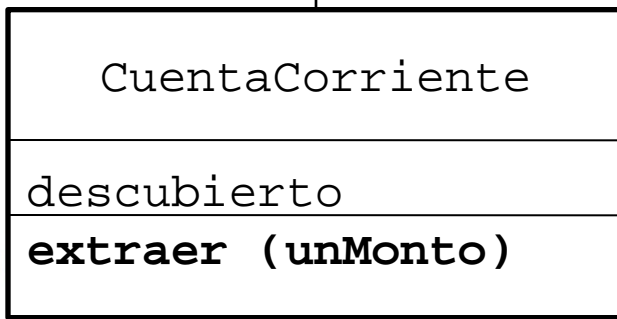
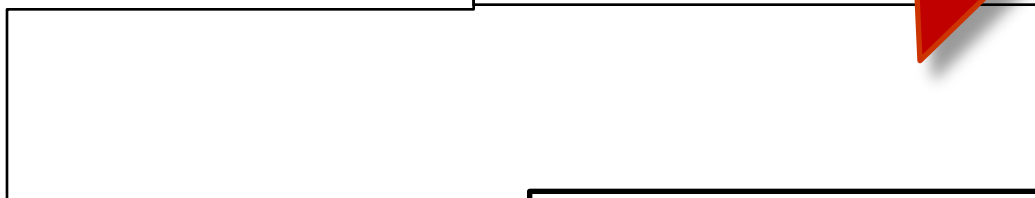
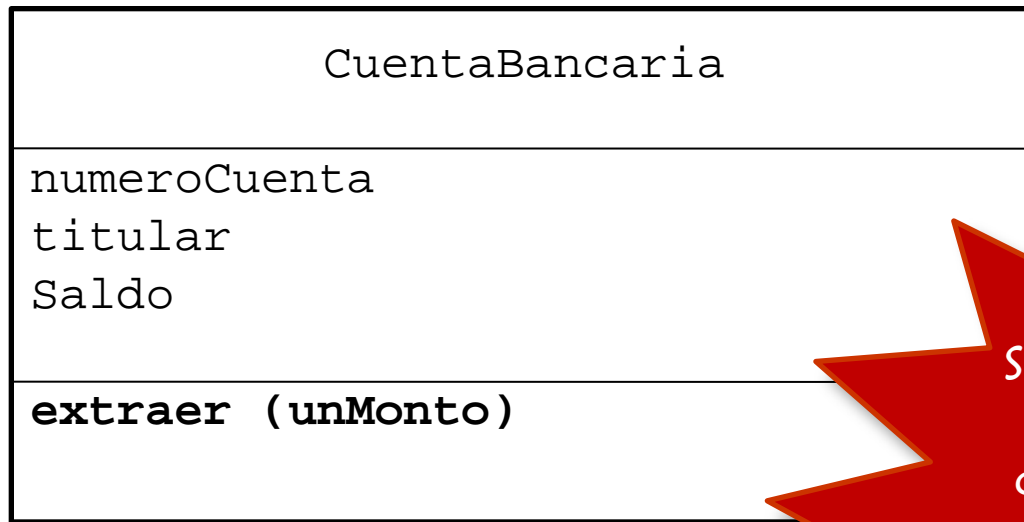


Ejemplo: Implementación de #extraer:

Opción 1: El método en la superclase puede estar **vacío**, indicando que debe implementarse en las subclases.



