Práctica 2 a: Eliminar Código Repetido

Tomás Felipe Melli

April 10, 2025

Índice

1 Introducción

Muchas veces vemos un programa donde es evidente que cachos de código, lógicamente, repiten los mismo. Por ejemplo, si tenemos dos ciclos, podríamos abstraernos a la idea de ciclo y sólo pasarle sobre qué lista por ejemplo, hacer qué cosa. La idea de esta práctica es generar el hábito de no repetir este tipo de estructuras que ensucian el código.

2 Customer Book Tests

2.1 Tests 1 y 2

Para los dos primeros test vemos que coincide :

```
test01AddingCustomerShouldNotTakeMoreThan50Milliseconds
    | customerBook millisecondsBeforeRunning millisecondsAfterRunning ||
    customerBook := CustomerBook new.

    millisecondsBeforeRunning := Time millisecondClockValue * millisecond.
    customerBook addCustomerNamed: 'John Lennon'.
    millisecondsAfterRunning := Time millisecondClockValue * millisecond.

    self assert: (millisecondsAfterRunning-millisecondsBeforeRunning) < (50 * millisecond)
```

```
test02RemovingCustomerShouldNotTakeMoreThan100Milliseconds
    | customerBook millisecondsBeforeRunning millisecondsAfterRunning paulMcCartney |
    customerBook := CustomerBook new.
    paulMcCartney := 'Paul McCartney'.

    customerBook addCustomerNamed: paulMcCartney.

    millisecondsBeforeRunning := Time millisecondClockValue * millisecond.
    customerBook removeCustomerNamed: paulMcCartney.
    millisecondsAfterRunning := Time millisecondClockValue * millisecond.
```

El hecho de declarar colaboradores temporales que capturar la diferencia de los tiempos de ejecución, de algo y luego constatan si cumple cierta diferencia de milisegundos. Eso se puede abstraer con un método como **mustDo: inMilliseconds:** que :

```
mustDo: aBlock inMilliseconds: milliseconds

    | millisecondsBeforeRunning millisecondsAfterRunning |

    millisecondsBeforeRunning := Time millisecondClockValue * millisecond.
    aBlock value.
    millisecondsAfterRunning := Time millisecondClockValue * millisecond.

    self assert: (millisecondsAfterRunning-millisecondsBeforeRunning) < (milliseconds * millisecond)
```

Como consecuencia nos queda un código más limpio.

```
test01AddingCustomerShouldNotTakeMoreThan50Milliseconds

    | customerBook |

    customerBook := CustomerBook new.

    self mustDo: [customerBook addCustomerNamed: 'John Lennon'] inMilliseconds: 50.
```

```
test02RemovingCustomerShouldNotTakeMoreThan100Milliseconds

    | customerBook |

    customerBook := CustomerBook new.

    self mustDo: [customerBook addCustomerNamed: 'Paul McCartney'.] inMilliseconds: 100
```

** La correción incluye realizar una abstracción un poco más general donde no le pasamos el número de *milisegundos*, sino un *time limit*. Es decir, un número multiplicado por su unidad. Además, dentro del método podemos abstraer la idea de calcular el *runtime*.

```
mustDo: aBlock inLessThan: aTimeLimit

    self assert: (self measureRuntimeBlock: aBlock) < aTimeLimit
```

```
measureRuntimeBlock: aBlockToMeasure

    | millisecondsBeforeRunning millisecondsAfterRunning |

    millisecondsBeforeRunning := Time millisecondClockValue * millisecond.
    aBlockToMeasure value.
    millisecondsAfterRunning := Time millisecondClockValue * millisecond.

    ^(millisecondsAfterRunning - millisecondsBeforeRunning).
```

2.2 Tests 3 y 4

```
test03CanNotAddACustomerWithEmptyName

    | customerBook |

    customerBook := CustomerBook new.

    [ customerBook addCustomerNamed: ''.
    self fail ]
        on: Error
        do: [ :anError |
            self assert: anError messageText = CustomerBook
customerCanNotBeEmptyErrorMessage.
            self assert: customerBook isEmpty ]
```

```
test04CanNotRemoveAnInvalidCustomer

    | customerBook johnLennon |

    customerBook := CustomerBook new.
    johnLennon := 'John Lennon'.
    customerBook addCustomerNamed: johnLennon.

    [ customerBook removeCustomerNamed: 'Paul McCartney'.
    self fail ]
        on: NotFound
        do: [ :anError |
            self assert: customerBook numberOfCustomers = 1.
            self assert: (customerBook includesCustomerNamed:
johnLennon) ]
```

