Práctica 2: Eliminar Código Repetido

Tomás Felipe Melli

April 7, 2025

Índice

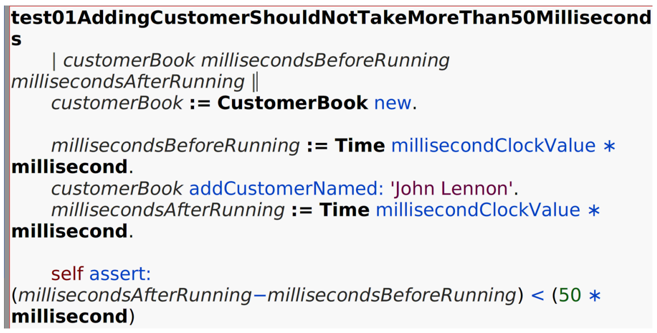
1 Introducción

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.

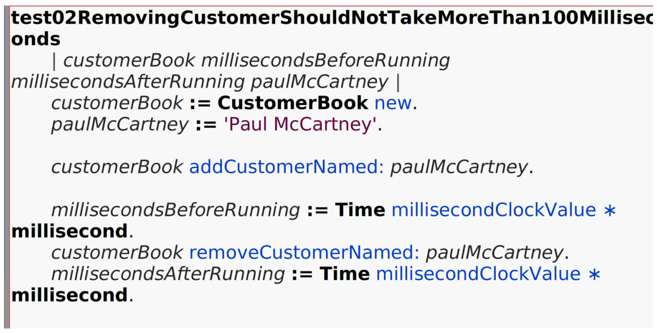
2 Customer Book Tests

2.1 Tests 1 y 2

Para los dos primeros test vemos que coincide :

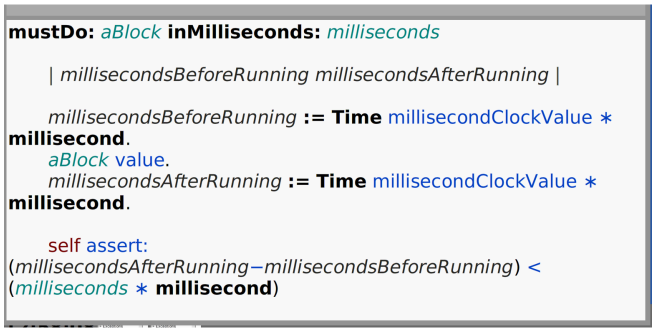


```
test01AddingCustomerShouldNotTakeMoreThan50Millisecond
s
      | customerBook millisecondsBeforeRunning
millisecondsAfterRunning ||
      customerBook := CustomerBook new.

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

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



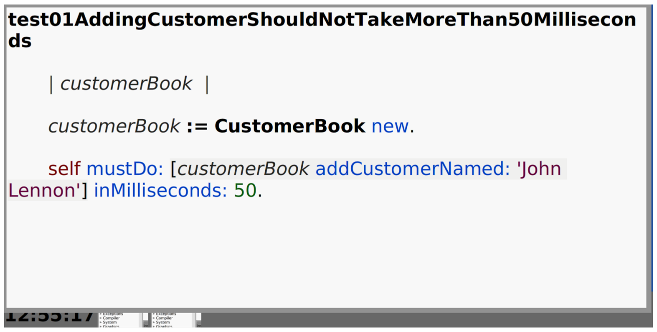
```
test02RemovingCustomerShouldNotTakeMoreThan100Millisec
onds
      | customerBook millisecondsBeforeRunning
millisecondsAfterRunning paulMcCartney |
      customerBook := CustomerBook new.
      paulMcCartney := 'Paul McCartney'.