Opción 1: #extraer vacío en la superclase



Opción 1: #extraer vacío en la superclase

CuentaBancaria>> extraer: unMonto

^self subclassResponsibility

CajaAhorro>> extraer: unMonto

self *hayFondos*: unMonto

if True: [self *incrementarExtr.*

self *decrementarSaldo*: unMonto]

CuentaCorriente>> extraer: unMonto

self *hayFondos*: unMonto

if True: [self *decrementarSaldo*: unMonto]

CajaAhorro>> hayFondos: unMonto

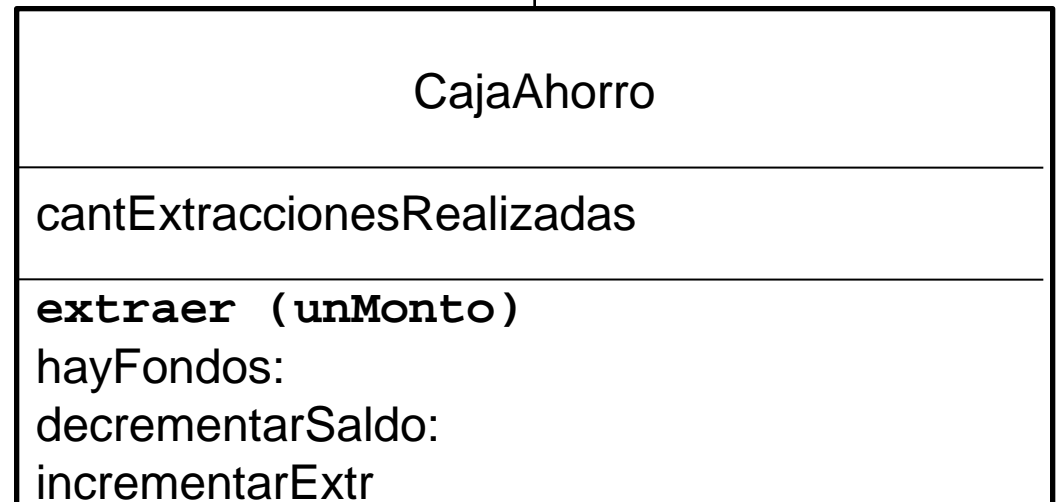
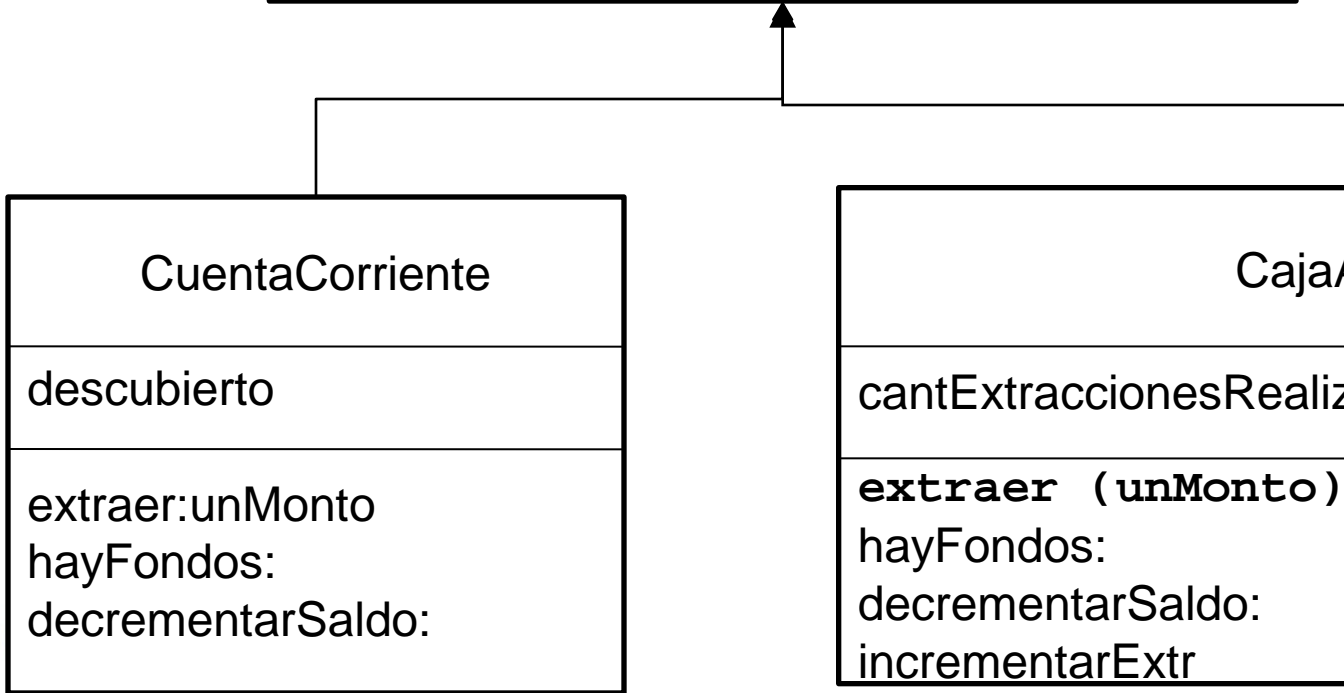
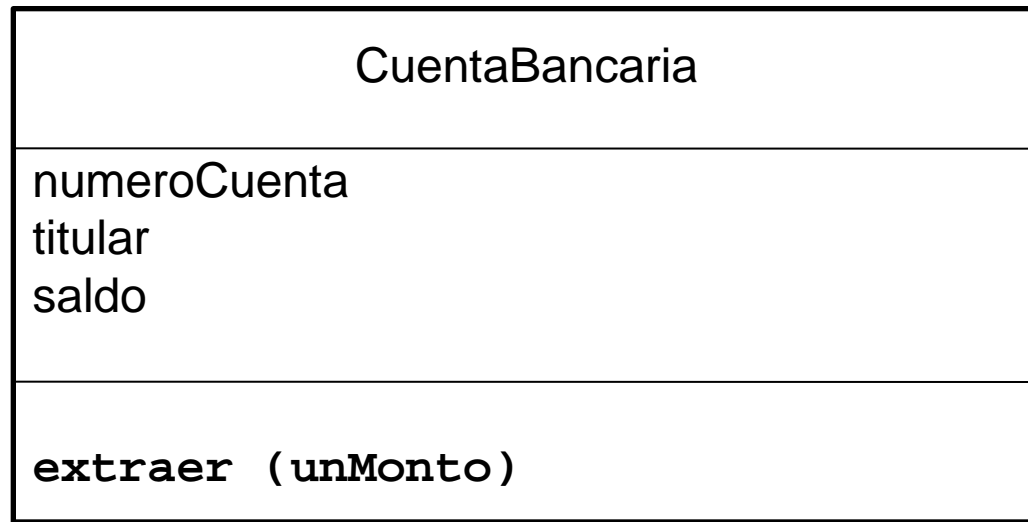
^saldo >= unMonto