A partir de ellos, vemos que la idea es : realizar una acción, fallar, capturar el error y hacer algo. Esto podemos abstraerlo y construir un método como **ifWeTryTo: withAGivenError: then:** que :

```
ifWeTryTo: aBlockThatFails withAGivenError: anError then:
doSomething

    [aBlockThatFails value. self fail] on: anError  do: doSomething
```

```
test03CanNotAddACustomerWithEmptyName

    | customerBook |

    customerBook := CustomerBook new.

    self ifWeTryTo: [ customerBook addCustomerNamed: ''.]
    withAGivenError: Error
    then: [ :anError |
            self assert: anError messageText = CustomerBook
customerCanNotBeEmptyErrorMessage.
            self assert: customerBook isEmpty ].
```

```
test04CanNotRemoveAnInvalidCustomer

    | customerBook |

    customerBook := CustomerBook new.

    self ifWeTryTo:
    [ customerBook addCustomerNamed: 'John Lennon'.
customerBook removeCustomerNamed: 'Paul McCartney'. ]
    withAGivenError: NotFound
    then: [ :anError |
            self assert: customerBook numberOfCustomers = 1.
            self assert: (customerBook includesCustomerNamed:
'John Lennon') ].
```

** en la corrección, los nombres que puse podrían ser más declarativos. Lo que está bueno de esto es prestar atención al **self fail** que está dentro del bloque. La idea de estos también era mostrar que *renombrar no es sacar código repetido*.

2.3 Tests 5 y 6

```
test05SuspendingACustomerShouldNotRemoveItFromCustom
erBook
    | customerBook paulMcCartney|
    customerBook := CustomerBook new.
    paulMcCartney := 'Paul McCartney'.

    customerBook addCustomerNamed: paulMcCartney.
    customerBook suspendCustomerNamed: paulMcCartney.

    self assert: 0 equals: customerBook numberOfActiveCustomers.
    self assert: 1 equals: customerBook
numberOfSuspendedCustomers.
    self assert: 1 equals: customerBook numberOfCustomers.
    self assert: (customerBook includesCustomerNamed:
paulMcCartney).
```

```
test06RemovingASuspendedCustomerShouldRemoveItFromC
ustomerBook
    | customerBook paulMcCartney|
    customerBook := CustomerBook new.
    paulMcCartney := 'Paul McCartney'.
    customerBook addCustomerNamed: paulMcCartney.
    customerBook suspendCustomerNamed: paulMcCartney.
    customerBook removeCustomerNamed: paulMcCartney.

    self assert: 0 equals: customerBook numberOfActiveCustomers.
    self assert: 0 equals: customerBook
numberOfSuspendedCustomers.
    self assert: 0 equals: customerBook numberOfCustomers.
    self deny: (customerBook includesCustomerNamed:
paulMcCartney).
```

En estos caso vemos que en ambos se agrega y suspende un customer. Eso se puede abstraer como así también todos los asser en métodos como **addAndSuspend: inBook:**, **do: assertThat** y **selfAssertNumber: inBook:**

```
addAndSuspend: aName inBook: aBook

    aBook addCustomerNamed: aName .
    aBook suspendCustomerNamed: aName .
```

```
do: aBlock assertThat: anAssertion

    [aBlock value.].[anAssertion value]
```

```
selfAssertNumber: aNumber inBook: aBook

    self assert: 0 equals: aBook numberOfActiveCustomers.
    self assert: aNumber equals: aBook
numberOfSuspendedCustomers.
    self assert: aNumber equals: aBook numberOfCustomers.
```

De manera de dejar dos tests más limpios :

```
test05SuspendingACustomerShouldNotRemoveItFromCustom
erBook

    | customerBook |
    customerBook := CustomerBook new.

    self addAndSuspend: 'Paul McCartney' inBook: customerBook.
    self selfAssertNumber: 1 inBook: customerBook
```

```
test06RemovingASuspendedCustomerShouldRemoveItFrom
CustomerBook

    | customerBook |

    customerBook := CustomerBook new.

    self addAndSuspend: 'Paul McCartney' inBook: customerBook.
    customerBook removeCustomerNamed: 'Paul McCartney'.

    self selfAssertNumber: 0 inBook: customerBook.
    self deny: (customerBook includesCustomerNamed: 'Paul
McCartney').
```

