Practica CAP - Informe

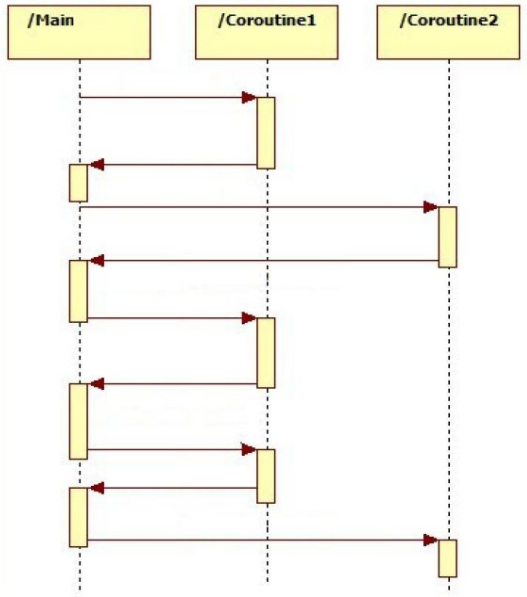
Sandra Flores Hidalgo, Raul Mateo Beneyto

Introducció

La pràctica de CAP d'enguany serà una investigació del concepte general d'estructura de
control, aprofitant les capacitats d'introspecció i intercessió que ens dóna Smalltalk. Estudiarem les conseqüències de poder guardar la pila d'execució (a la que tenim accés gràcies a la pseudo-variable thisContext) per fer-la servir i/o manipular-la. El fet de poder guardar i restaurar la pila d'execució d'un programa ens permet implementar qualsevol estructura de control i implementar la versió més flexible i general de les construccions que manipulen el flux de control d'un programa: les continuacions. Utilitzant les continuacions implementarem una estructura de control anomenada Corutina (coroutine).

Enunciat

La idea de l'estructura de control anomenada corutina és la següent: imaginem una funció (o procediment, o subrutina) tal i com ja les coneixem de C o C++. Les funcions s'invoquen, executen el seu cos, i quan acaben retornen. Si les tornem a invocar, torna a executar-se tot el cos de la funció, i quan acaba retorna. Les corutines funcionen diferent: Quan una corutina C1 invoca una altra corutina C2, s'atura i espera que se la torni a invocar. Si això passa, l'execució de C1 es reprén just en el moment en que va invocar C2; si ara C1 torna a invocar C2, l'execució de C2 es reprén en el moment que va decidir invocar a un altre corutina. La idea es pot veure en aquest gràfic:



Hi ha diverses maneres de definir les corutines, en funció de diverses característiques i la literatura sobre el tema és nombrosa. Nosaltres ens limitarem a una definició senzilla, adaptada a Smalltalk i a la OOP, que ens permetrà jugar amb el concepte1. Això és el que us demano a l'enunciat, tot seguit.

Cal fer una classe: **Coroutine**, tal que construeixi objectes que es comportin com a corutines (en aquest sentit, podriem pensar que són com una mena de blocs). Per a això tenim un mètode **Coroutine class >> #maker:** que farem servir per instanciar aquesta classe. Caldrà passar un bloc com a paràmetre, que és on posarem el codi de la corutina. Aquest bloc serà un bloc de dos paràmetres:

<instancia de corutina> ← **Coroutine maker: [:resume :value | . . .]**

En els punts suspensius és on posarem el codi de la corutina. Aquesta corutina, via el paràmetre **:resume**, pot invocar altres corutines:

resume value: <nom corutina> value: <valor passat a la corutina invocada>

ja que **resume** ha de ser un bloc amb dos paràmetres: el primer és una referència a una altra corutina, el segon és el valor que se li passarà a aquesta corutina com a valor de retorn de la crida a corutina que va fer el darrer cop que es va executar.

La manera com s'invoca una corutina serà mitjançant un mètode de la classe **Coroutine >> #value:**. En un programa fet amb corutines només cal arrencar la primera corutina, a partir d'aquest moment les corutines es comencen a cridar entre elles. El bloc que passarem com a paràmetre **:resume** al bloc amb que es construeix la corutina (el paràmetre de **#maker:**) si el pensem bé, pot ser sempre el mateix i independent de qualsevol codi que posem dins les corutines. També cal que penseu que la corutina, quan s'invoca per primer cop, ha de posar en marxa el codi especificat en el bloc que es passa com a paràmetre a **#maker:**, però després, quan s'invoca retornant d'una crida anterior (amb **resume value: c value: v**), ha de fer una altra cosa (ha de fer **c value: v**, però no només això! aquesta és una de les coses que heu de pensar).