CuentaCorriente>> hayFondos: unMonto

^saldo >= unMonto + descubierto



Opción 1: #extraer vacío en la superclase



Ejemplo: Implementación de #extraer:

Opción 2: Podemos reusar código en común usando *super*

Teníamos.....

CuentaBancaria>> extraer: unMonto

^self subClassResponsability

CajaAhorro>> extraer: unMonto

self *hayFondos*: unMonto

if True:[self *incrementarExtr.*

self *decrementarSaldo*: unMonto

CuentaCorriente>> extraer: unMonto

self *hayFondos*: unMonto

if True:[self *decrementarSaldo*: unMonto]

CajaAhorro>> *decrementarSaldo* : unMonto

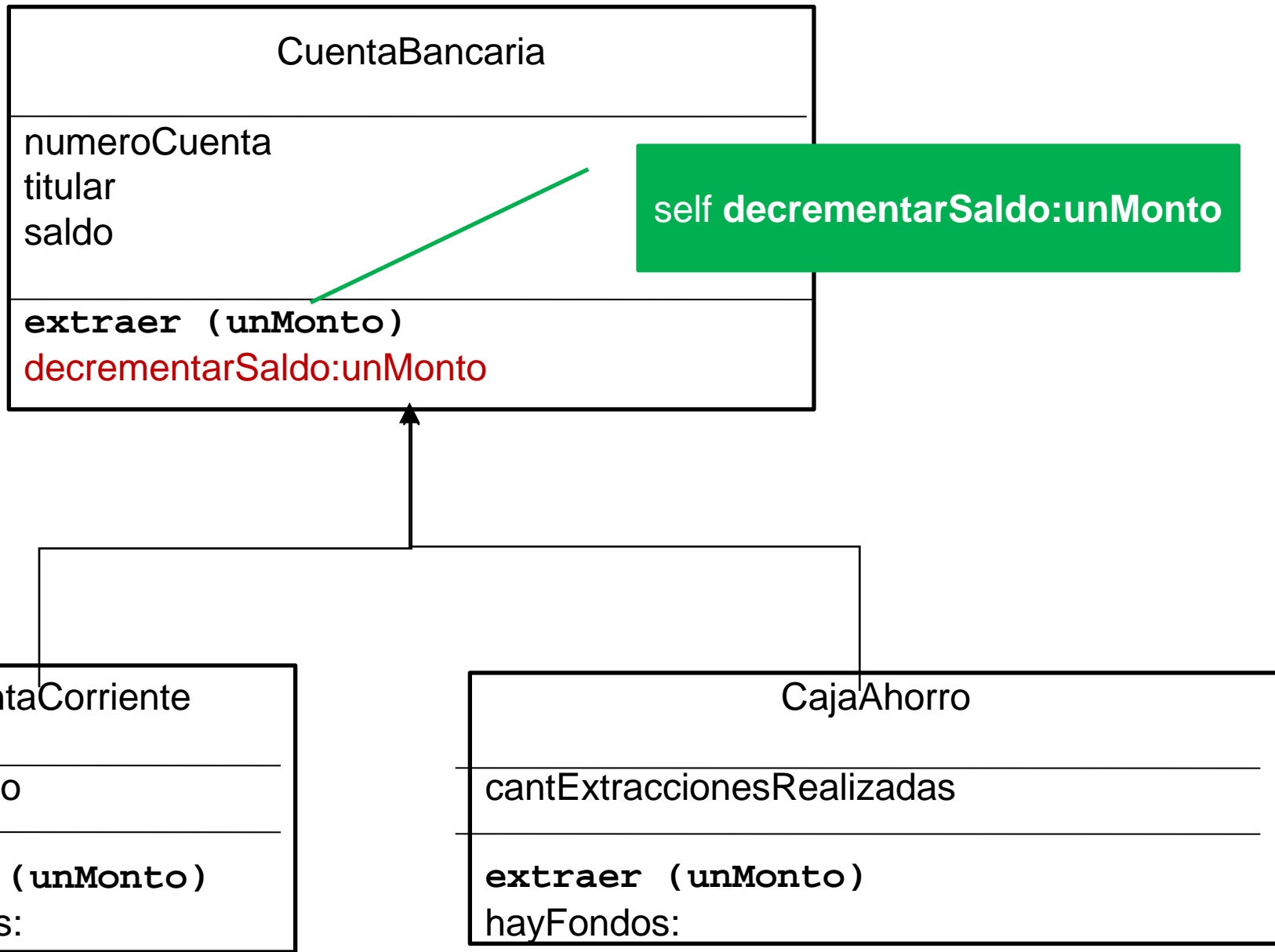
^saldo:= saldo - unMonto

CuentaCorriente>> *decrementarSaldo* : unMonto

^saldo:= saldo - unMonto



Opción 2: #extraer redefinido en las subclases (uso de super)



Opción 2: #extraer redefinido en las subclases (uso de super)

CuentaBancaria>> extraer: unMonto
self decrementarSaldo:unMonto

CajaAhorro>> extraer: unMonto
self hayFondos: unMonto
if True:[self incrementarExtr.
super extraer:unMonto]

CuentaCorriente>> extraer: unMonto
self hayFondos: unMonto
if True:[super extraer:unMonto]

CajaAhorro>> hayFondos: unMonto
^saldo >= unMonto

CuentaCorriente>> hayFondos: unMonto
^saldo >= unMonto + descubierto



Ejemplo: Implementación de #extraer:

