Refactorings De Colecciones En Smalltalk

Anastopulos Matias

2 de septiembre de 2015

1. Motivación

Lo que trataremos en este articulo son ciertos problemas recurrentes que se presentan a la hora de trabajar con colecciones de objetos. Muchos programadores, que vienen de otros lenguajes, encuentran soluciones a estos problemas que utilizaban antes de empezar con Smalltalk. Sin embargo, estas soluciones, a la manera de otros lenguajes como Java y C++, en general llevan a código mas complejo, por lo tanto, el motivo de esta exposición es presentar las soluciones a estos problemas, de la manera en que se resuelven en Smalltalk.

Smalltalk posee una serie de métodos llamados métodos iteradores que simplifican mucho el trabajo con colecciones. Estos se basan en bloques de código, que no son otra cosa que objetos de la clase BlockClosure. Proveen una forma muy elegante y compacta de resolver estos problemas, que de otra forma requerirían código más largo y complejo, y por lo tanto, requerirían más trabajo para el programador a la hora de trabajar con el mismo.

La siguiente exposición es una serie de refactorizaciones. Dichas refactorizaciones se basan en llevar código deficiente, y que no esta hecho a la manera de Smalltalk, a una forma mas elegante. Las refactorizaciones se exponen en una forma parecida a la del libro de Fowler, Refactoring. Cada refactorización posee un nombre, para poder identificarla, un problema a resolver, la solución, algunos ejemplos de la misma y finalmente algunas notas de explicación.

2. Catálogo

2.1. Reemplace Índice Por Do

2.1.1. Problema

Está recorriendo los elementos de una colección usando un índice.

2.1.2. Solución

Utilice el método 'do:' en su lugar.

2.1.3. Ejemplo

Antes de la refactorización:

```
nombres := OrderedCollection new.
nombres add: 'Matias'.
nombres add: 'Nicolas'.
nombres add: 'Pablo'.

indice := 1.

[ indice <= 3 ] whileTrue: [
        Transcript show: ( nombres at: indice ) ; cr.
        indice := indice + 1.
]
```

Despues de la refactorización:

```
nombres := OrderedCollection new.
nombres add: 'Matias'.
nombres add: 'Nicolas'.
nombres add: 'Pablo'.

nombres do: [ :nombre | Transcript show: nombre ; cr ]
```

2.1.4. Explicación

La forma de recorrer una colección en Smalltalk no es mediante índices, es mediante los métodos iteradores. La mayoría de los refactorings que vamos a dar los utilizan. Los métodos iteradores se basan en un bloque que es enviado como parámetro al método. En este caso la sintaxis es:

coleccion do: [**:elemento** | **procesamiento del elemento**]

coleccion, es la colección que contiene los elementos sobre los que queremos iterar. Lo que sigue es el nombre del método 'do:' que recibe un bloque como parámetro. El bloque se va a llamar para cada uno de los elementos de la colección. **:elemento** es la variable temporal sobre la que se guarda el elemento actual en la iteración durante la ejecución del bloque. Para acceder al elemento actual en el bloque de procesamiento, utilizamos el nombre **:elemento**.

2.1.5. Variante Separated By

Hay una variante del 'do:', 'separatedBy:', que llama a un segundo bloque entre cada elemento. Por ejemplo si deseamos imprimir una serie de nombres separados por un guión hacemos lo siguiente:

```
nombres := OrderedCollection new.
nombres add: 'Matias'.
nombres add: 'Nicolas'.
nombres add: 'Pablo'.
```

```
5
6  nombres do: [ :nombre | Transcript show: nombre ]
7          separatedBy: [ Transcript show: ' - ' ]
```

2.2. Reemplace Do Por Select/Reject

2.2.1. Problema

Está seleccionando algunos elementos en particular de una colección a través de un 'do:' y una colección auxiliar.

2.2.2. Solución

Reemplace el 'do:' por un 'select:'

2.2.3. Ejemplo

Antes de la refactorización:

```
1  lapices := OrderedCollection new.
2
3  lapiz1 := Lapiz new.
4  lapiz1 color: 'rojo'.
5  lapices add: lapiz1.
6
7  lapiz2 := Lapiz new.
8  lapiz2 color: 'amarillo'.
9  lapices add: lapiz2.
10
11 lapiz3 := Lapiz new.
12 lapiz3 color: 'rojo'.
13 lapices add: lapiz3.
14
15 lapicesBuscados := OrderedCollection new.
16 lapices do: [ :lapiz |
17         ( lapiz color = 'rojo' ) ifTrue: [
18                 lapicesBuscados add: lapiz
19         ]
20 ].
```

Despues de la refactorización:

```
1  lapices := OrderedCollection new.
2
3  lapiz1 := Lapiz new.
4  lapiz1 color: 'rojo'.
5  lapices add: lapiz1.
6
7  lapiz2 := Lapiz new.
8  lapiz2 color: 'amarillo'.
```

```
lapices add: lapiz2.

lapiz3 := Lapiz new.
lapiz3 color: 'rojo'.
lapices add: lapiz3.

lapicesBuscados := lapices select: [ :lapiz | lapiz color = 'rojo' ].
```

2.2.4. Explicación

Si bien el utilizar un 'do:' y una colección auxiliar para recolectar una serie de elementos con una propiedad especial funciona, Smalltalk ofrece una solución mucho más elegante. La idea es utilizar el método iterador 'select:', que posee la siguiente sintaxis:

seleccionados := **colección** do: [:**elemento** | **test**]

Al igual que el 'do:', select invoca el bloque para cada elemento de la colección, pasando el elemento actual al bloque a través de la variable **elemento**. **test** es una expresión que debe devolver true o false. Si la expresión es verdadera, el elemento actual se agrega a la colección **seleccionados**.