      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 :

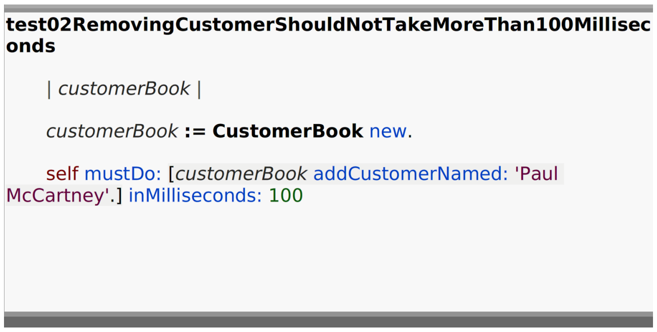


```
mustDo: aBlock inMilliseconds: milliseconds

      | 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.

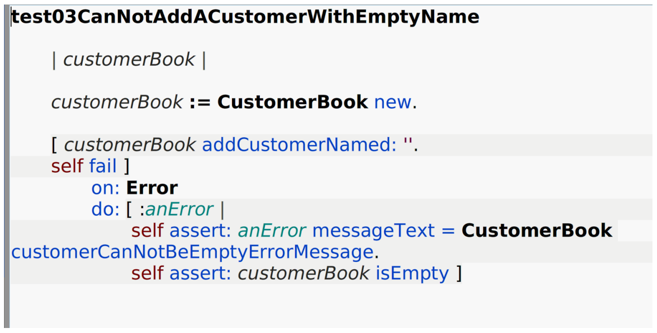


```
test01AddingCustomerShouldNotTakeMoreThan50Millisecon
ds

      | customerBook |

      customerBook := CustomerBook new.

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

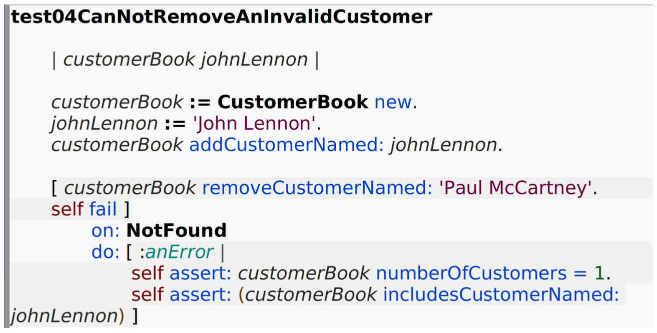


```
test02RemovingCustomerShouldNotTakeMoreThan100Millisec
onds

      | customerBook |

      customerBook := CustomerBook new.

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

2.2 Tests 3 y 4

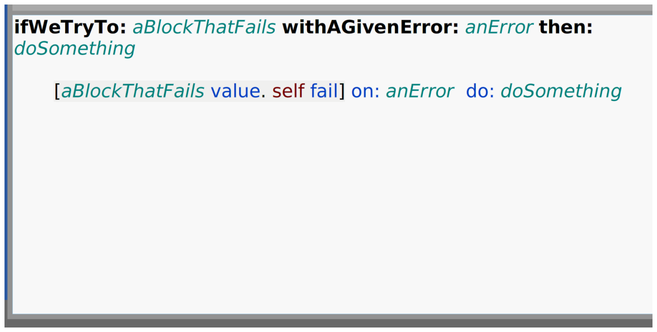


```
test03CanNotAddACustomerWithEmptyName

      | customerBook |

      customerBook := CustomerBook new.

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

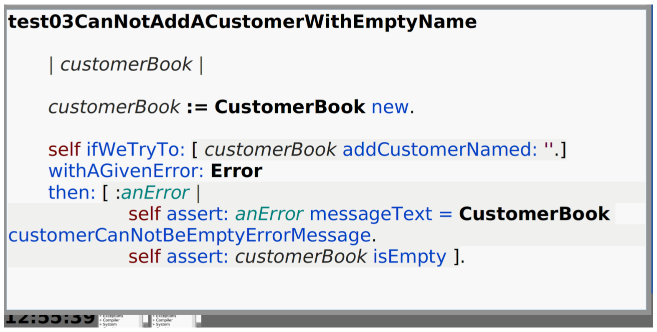


```
test04CanNotRemoveAnInvalidCustomer

      | 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) ]
```

