Pràctica CAP Q1 curs 2014-15
(*Blink*)

- A realitzar en grups de **2 persones.**
- A entregar com a molt tard el **20 gener de 2014.**

Descripció resumida:
tl;dr → Aquesta pràctica va de continuacions.
La pràctica de CAP d'enguany serà una investigació guiada 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*.

Material a entregar:
tl;dr → Amb l'entrega del codi que resol el problema que us poso a la pràctica no n'hi ha prou. Cal entregar un informe i els tests que hagueu fet.
Haureu d'implementar el que us demano i entregar-me finalment un ***informe*** on m'explicareu, què heu aprés, i *com* ho heu arribat a aprendre (és a dir, m'interessa especialment el codi lligat a les proves que heu fet per saber si la vostra pràctica és correcte). Les vostres respostes seran la demostració de que heu entés el que espero que entengueu. El format de l'informe és lliure, i el ***codi*** que m'heu d'entregar me'l podeu entregar de diverses maneres: via un fitxer .st obtingut d'un File Out, via un paquet .mcz o via un paquet a SmalltalkHub. Qualsevol d'aquestes maneres és vàlida. Utilitzareu **Pharo 3.0** (`pharo.org`) per fer la pràctica.

Nota: Abans de començar, llegiu i estudieu amb atenció el capítol 14, *Blocks: A Detailed Analysis*, del llibre *Deep into Pharo* (`deepintopharo.com`). Heu d'entendre perfectament què és un context d'execució (instància de `MethodContext`). Ja ho vam explicar a classe, però l'explicació en aquest capítol d'aquest llibre és més completa. També repasseu el que vam explicar a classe de continuacions (si voleu, també podeu veure el capítol 10 de la tercera temporada del Doctor Who, anomenat *Blink*, de l'any 2007).

Enunciat:
Tothom sap que la Tardis és la màquina del temps en la que viatgen habitualment el Doctor i els seus acompanyants. Així, el Doctor és capaç d'anar i transportar objectes i persones entre diferents moments del temps. Això forma part de la tecnologia que dominen els Senyors del Temps, originaris del planeta Gallifrey. Aquests, però, no són els únics que tenen habilitats de control sobre el temps. Existeixen uns éssers espantosos anomenats *àngels ploraners* que s'alimenten de l'energia residual derivada dels viatges temporals. Així, els àngels ploraners, per sobreviure, envien altres éssers enrera en el temps a un moment indeterminat. Això ho poden fer només tocant-los.

Si considerem el flux d'un programa com una mena de flux temporal lineal (no estem tenint en compte ni el *multi-threading* ni la concurrència ni el paral·lelisme) on les accions elementals del procés de càlcul tenen lloc seqüencialment, i a més sabem què són les continuacions, podem plantejar-nos fer noves estructures de control que serveixin per emular el que fan els àngels ploraners, i les possibles correccions que el doctor pot fer de les seves malifetes.

Afegirem una classe nova `Tardis` amb tres mètodes nous, `#fita:`, `#angelPloraner:` i `#doctor:` (aquests mètodes van al ***`Class side`***, perque no ens cal instanciar la classe). Un procés de càlcul que utilitzi aquests missatges ha d'acabar *malgrat l'existència d'àngels ploraners*. Un àngel ploraner enviarà el control a la *darrera* fita creada, amb el mateix entorn que tenia quan es va marcar la fita. El valor passat a l'àngel ploraner serà el retornat quan es torni a la fita, com si fos el valor retornat per l'enviament del missatge `#fita:` a la classe `Tardis`. Si trobéssim un altre àngel ploraner, aquest ens enviaria a la *penúltima* fita, no a la que acabem d'utilitzar. És a dir, cada fita es pot fer servir només un cop. Una successió d'angels ploraners ens envia cada cop més enrera, cap a fites creades abans de les fites ja utilitzades.

El Doctor és capaç d'arreglar (parcialment) el mal que fan els àngels ploraners. Cada cop que ens trobem al Doctor (és a dir, l'enviament del missatge `#doctor:`), avancem allà on ha actuat el darrer àngel ploraner que hem trobat, amb el mateix entorn que tenia l'àngel ploraner quan ens ha fet retrocedir a una fita. L'argument que li donem al Doctor és el que retornarà l'enviament de missatge de l'àngel ploraner.

Si s'invoca un àngel ploraner sense que s'hagi marcat cap fita, o s'invoca al Doctor sense que cap àngel ploraner hagi fet res, el resultat és que els enviaments de missatge `#angelPloraner:` i `#doctor:` es comporten com la funció identitat, és a dir, retornen el seu argument i ja està.

Concretant, implementeu la classe `Tardis` amb els mètodes en el ***`Class side`*** següents:

- `fita: anObject`

El resultat d'invocar `Tardis fita: <expressió>` retorna el resultat d'avaluar l'`<expressió>` (però aquesta avaluació no la fa el mètode `#fita:`, es fa sempre pel fet de passar l'`<expressió>` com a argument d'un missatge). És una mena de funció identitat. El que té d'interessant aquest mètode és que, com a *side effect*, guarda el context actual per ser utilitzat després per un àngel ploraner.

- `angelPloraner: anObject`

El resultat d'invocar `Tardis angelPloraner: <expressió>` envia el flux de control allà on hem creat una fita més recentment (l'última fita creada). L'`<expressió>` avaluada és el resultat de la invocació `Tardis fita: <expressió2>`, ara que es fa per segona vegada. Aquesta invocació també guarda el context actual, tal com feiem en crear una fita, per ser utilitzat després pel doctor.

- **`doctor: anObject`**

El resultat d'invocar **`Tardis doctor: <expressió>`** envia el flux de control allà on hem invocat un àngel ploraner més recentment, amb el mateix entorn que tenia l'àngel ploraner quan va actuar per enviar-nos fins a una fita. L'**`<expressió>`** avaluada és el resultat de la invocació **`Tardis angelPloraner: <expressió2>`**, ara que es fa per segon cop.

Aquests són els principals mètodes que us caldrà implementar. Potser us caldrà inicialitzar la **`Tardis`**, i per a això voldreu tenir un **`Tardis class >> initialize`** també.

Si ho heu fet bé, després d'executar aquest codi:

```
Tardis initialize.
(TA auxWith:$a and:((TA auxWith:$b and:(Tardis fita:3)) = 4 ))
 ifTrue: [  Transcript show: 'aqui!'; cr.
            TA auxWith:$c and:(Tardis doctor:'tot va be').]
 ifFalse:[(TA auxWith:$d and:((TA auxWith:$e and:(Tardis fita: 4))= 5))
            ifTrue: [ TA auxWith:$f and:(Tardis doctor:(TA auxWith:$g and:(Tardis angelPloraner:4)))]
            ifFalse: [ Transcript show: 'hem arribat fins aqui!'; cr.
                       TA auxWith: $h and: (Tardis angelPloraner: 5).]]
```

heu de veure al **`Transcript`** els següents missatges:

```
==> b : 3
==> a : false
==> e : 4
==> d : false
hem arribat fins aqui!
==> e : 5
==> d : true
==> b : 4
==> a : true
aqui!
==> g : tot va be
==> h : tot va be
```

[on **`TA class >> #auxWith:and:`** és només un mètode auxiliar d'una classe **`TA`** auxiliar que escriu al **`Transcript`** un missatge i retorna el seu segon argument:

```
auxWith: anObject and: anotherObject
      Transcript show:'==> ',anObject asString,' : ',anotherObject asString; cr.
      ^ anotherObject
]
```