El método 'reject:' funciona a la inversa del select. Se queda solo con los elementos en los que la condición es falsa.

2.3. Reemplace Do Por Collect

2.3.1. Problema

Está recolectando elementos relacionados con los elementos de la colección, pero lo está haciendo a través de un do.

2.3.2. Solución

Reemplace el 'do:' por un 'collect.'

2.3.3. Ejemplo

Antes de la refactorización:

```
personas := OrderedCollection new.

personas add: ( Persona newConNombre: 'Matias' yMail: 'Matias@gmail.com' ).
personas add: ( Persona newConNombre: 'Nicolas' yMail: 'Nicolas@gmail.com' ).

mails := OrderedCollection new.

personas do: [ :persona | mails add: persona mail ].
```

Despues de la refactorización:

```
personas := OrderedCollection new.

personas add: ( Persona newConNombre: 'Matias' yMail: 'Matias@gmail.com' ).
personas add: ( Persona newConNombre: 'Nicolas' yMail: 'Nicolas@gmail.com' ).

mails := personas collect: [ :persona | persona mail ].
```

2.3.4. Explicación

La idea básica es, dada una colección, recolectar una serie de elementos relacionados con los de la misma. Por ejemplo, en el caso anterior, los mails de una serie de personas. Otro ejemplo podría ser, dado una serie de números, recolectar el cuadrado de los mismos. Es como si recolectara las imágenes de los elementos de una colección a través de una función definida mediante el bloque. Es por eso que en otros lenguajes, como en Ruby, el collect se llama map (Mapeo). La sintaxis es:

conjunto imagen := **colección** collect: [:**elemento** | **mapeo**]

colección se mapea en el **conjunto imagen**, a través del mapeo definido en el bloque. El bloque recibe los elementos, uno por uno, a traves de la variable **elemento**, y el mapeo debe computar el elemento relacionado con el recibido, que va a almacenarse en el **conjunto imagen**.

2.4. Reemplace Do Por Inject Into

2.4.1. Problema

Está recorriendo con 'do:' una serie de elementos de una colección y acumulando un cierto valor a partir de ellos, como por ejemplo la suma de los mismos.

2.4.2. Solución

Reemplace el 'do:' por un 'inject:into:'

2.4.3. Ejemplo

Antes de la refactorización:

```
valores := OrderedCollection withAll: #( 3 5 9 3 ).

acumulador := 0.
valores do: [ :valor | acumulador := acumulador + valor ].
```

Despues de la refactorización:

```
valores := OrderedCollection withAll: #( 3 5 9 3 ).

total := valores inject: 0 into: [ :acumulador :valor | acumulador + valor ].
```

2.4.4. Explicación

El ejemplo anterior computa la suma de todos los elementos de una colección. Smalltalk ofrece, para realizar acumulaciones de este estilo, un método iterador llamado 'inject:into:'. La sintaxis es:

total := **colección** inject: **valor inicial**
into: [:**acumulador** :**elemento** | **computo del valor siguiente**]

El bloque recibe el valor acumulado de los ciclos anteriores y el elemento actual a través de **acumulador** y **elemento**. Se computa el valor siguiente en el bloque, y una vez pasados todos los elementos se guarda el valor en **total**. **valor inicial** contiene el valor que se pasa al **acumulador** en el primer ciclo.

2.5. Reemplace Do por Detect

2.5.1. Problema

Está buscando un elemento con una característica en particular en una colección con un 'do:'.

2.5.2. Solución

Reemplace el 'do:' por un 'detect:'.

2.5.3. Ejemplo

Antes de la refactorización:

```
1 buscarVolcal: unaCadena
2         unaCadena do: [ :caracter | ( caracter isVowel ) ifTrue: [ ^caracter ] ].
3 ... En otra parte se llama al metodo ...
4 caracterBuscado := self buscarVolcal: 'hola'.
```

Despues de la refactorización:

```
1 caracterBuscado := 'hola' detect: [ :caracter | caracter isVowel ].
```

2.5.4. Explicación

El método 'detect:' se usa para encontrar el primer elemento en una colección que cumpla con un determinado requerimiento. La sintaxis es:

buscado := **colección** detect: [:**elemento** | **test**]

El requerimiento es especificado a través de un **test** en el bloque que debe devolver verdadero o falso. El bloque se invoca para cada elemento de la **colección** y el elemento actual de la iteración es pasado a traves de la variable **elemento**. Si el **test** se evalúa como verdadero, ese **elemento** es devuelto por el método.