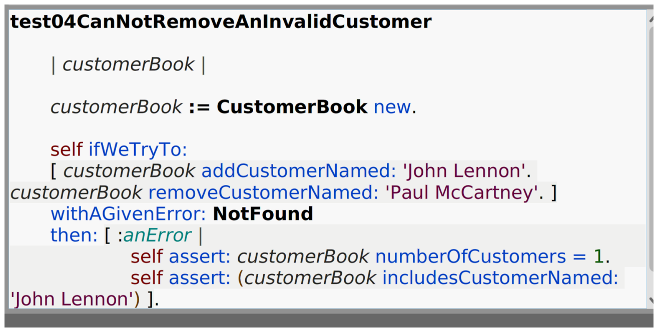
A partir de ellos, vemos que la idea es : realizar una acción, fallar, capturar el error y hacer algo. Esto podemos abstraerlo y construir un método como **ifWeTryTo: withAGivenError: then:** que :



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

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



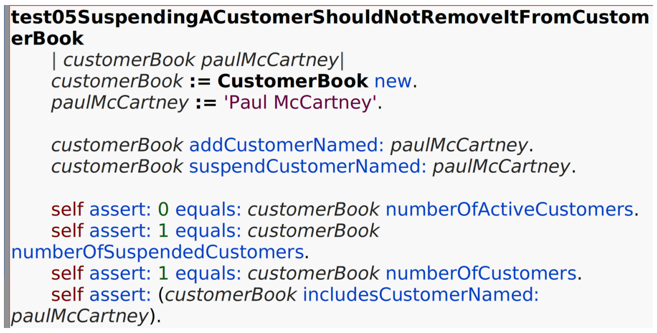
```
test03CanNotAddACustomerWithEmptyName

	| customerBook |

	customerBook := CustomerBook new.

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



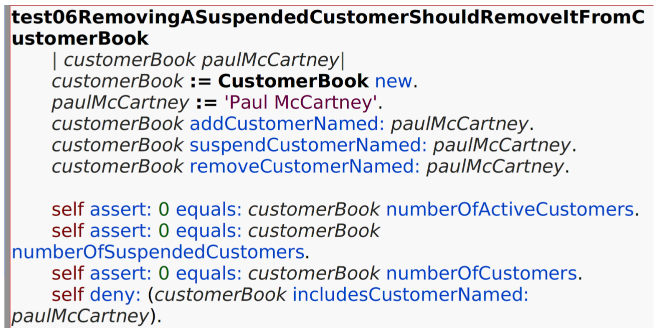
```
test04CanNotRemoveAnInvalidCustomer

	| 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') ].
```

** tal vez se podría reescribir mejor

2.3 Tests 5 y 6

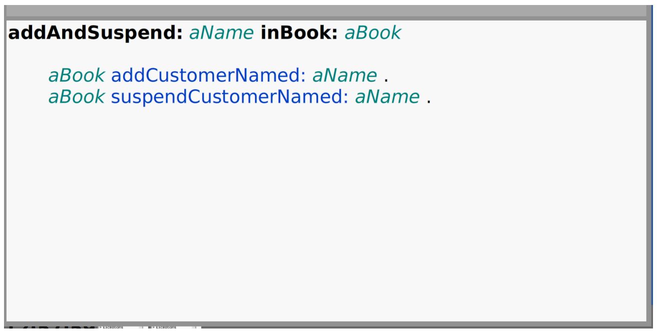


```
test05SuspendingACustomerShouldNotRemoveItFromCustom
erBook
	| customerBook paulMcCartney|
	customerBook := CustomerBook new.
	paulMcCartney := 'Paul McCartney'.

	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).
```

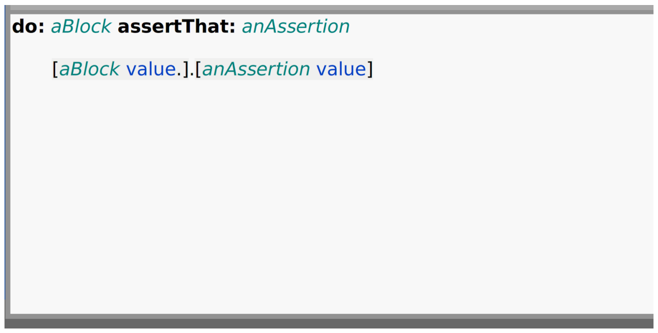