Veiem un exemple. Si executem:

```
| a b c |
a := Coroutine maker: [ :resume :value |
        'This is A' traceCr.
       ('Came from ', (resume value: b value: 'A')) traceCr..
        'Back in A' traceCr.
       ('Came from ', (resume value: c value: 'A')) traceCr ].
b := Coroutine maker: [ :resume :value |
        '   This is B' traceCr.
       ('   Came from ', (resume value: c value: 'B')) traceCr..
        '   Back in B' traceCr.
       ('   Came from ', (resume value: a value: 'B')) traceCr ].
c := Coroutine maker: [ :resume :value |
        '      This is C' traceCr.
       ('      Came from ', (resume value: a value: 'C')) traceCr..
        '      Back in C' traceCr.
       ('      Came from ', (resume value: b value: 'C')) traceCr ].
a value: nil          "Aquest valor que passem a 'a' és irrellevant"
```

El resultat al Transcript és:

```
This is A
       This is B
           This is C
Came from C
Back in A
           Came from A
           Back in C
       Came from C
       Back in B
Came from B
```

Apartat A

Implementació de la Coroutine

Primer de tot, hem creat una nova classe **Coroutine** amb les variables d'instància que necessitarem: **aBlock** (que serà la part del codi a executar), **aContext** (per saber si executar el bloc de codi de la instància, o continuar amb la següent corutina), **aResumeBlock** (serà el bloc de codi que utilitzarem per iniciar una corutina i pausar-la fins que torni a ser cridada).

```
Object subclass: #Coroutine
   instanceVariableNames: 'aBlock aContext aResumeBlock'
   classVariableNames: ''
   package: 'Practica-CAP'
```

Seguidament, hem creat els **getters** i **setters** automàticament amb **Refactoring -> Class Refactoring -> Generate Accessors**.

A continuació hem definit un mètode de classe anomenat **maker**, tal i com posava a l'enunciat.

```
maker: aBlock
   "Initialize a coroutine"

   | aCoroutine |
   aCoroutine := self new.
   aCoroutine aBlock: aBlock.
   aCoroutine aContext: nil.
   aCoroutine aResumeBlock: [ :resume :value | Continuation callcc:
                                                [
                                                :cc | aCoroutine aContext: cc.
                                                resume value: value.
                                                ].
                            ].
   ^ aCoroutine.
```

Aquest mètode és un constructor que genera una nova instància de **Coroutine**; en la que li passem un bloc de codi que serà assignat a l'atribut **aBlock** de la nova instància. Com aquest mètode serà executat un sol cop (al principi) li assignem **nil** a **aContext**. Finalment, assignem a **aResumeBlock** el bloc de codi que s'encarregarà d'emmagatzemar l'execució recursiva de les corutines.

Per acabar, ens falta crear el mètode **value**, que s'encarregarà de redirigir l'execució del següent bloc de codi, depenent del valor de **aContext**.

```
value: aString
   "How to continue with the execution"

   aContext notNil
      ifTrue: [ aContext value: aString ]
      ifFalse: [ aBlock value: aResumeBlock value: aString ].
```

En cas de que **aContext** sigui **nil**, executarà el bloc de codi **aBlock** assignat a la mateixa instància. En cas contrari, que no sigui **nil** se li passarà el valor al codi asignat a **aContext**.

Tests per assegurar el correcte funcionament de les Coroutines

Classe CoroutineTests

```
TestCase subclass: #CoroutineTests
   instanceVariableNames: 'resultDefault resultFactorial1 resultFactorial2 wordsResult trianguloResult'
   classVariableNames: ''
   package: 'Practica-CAP'
```

El primer pas per implementar els tests és crear la classe **CoroutineTests**. Pel correcte funcionament dels tests hem de assignar **TestCase** com a classe pare. Seguidament indiquem les variables d'instància necessàries pels tests que implementarem.

Mètode setUp

Mètode per inicialitzar les variables d'instàncie per, posteriorment, comparar amb els resultats dels tests.

testDefault

A l'enunciat de la pràctica ens indiquen el primer test que hauria de passar per saber si la **Coroutine** es comporta de la manera que s'espera. Així que el primer test implementat és aquest.

testReverseWords

Aquest test, donades dos variables **word** i **iteracions**, imprimeix per pantalla la paraula word repetida **iteracions** cops del revés. Es tracta de dos coroutines, una encargada de fer un *loop for* de **1** a **iteracions**, i per cada iteració crida a la continuació de la segona coroutina, on aquesta executa un bucle infinit (*while(true);*), on cada iteració rep la paraula word rebuda de la primera coroutine i el retorna *reversed.* Seguidament la primera coroutine se'l guarda i segueix amb la següent iteració.

testTriangulo

Aquest test és bastant semblant a l'anterior però amb alguna sutil diferencia. Tracta de dues coroutines que, donades un numero de iteracions, retorna una piramide de sequencies de números seguits. Per exemple:

Iteracions: 5
1
12
123
1234
12345

La primera coroutina executa un *loop for* del **0** a **iteracions**, per cada iteració crida a la continuació de la segona coroutine amb el valor de **i**, on també executa un bucle infinit com a l'anterior test. Però en aquest, per cada iteració del bucle infinit executa un *loop for* de **0** al **valor** que li ha passat la primera coroutina, i guarda els valors en una array. Al acabar el bucle for, cridarà la continuació de la primera coroutine on aquesta seguirà amb la següent iteració després de guardar l'array que ha tornat la segona coroutine a l'array **res**ultat.