Final CAP

Curs 2017-18 (18/I/2018) Duració: 3 hores

1.- (1.5 punts) Explica quins són els *join points* triats per aquest *pointcut*, que vam veure a la solució de l'exemple de Fibonacci:

```
execution(* *.f(..)) && !cflowbelow(execution(* *.f(..)))
```

El pointcut `execution(* *.f(..)) && !cflowbelow(execution(* *.f(..)))` selecciona les crides de primer nivell de la funció `f`, excloent les crides recursives.

2.- (1.5 punts) Això ja us ho pregunto a la pràctica, així us faig començar a pensar-hi (si no heu començat encara): Implementeu en Javascript una funció `callcc(f)` que funcioni com l'estructura de control que ja coneixeu de Pharo, fent servir, naturalment, la funció `Continuation()` de Javascript.

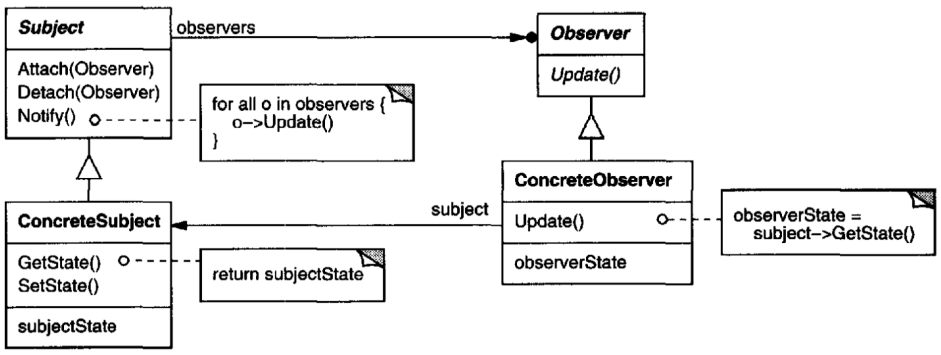
Recordem com funciona `callcc`: Aquesta funció s'invoca amb una funció com a paràmetre, que s'invoca amb la continuació corresponent a la crida a `callcc` com a paràmetre:

```
function callcc(f) {
     var k = new Continuation();
     return f(k);
}
```

3.- (1 punt) Doneu una expressió pel següent *pointcut*: L'execució de qualsevol mètode definit al codi font d'una classe que és subclasse de la classe A, però que NO està definit a la classe A.

```
execution(* *(..)) && within(A+) && !within(A)
```

4.- (2 punts) El patró `Observer` es representa en llenguatges amb classes de la següent manera (segons el llibre clàssic *Design Patterns*):



La idea és, molt per sobre, que hi ha un objecte (instància de `ConcreteSubject`) que admet observadors que seran notificats de canvis en aquest objecte, i objectes que poden notificar l'interés per aquests canvis (les instàncies de `ConcreteObserver`). En Javascript sense classes podem simplificar considerant que tenim una funció `Subject()` amb la que

crear objectes susceptibles de ser observats: **`var subj = new Subject()`**, amb operacions **`subj.attach(observer)`**, **`subj.detach(observer)`**, **`subj.notify()`**, **`subj.getState()`**. També necessitarem la funció **`Observer`** tal que permeti crear observadors: **`var o = new Observer()`** amb operacions **`o.update(subject)`**.

Feu una implementació *senzilla* d'aquestes funcions **`Subject()`** i **`Observer()`**., considerant que no podeu ser massa específics a l'hora d'implementar **`getState`** o **`update`**.

```
function Subject() {
  this.observers = [];
}

Subject.prototype.attach = function (observer) {
  this.observers.push(observer);
}

Subject.prototype.detach = function (observer) {
  this.observers = this.observers.filter( function (obs) {
    return (obs !== observer);
  })
}

Subject.prototype.notify = function () {
  this.observers.forEach( function (observer) {
    observer.update(this);
  })
}

Subject.prototype.getState = function () { /* . . . */ }

function Observer() { }

Observer.prototype.update = function (subject) {
  /* ... fer alguna cosa amb subject.getState() ... */
}
```

5.- (2 punts) Ja vam veure a classe que ECMAScript 5 inclou a **`Object`** la funció **`create(p)`** per poder crear objectes amb un prototipus donat **`p`**. En versions d'ECMAScript anteriors, on **`create(p)`** no existeix, per fer el mateix hom podia definir aquesta funció:

```
function inherit(p) {
   if (p == null) throw TypeError(); // p no ha de ser null
   var t = typeof p;
   if (t !== "object" && t !== "function") throw TypeError();
   function f() {};
   f.prototype = p;
   return new f();
```

```
}
```

Explica amb el màxim detall possible com funciona **`inherit(p)`**.