```
test06RemovingASuspendedCustomerShouldRemoveItFromC
ustomerBook
	| customerBook paulMcCartney|
	customerBook := CustomerBook new.
	paulMcCartney := 'Paul McCartney'.
	customerBook addCustomerNamed: paulMcCartney.
	customerBook suspendCustomerNamed: paulMcCartney.
	customerBook removeCustomerNamed: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

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

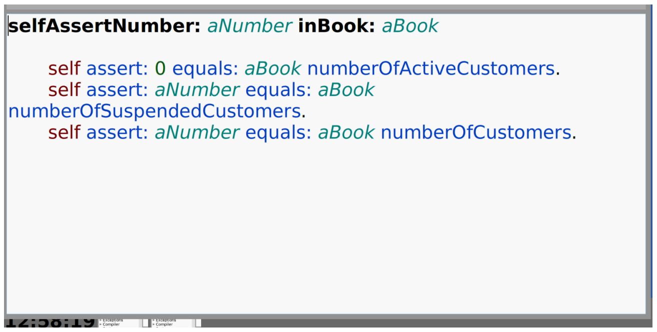
En estos caso vemos que en ambos se agrega y suspende un customer. Eso se puede abstraer como así también todos los asser en métodos como **addAndSuspend: inBook:**, **do: assertThat** y **selfAssertNumber: inBook:**



```
addAndSuspend: aName inBook: aBook

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

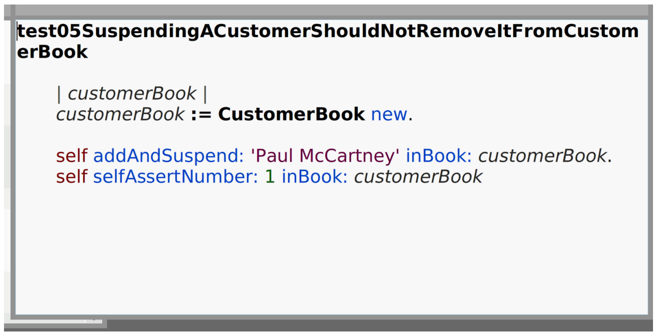


```
do: aBlock assertThat: anAssertion

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

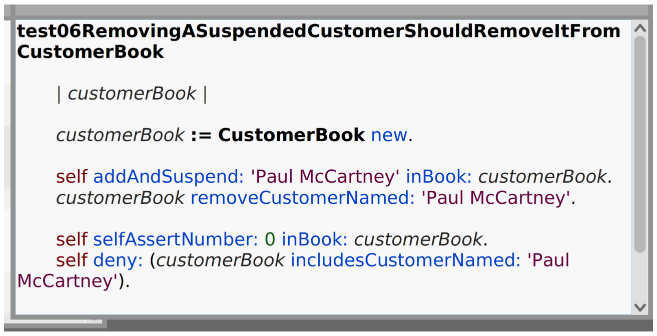


```
selfAssertNumber: aNumber inBook: aBook

	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 :

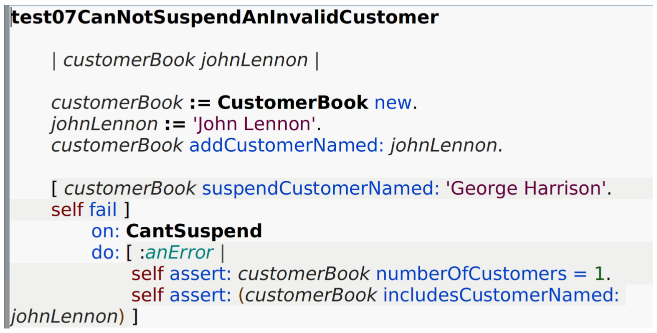


```
test05SuspendingACustomerShouldNotRemoveItFromCustom
erBook

	| customerBook |
	customerBook := CustomerBook new.