Opción 3: podemos abstraer en la superclase “la forma o esqueleto” del método e implementar el comportamiento específico en las subclases (uso de *self*)

Teníamos....

```
CuentaBancaria>> extraer: unMonto  
    self decrementarSaldo:unMonto
```

```
CajaAhorro>> extraer: unMonto  
    self hayFondos: unMonto  
    ifTrue:[ self incrementarExtr.  
             super extraer:unMonto]
```

```
CuentaCorriente>> extraer: unMonto  
    self hayFondos: unMonto  
    ifTrue:[ super extraer:unMonto]
```

Ejecución
de la
extracción

```
CajaAhorro>> hayFondos: unMonto  
    ^saldo >= unMonto
```

```
CuentaCorriente>> hayFondos: unMonto  
    ^saldo >= unMonto + descubierto
```



Definimos el # ***ejecutarExtraccion:***

CuentaBancaria>> extraer: unMonto
self decrementarSaldo:unMonto

CajaAhorro>> extraer: unMonto
self hayFondos: unMonto
if True:[self ***ejecutarExtraccion:*** unMonto]

CuentaCorriente>> extraer: unMonto
self hayFondos: unMonto
if True:[self ***ejecutarExtraccion:*** unMonto]

CajaAhorro>> *ejecutarExtraccion:*** unMonto**

CuentaCorriente>> *ejecutarExtraccion:*** unMonto**

Ambos
metodos
son
iguales



Implementamos el #extraer: en la superclase

CuentaBancaria>> extraer: unMonto

self hayFondos: unMonto

if True: [self *ejecutarExtraccion*: unMonto]

CajaAhorro>> *ejecutarExtraccion*: unMonto

self incrementarExtr.

