Sviluppo software in gruppi di lavoro complessi[1]

Mattia Monga

Dip. di Informatica
Università degli Studi di Milano, Italia
mattia.monga@unimi.it

Anno accademico 2016/17, I semestre

Lezione XIX: *Eiffel*

Procedurale vs. Dichiarativo

Spesso si scrivono le “stesse” cose due volte:

```
do                                ensure
  balance := balance - x            balance = old balance - x
```

- Implementazione e specifica
- How & What

Il client è responsabile delle precondizioni, il fornitore di postcondizioni e invarianti.

Contratti ed ereditarietà

Il *principio di sostituzione di Liskov* stabilisce che, perché un oggetto di una classe derivata soddisfi la relazione `is-a`, ogni suo metodo:

- deve essere accessibile a pre-condizioni uguali o piú deboli del metodo della superclasse;
- deve garantire post-condizioni uguali o piú forti del metodo della superclasse;

Altrimenti il “figlio” non può essere sostituito al “padre” senza alterare il sistema.

Principio di sostituibilità

Le due condizioni sono quindi:

$$PRE_{parent} \implies PRE_{derived} \quad (1)$$
$$POST_{derived} \implies POST_{parent} \quad (2)$$

- (1) in un programma corretto non può succedere che PRE_{parent} valga e $PRE_{derived}$ no; l'oggetto *evoluto* deve funzionare in ogni stato in cui funzionava l'originale: non può avere **obbligazioni** piú stringenti, semmai piú lasche.
- (2) in un programma corretto non può succedere che valga $POST_{derived}$ ma non $POST_{parent}$; un stato corretto dell'oggetto *evoluto* deve essere corretto anche quando ci si attende i **benefici** dell'originale.

Principio di sostituibilità (cont.)

Un modo per garantire che le condizioni (1) e (2) siano automaticamente vere consiste nell'assumere implicitamente che, se la classe evoluta specifica esplicitamente una precondizione P e una postcondizione Q, le reali pre- e post-condizioni siano:

$$PRE_{derived} = PRE_{parent} \vee P \quad (3) \qquad PRE_{parent} \implies PRE_{derived}$$
$$POST_{derived} = POST_{parent} \wedge Q \quad (4) \qquad POST_{derived} \implies POST_{parent}$$

In Eiffel: `require else` e `ensure then`

Contratti "astratti"


```
extend (x: G)
-- Add `x' at end of list.
  require
      space_available: not full
  deferred
  ensure
      one_more:
      count = old count + 1
  end

full: BOOLEAN
-- Is representation full?
-- (Default: no)
    do
        Result := False
    end
```

Stronger precondition... ma **weaker (uguali in realtà) in astratto**

```
full: BOOLEAN
  -- Is representation full?
  -- (Answer: if and only if
  --  number of items is equal
  --  to capacity)
  do
      Result := (count = capacity)
  end
```


Problema: i parametri...

- Animale mangia Cibo
- Mucca (is_a Animale) mangia Erba (is_a Cibo)

Ma questa **covarianza** è contraria al principio di Liskov perché restringe le precondizioni. La controvarianza (Mucca mangia Cosa, Sather) e l'invarianza (Mucca mangia Cibo, Java) vanno bene.
Eiffel invece è covariante... (il che, impedendo un controllo di conformità statico, introduce parecchie complicazioni $\rightsquigarrow$ CATcall, run time type identification...).

Trattamento delle situazioni anomale

Nel modello di Eiffel hanno un ruolo importante le eccezioni, che vengono trattate in un modo differente da quello dei piú diffusi linguaggi di programmazione (Ada-like).

Exception

An exception is a run-time event that may cause a routine call to fail (**contract violation**). A failure of a routine causes an exception in its caller.

Anti-pattern


```
sqrt (n: REAL): REAL
do
 if x < 0.0 then
  raise Negative
 else
  normal_square_root_computation
end
exception
 when Negative =>
  print("Negative argument%N")
  return
 when others => ..
end
```

In questo caso il meccanismo delle eccezioni è usato come strumento di controllo del flusso!

Il trattamento delle eccezioni in Eiffel

Due modalità:

1. **Failure** (organized panic): clean up the environment, terminate the call and report failure to the caller.
2. **Retry**: attempt to change the conditions that led to the exception and to execute the routine again from the start.

Per trattare il secondo caso, Eiffel introduce il costrutto `rescue/retry`. Se il corpo del 'rescue' non fa 'retry', si ha un failure.

Esempio


```
quasi_inverse (x: REAL): REAL
                -- 1/x if possible, otherwise 0
        local
                division_tried: BOOLEAN
        do
                if not division_tried then
                        Result := 1/x
                else
                        Result := 0
                end
        rescue
                division_tried := True
                Retry := True
        end
```

Correttezza

Per ogni feature (pubblica) f:

- $\{PRE_f \wedge INV\}\ body_f\ \{POST_f \wedge INV\}$
- $\{True\}\ rescue_f\ \{INV\}$
- $\{True\}\ retry_f\ \{INV \wedge PRE_f\}$

L'assenza di `rescue` dovrebbe corrispondere perciò a una procedura che ripristina l'invariante!

Hello world!


```
note
    description: "Esempio base per iniziare"
    author: "Mattia Monga"

class
    HELLO

create
    make

feature -- sezione: puo` contenere molte feature
    make
        local
         s: STRING
        do
            s := "Hello"
            print (s + " World%N")
            io.put_string ("Hello World"); io.put_new_line
        end
end
```


Si può compilare senza Eiffel Studio:
`ec -class_file.e hello.e`

Caratteristiche di Eiffel

- Le variabili (entities) possono essere reference (default) o `expanded`
- `Void`, `True`, `False`, `Result`, `Current`, `Precursor`, `ANY`, `NONE`
- assegnamento `:=`, clone `x := y.twin` (o `x := y.deep_twin`)
- reference `x = y`, expanded `x ~ y`, `equal(x, y)` funziona anche se `x` è `Void`
- gli attributi sono feature in sola lettura

Strutture di controllo


```
if .. then
   ..
elseif .. then
   ..
else
   ..
end
```

```
inspect
    exp
when v1 then
    inst
when v2 then
    inst2
..
else
    inst0
end
```

```
from my_list.start until my_list.off
loop
   print (my_list.item) my_list.forth
end

-- command
across my_list as ic
loop
  print (ic.item)
end

-- query
across my_list as ic
all
  ic.item > 3
end

-- query
across my_list as ic
some
  ic.item > 3
end
```

Particolarità ereditarietà

- ereditarietà multipla
- `rename`, `export`, `undefine`, `redefine` (`Precursor`), `select` (utile solo nel caso di *diamond inheritance*)
- ereditarità non conforme `inherit {NONE}`: senza polimorfismo, riuso puro: analogo al copia e incolla...