2.4 Tests 7 y 8

```
test07CanNotSuspendAnInvalidCustomer

    | customerBook johnLennon |

    customerBook := CustomerBook new.
    johnLennon := 'John Lennon'.
    customerBook addCustomerNamed: johnLennon.

    [ customerBook suspendCustomerNamed: 'George Harrison'.
    self fail ]
        on: CantSuspend
        do: [ :anError |
            self assert: customerBook numberOfCustomers = 1.
            self assert: (customerBook includesCustomerNamed:
johnLennon) ]
```

```
test08CanNotSuspendAnAlreadySuspendedCustomer

    | customerBook johnLennon |
    customerBook := CustomerBook new.
    johnLennon := 'John Lennon'.
    customerBook addCustomerNamed: johnLennon.
    customerBook suspendCustomerNamed: johnLennon.

    [ customerBook suspendCustomerNamed: johnLennon.
    self fail ]
        on: CantSuspend
        do: [ :anError |
            self assert: customerBook numberOfCustomers = 1.
            self assert: (customerBook includesCustomerNamed:
johnLennon) ]
```

Se repite el catch del cantSuspend. Por eso planteamos el método **catchCantSuspend: inBook:**

```
catchCantSuspend: aName inBook: aBook

    [ aBook suspendCustomerNamed: aName .
    self fail ]
        on: CantSuspend
        do: [ :anError |
            self assert: aBook numberOfCustomers = 1.
            self assert: (aBook includesCustomerNamed: 'John
Lennon') ]
```

Con ello dejamos un código mucho más limpio :

```
test07CanNotSuspendAnInvalidCustomer

    | customerBook |

    customerBook := CustomerBook new.
    customerBook addCustomerNamed: 'John Lennon'.
    self catchCantSuspend: 'George Harrison' inBook: customerBook.
```

```
test08CanNotSuspendAnAlreadySuspendedCustomer

    | customerBook |

    customerBook := CustomerBook new.
    self addAndSuspend: 'John Lennon' inBook: customerBook.
    self catchCantSuspend: 'John Lennon' inBook: customerBook.
```

3 Customer Book

Tenemos dos ciclos que en dos listas diferentes que podríamos abstraer. En este caso, lo hacemos sobre el mismo método.

```
removeCustomerNamed: aName
    1 to: active size do:
    [ :index |
        aName = (active at: index)
            ifTrue: [
                active removeAt: index.
                ^ aName
            ]
    ].
    1 to: suspended size do:
    [ :index |
        aName = (suspended at: index)
            ifTrue: [
                suspended removeAt: index.
                ^ aName
            ]
    ].
    ^ NotFound signal.
```

Antes yo había hecho :

```
iterateInList: aList  andRemoveName: aName

    1 to: aList size
    do: [ :index |
        aName = (aList at: index)
            ifTrue: [
                aList removeAt: index.
                ^ aName
            ]
    ].
```

```
brows sendrs implmntrs versins inhertnc hirrchy nstVrs clssVrs shw

removeCustomerNamed: aName

    self iterateInList: active andRemoveName: aName.
    self iterateInList: suspended andRemoveName: aName.

    ^ NotFound signal.
```

Pero el problema de esto son los **return** dentro de cada mensaje. Ya que, cuando llega el ret, el contexto de ejecución del self actual no sale, sino que vuelve a sí mismo. Cómo ? Evalúa en *active si está, lo encuentra y hace ret.* Evalúa en suspended, lo encuentra y hace ret. Retorna *NotFound signal.* Ese problema se resuelve creando un método que tiene los dos bloques que queremos chequear de forma de hacer algo del estilo *concatenado.*

```
removeCustomerNamed: aName

    self removeFrom: active theElement: aName  ifAbsent: [
        self removeFrom: suspended theElement: aName ifAbsent: [^NotFound signal]
        ]
```

```
removeFrom: aCollection theElement: anElementToRemove ifAbsent: aBlock

    1 to: aCollection size do:
    [ :index |
        anElementToRemove = (aCollection at: index)
            ifTrue: [
                aCollection removeAt: index.
                ^ anElementToRemove
            ]
    ].

    "tenemos que encadenar"
    ^ aBlock value.
```