self decrementarSaldo: unMonto

CuentaCorriente>> *ejecutarExtraccion*: unMonto

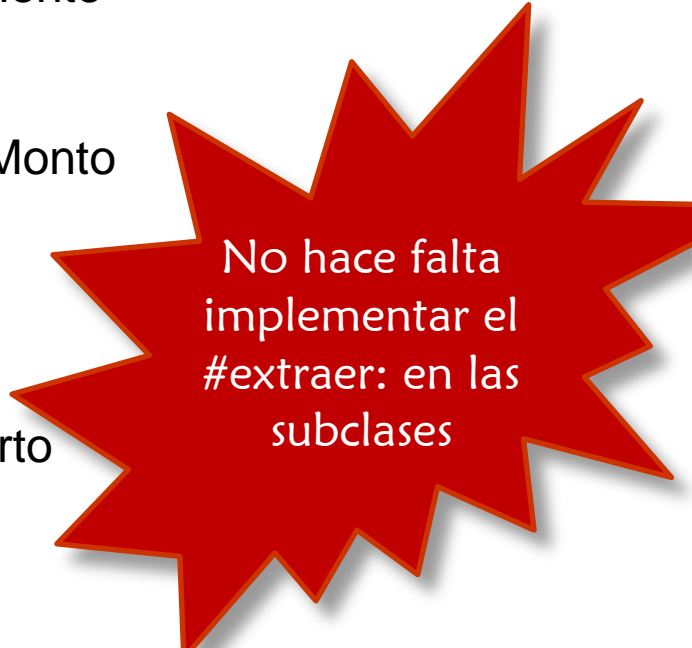
self decrementarSaldo: unMonto

CajaAhorro>> hayFondos: unMonto

^saldo >= unMonto

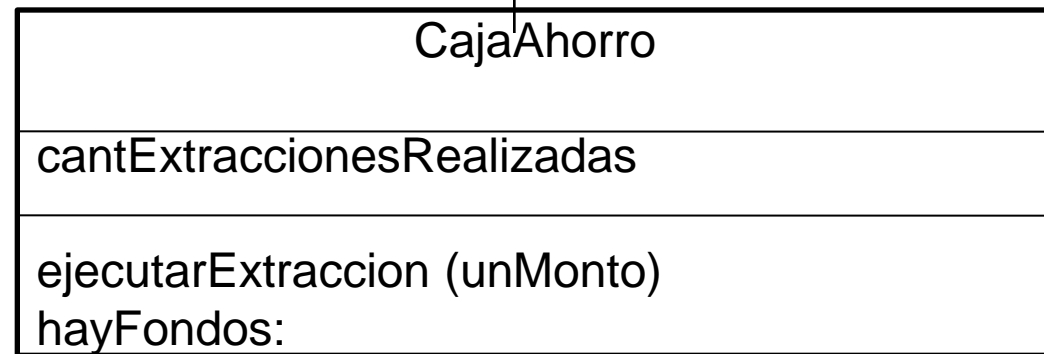
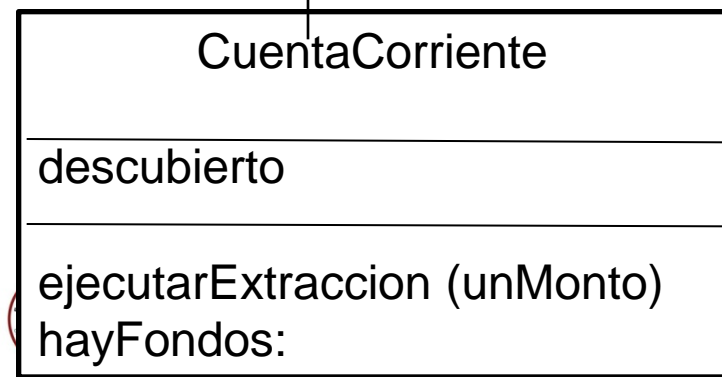
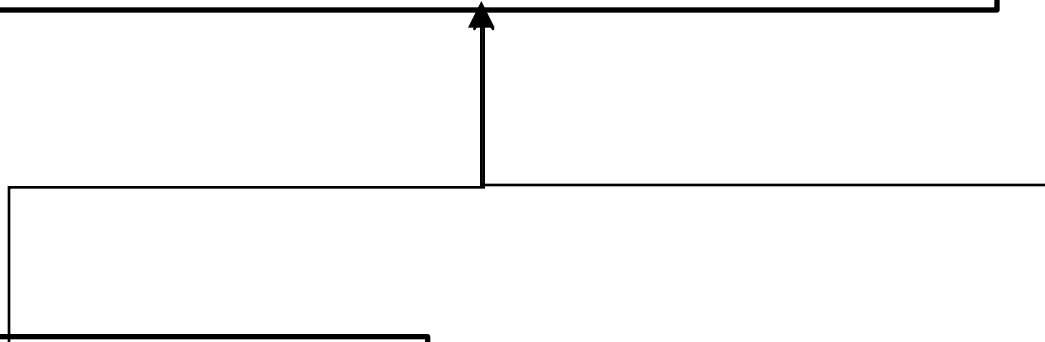
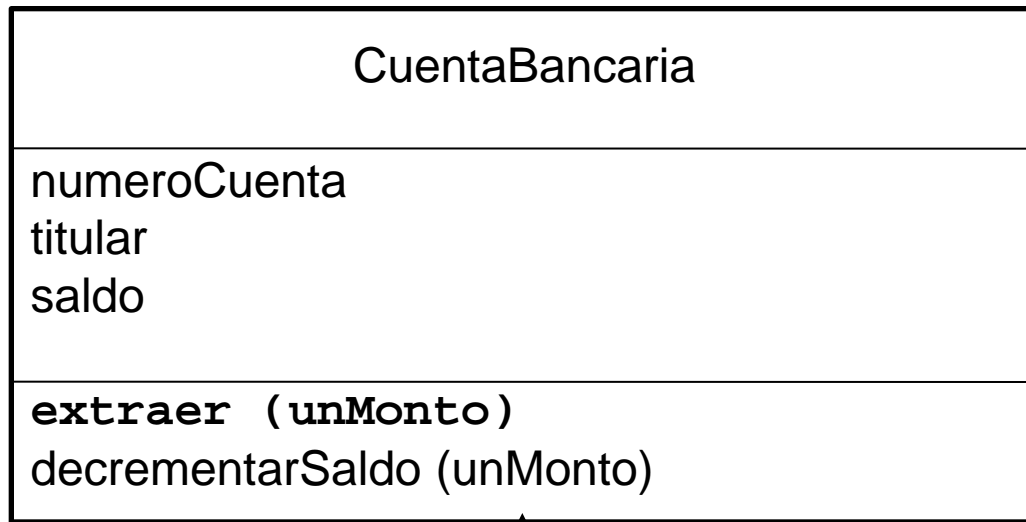
CuentaCorriente>> hayFondos: unMonto