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

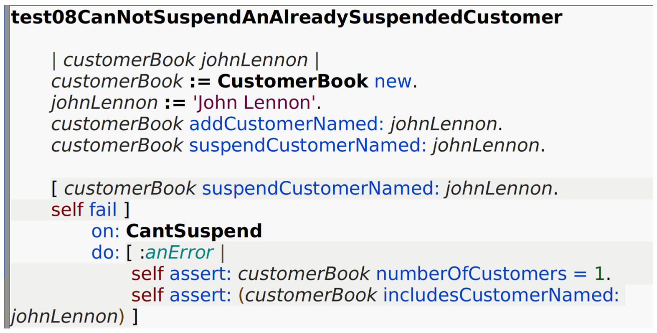


```
test06RemovingASuspendedCustomerShouldRemoveItFrom
CustomerBook

	| 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

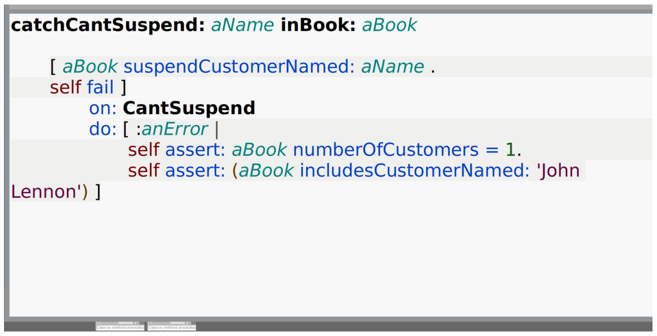


```
test07CanNotSuspendAnInvalidCustomer

	| 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) ]
```



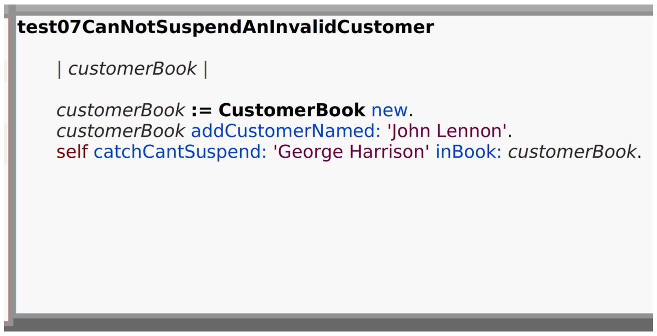
```
test08CanNotSuspendAnAlreadySuspendedCustomer

	| 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:**

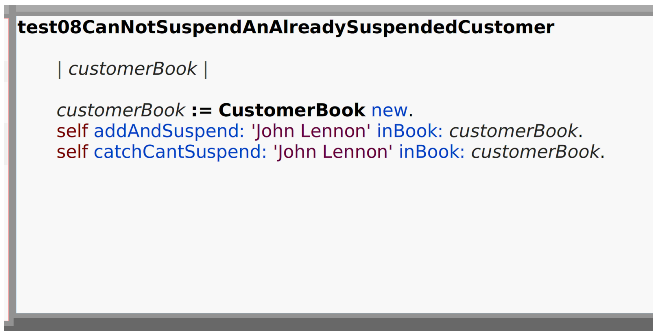


```
catchCantSuspend: aName inBook: aBook

	[ 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 :



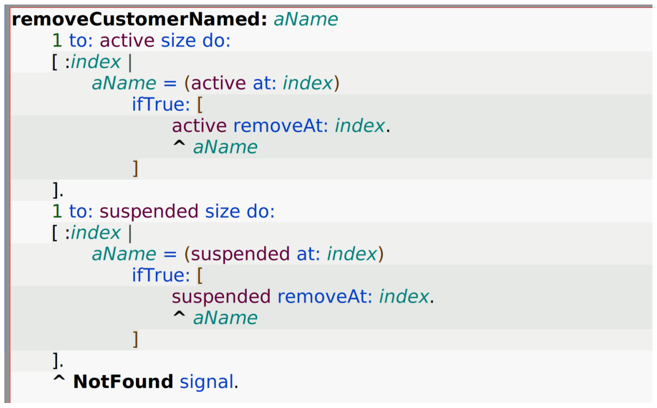
```
test07CanNotSuspendAnInvalidCustomer

	| 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...



```
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.
```