Després de comprovar que **`p`** sigui un objecte adequat com a prototipus d'un altre, el que fa el codi és:

`function f() {};` aquí definim una funció *de mentida*, no cal que faci res
`f.prototype = p;` assignem **`p`** com a valor de l'atribut **`prototype`** de **`f`**
`return new f();`

finalment retornem una *instància* de **`f`**. És a dir, sabem que si invoquem una funció amb un **`new`** al davant, l'objecte creat tindrà com a prototipus el mateix objecte referenciat pel valor de l'atribut **`prototype`** de **`f`**.

Així doncs, el resultat és un objecte buit (cap propietat) que té com a prototipus a **`p`**.

6.- (2 punts) El llenguatge de programació C disposa d'una llibreria que proveeix de dues funcions, **`setjmp`** i **`longjmp`** (*header* **`setjmp.h`**, si es vol fer servir). Veieu-ne dos exemples (compilar amb **`gcc -o exemple exemple.c`** i executar amb **`./exemple`**)

exemple1.c

```
#include <stdio.h>
#include <setjmp.h>

int main() {

  jmp_buf env;
  int i;

  i = setjmp(env);
  printf("i = %d\n", i);

  if (i != 0)
     return(0);

  longjmp(env, 2);
  printf("surt aixo?\n");
}
```

RESULTAT DE COMPILAR I EXECUTAR:

```
i = 0
i = 2
```

exemple2.c

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf buf;

void segon(void) {
    printf("segon\n");
    longjmp(buf,1);
}

void primer(void) {
    segon();
    printf("primer\n");
}

int main() {
    if (!setjmp(buf))
        primer();
    else
        printf("main\n");
   return 0;
}
```

RESULTAT DE COMPILAR I EXECUTAR:

```
segon
main
```

Ja sé que no hem vist ni C (tot i que el coneixeu d'altres assignatures) ni la parella **`setjmp`**/**`longjmp`**, però a classe hem parlat molt de reificar la pila d'execució, de continuacions, de salts no locals, etc. Amb tot aquest coneixement i aquest dos exemples

senzills us podeu fer una idea aproximada de què fan **`setjmp`**/**`longjmp`** (investigueu una mica).

Així doncs, *expliqueu què fan* **`setjmp`**/**`longjmp`** i, a partir del que heu deduit, *feu una comparació amb* **`callcc:`** *de Pharo i* **`Continuation()`** *de Javascript*.

Nota: Entenc que us quedareu només amb una idea superficial i no podreu deduir les subtileses de **`setjmp`**/**`longjmp`**. No passa res, ja m'ho penso això, no espero que ho trobeu.

A la Wikipedia ens diuen:

`setjmp`: *Sets up the local jmp_buf buffer and initializes it for the jump. This routine saves the program's calling environment in the environment buffer specified by the env argument for later use by longjmp. If the return is from a direct invocation, setjmp returns 0. If the return is from a call to longjmp, setjmp returns a nonzero value.*

`longjmp:` *Restores the context of the environment buffer env that was saved by invocation of the setjmp routine in the same invocation of the program. Invoking longjmp from a nested signal handler is undefined. The value specified by value is passed from longjmp to setjmp. After longjmp is completed, program execution continues as if the corresponding invocation of setjmp had just returned. If the value passed to longjmp is 0, setjmp will behave as if it had returned 1, otherwise, it will behave as if it had returned value.*

Si ho comparem amb les continuacions *completes* que ens proporcionen Pharo, Scheme, Javascript, Standard ML i altres llenguatges de programació veiem que **`setjmp`**/**`longjmp`** no són més que un cas especial, on suposem que el context al que tornem encara existeix, ja que no es guarda la pila d'execució completa per restaurar-la després: *If the function in which setjmp was called returns, it is no longer possible to safely use longjmp with the corresponding jmp_buf object. This is because the stack frame is invalidated when the function returns.*