^saldo >= unMonto + descubierto



No hace falta
implementar el
#extraer: en las
subclases

Opción 2: #extraer redefinido en las subclases (uso de super)



Otros ejemplos de la Opción 3: #between: and:

Magnitude|Kernel

- ExactFloatPrintPolicy
- FloatPrintPolicy
- InexactFloatPrintPolicy
- Magnitude**
- Number
- Float
- BoxedFloat64

-- all --
comparing
hash
testing

between: and:
compareWith:ifLesser:ifEqual:ifGreater:
hash
max

between: min and: max
"Answer whether the receiver is less than or equal to the argument, max,
and greater than or equal to the argument, min."

`^self >= min and: [self <= max]`

Magnitude|Kernel

- Number
- Float
- BoxedFloat64
- SmallFloat64
- Fraction**
- ScaledDecimal
- Integer

-- all --
arithmetic
comparing
converting
mathematical fi
printing
private
self evaluating
testing

<= aNumber
aNumber isFraction ifTrue:
[^a numerator * aNumber denominator <= (aNumber nume
denominator)].
^a aNumber adaptToFraction: self andCompare: #<=

Magnitude|Kernel

- Number
- Float
- BoxedFloat64
- SmallFloat64
- Fraction
- ScaledDecimal
- Integer**

-- all --
accessing
arithmetic
benchmarks
bit manipulation
comparing
converting
converting-arra
enumerating

<= aNumber
aNumber isInteger ifTrue:
[self negative == aNumber negative
ifTrue: [self negative
ifTrue: [^a (self digitCompare: aNumber) >= 0]
ifFalse: [^a (self digitCompare: aNumber) <= 0]]
ifFalse: [^a self negative]].
^a aNumber adaptToInteger: self andCompare: #<=

Otros ejemplos de la Opción 3: #select:

The screenshot shows the Ruby IDE interface with the `Collection>>#select:` window open. The left sidebar shows the project structure with `Collections-Abstract` selected. The main window displays the documentation for the `#select` method.

Collection>>#select:

History Navigator

- comparing
- converting
- copying
- enumerating
- filter streaming
- math functions

- reject:thenCollect:
- reject:thenDo:
- select:**
- select:thenCollect:
- select:thenDo:
- union:

select: aBlock

"Evaluate aBlock with each of the receiver's elements as the argument. Collect into a new collection like the receiver, only those elements for which aBlock evaluates to true. Answer the new collection."

```
| newCollection |
newCollection := self copyEmpty.
self do: [ :each |
  (aBlock value: each)
  ifTrue: [ newCollection add: each ] ].
^newCollection
```

The screenshot shows the Ruby IDE interface with the `Collection>>#do:` window open. The left sidebar shows the project structure with `Collections-Abstract` selected. The main window displays the documentation for the `#do` method.