En caso de que el **test** no sea verdadero para ninguno de los elementos existe una variante 'detect:ifNone:' con un bloque adicional que contiene el objeto que es devuelto por el método en caso de que ninguno de los elementos pase el **test**.

buscado := **colección** detect: [:**elemento** | **test**]
ifNone: [**objeto por defecto**]

2.6. Reemplace Do por Count

2.6.1. Problema

Está contando la cantidad de elementos que cumplen con una condición con un 'do:' y una variable contadora.

2.6.2. Solución

Reemplace el 'do:' por un 'count:'.

2.6.3. Ejemplo

Antes de la refactorización:

```
lapices := OrderedCollection new.

lapiz1 := Lapiz new.
lapiz1 color: 'rojo'.
lapices add: lapiz1.

lapiz2 := Lapiz new.
lapiz2 color: 'amarillo'.
lapices add: lapiz2.

lapiz3 := Lapiz new.
lapiz3 color: 'rojo'.
lapices add: lapiz3.

cantidad := 0.
lapices do: [ :lapiz | ( lapiz color = 'rojo' ) ifTrue: [
        cantidad := cantidad + 1
        ]
].
```

Despues de la refactorización:

```
lapices := OrderedCollection new.

lapiz1 := Lapiz new.
lapiz1 color: 'rojo'.
lapices add: lapiz1.

lapiz2 := Lapiz new.
lapiz2 color: 'amarillo'.
lapices add: lapiz2.

```

```
lapiz3 := Lapiz new.
lapiz3 color: 'rojo'.
lapices add: lapiz3.

cantidad := lapices count: [ :lapiz | lapiz color = 'rojo' ].
```

2.6.4. Explicación

Smalltalk posee un método iterador 'count:' que provee una forma más elegante de contar los elementos que cumplen con una condición. La sintaxis es:

total := **colección** count: [:**elemento** | **test**]

El bloque se va a invocar para cada uno de los elementos de la colección, pasando el elemento actual de la iteración a través de la variable **elemento**. Si **test** se evalua como true para ese elemento, la cuenta de elementos que cumplen con el test se incrementa en 1. Al final del cómputo, 'count:' devuelve el total de la elementos que pasaron el **test**. Esta es una forma mas compacta de contar elementos ya que no necesitamos ninguna variable adicional para llevar la cuenta de elementos.

2.7. Reemplace Do por Includes

2.7.1. Problema

Está recorriendo una colección con un 'do:' para saber si incluye un elemento.

2.7.2. Solución

Reemplace el 'do:' por un 'includes:'.

2.7.3. Ejemplo

Antes de la refactorización:

```
nombres := OrderedCollection new.
nombres add: 'Matias'.
nombres add: 'Nicolas'.
nombres add: 'Pablo'.

encontrado := false.
nombres do: [ :nombre | ( nombre = 'Matias' ) ifTrue: [ encontrado := true ] ].
```

Despues de la refactorización:

```
nombres := OrderedCollection new.
nombres add: 'Matias'.
nombres add: 'Nicolas'.
nombres add: 'Pablo'.

encontrado := nombres includes: 'Matias'.
```

2.7.4. Explicación

Una forma mucho más simple de encontrar los elementos es usando la interfaz que las colecciones nos proveen, en este caso a través del método 'includes:'. La sintaxis es:

encontrado := **colección** includes: **elemento**

Básicamente devuelve true o false si el **elemento** está o no en la **colección**.

2.8. Reemplace Do por AnySatisfy

2.8.1. Problema

Está recorriendo con un 'do:' una colección para saber si algún elemento cumple una condición.

2.8.2. Solución

Reemplace el 'do:' por un 'anySatisfy:'.

2.8.3. Ejemplo

Antes de la refactorización:

```
1 valores := OrderedCollection withAll: #( 3 5 6 8 ).
2
3 flag := false.
4 valores do: [ :valor | ( valor > 7 ) ifTrue: [ flag := true ] ].
```

Despues de la refactorización:

```
1 valores := OrderedCollection withAll: #( 3 5 6 8 ).
2
3 hayUnoMayorASiete := valores anySatisfy: [ :valor | valor > 7 ].
```

2.8.4. Explicación

Smalltalk provee el método 'anySatisfy:' para verificar si algún elemento cumple con una condición. Esta es una solución mas elegante que usar un 'do:' y un flag. La sintaxis es:

satisface := **colección** anySatisfy: [:**elemento** | **test**]

El bloque se invoca para cada uno de los elementos de la **colección**, pasando el elemento actual de la iteración a través de la variable **elemento**. Si para algun **elemento** el **test** se evalua como true, el método devuelve true, mientras que si es falso para todos los elementos el método devuelve false.

3. Bibliografía

[1] Martin Fowler, Kent Beck, Shane Harvie, Jay Fields. Refactoring Ruby Edition. Pearson Education. United States. 2009.

[2] Andrew P. Black, Oscar Nierstrasz, Stéphane Ducasse, Damien Pollet. Pharo By Example. Square Bracket Associates. Switzerland. 2013.