Collection>>#do:

History Navigator

- comparing
- converting
- copying
- enumerating
- filter streaming
- math functions

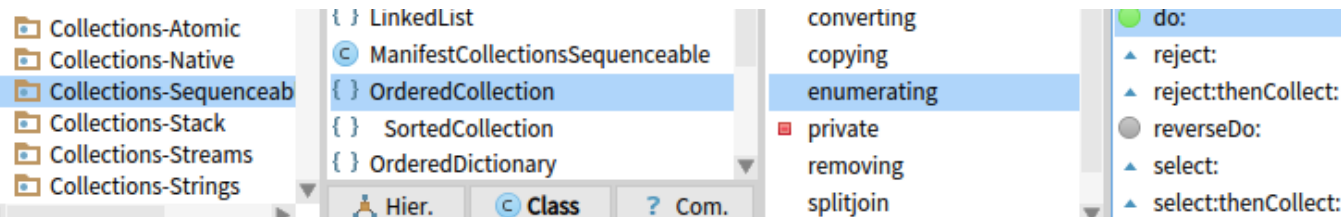
- difference:
- do:**
- do:separatedBy:
- do:without:
- findFirstInByteString:startingAt:
- flatMap:

do: aBlock

"Evaluate aBlock with each of the receiver's elements as the argument."

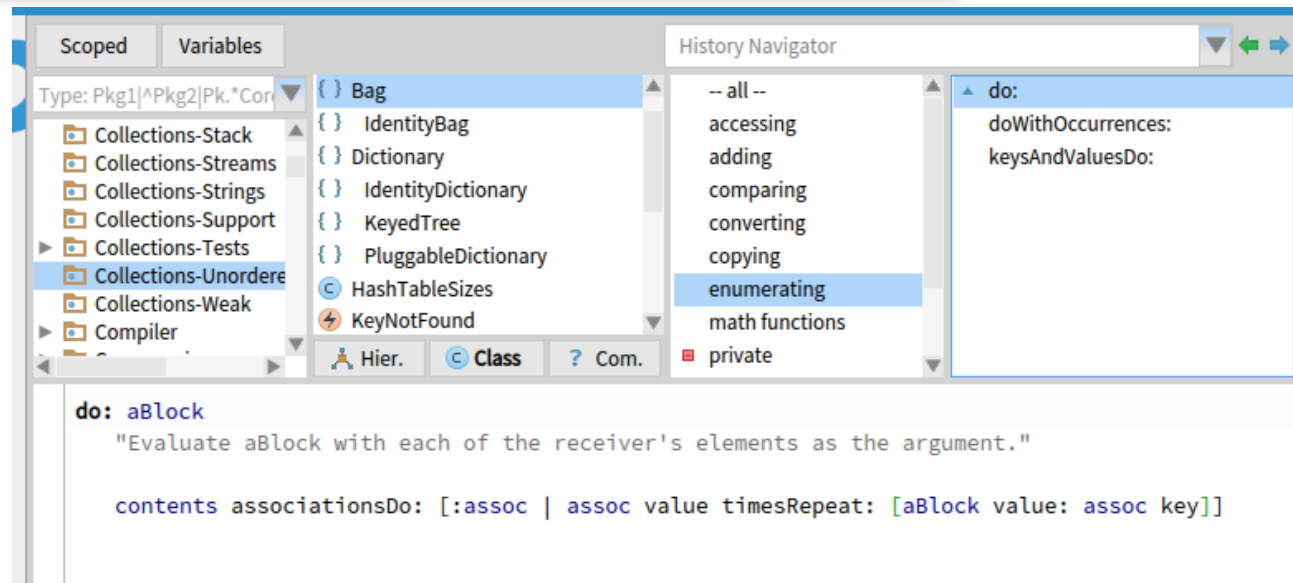
```
self subclassResponsibility
```

Otros ejemplos de la Opción 3: #select:



```
do: aBlock
    "Override the superclass for performance reasons."

    firstIndex to: lastIndex do: [ :index |
        aBlock value: (array at: index) ]
```



```
do: aBlock
    "Evaluate aBlock with each of the receiver's elements as the argument."

    contents associationsDo: [:assoc | assoc value timesRepeat: [aBlock value: assoc key]]
```

