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 2017/18, I semestre

Lezione VI: *Design by Contract*

Contratti

La tripla di Hoare $\{P\}S\{Q\}$ può diventare un **contratto** fra chi *implementa* (fornitore) S e chi *usa* (cliente) S

- L'implementatore di S si impegna a garantire Q in tutti gli stati che soddisfano P
- L'utilizzatore di S si impegna a chiedere il servizio in un stato che soddisfa P ed è certo che se S termina, si giungerà in uno stato in Q vale

Il lavoro dell'implementatore è particolarmente facile quando: Q è `True` (vera per ogni risultato!) o quando P è `False` (l'utilizzatore non riuscirà mai a portare il sistema in uno stato in cui tocchi fare qualcosa!). Weakest precondition (data Q) o strongest postcondition (data P) **determinano** il ruolo di una feature.

Eiffel

Eiffel

Un linguaggio *object-oriented* che introduce i **contratti** nell'interfaccia delle classi. Il contratto di default per un metodo ("feature") F è $\{True\}F\{True\}$.

```
feature
 decrement is
 -- Decrease counter by one.
 require
  item > 0              -- pre-condition
 do
  item := item - 1      -- implementation
 ensure
  item = old item - 1   -- post-condition
```

Eiffel

Eiffel è esplicitamente progettato come linguaggio "di progetto", non solo "di programmazione":

> *"specify, design, implement and modify quality software" [Ecma standard 367]*

"Programmazione in grande" con oggetti, derivati da classi organizzate in gerarchie di ereditarietà e raggruppate in cluster, che forniscono feature (command o query) ai loro client.

Organizzazione delle asserzioni

- pre/post-condizioni sulle feature (`require`, `ensure`)
- Invarianti di classe (`invariant`)
- asserzioni (`check`)
- loop invariant
  (`from .. invariant .. until .. variant .. loop .. end`)

Il supporto del linguaggio

- **invarianti di classe** sono condizioni che devono essere vere in ogni momento "critico", ossia osservabile dall'esterno. In pratica e come se facessero parte di ogni pre- e post-condizione.
- è possibile avere un supporto run-time alle **violazioni:** se una condizione non vale viene sollevata un'eccezione
- L'eccezione porta il sistema nel precedente stato stabile ed è possibile
  - terminare con un fallimento
  - riprovare


```
class ROOT_TEST_STABLE_STATES
create make
feature {NONE}
    secret: BOOLEAN
feature {ANY}
   make -- root class cannot have preconditions
    -- require ok_pre("make")
    do
      print("Executing make%N")
      mycommand; secret := TRUE
    ensure ok_post("make")
    end
  mycommand
    require ok_pre("mycommand")
    do
      print("Executing mycommand%N")
      secret := FALSE; myother("1"); secret := TRUE
      -- But what happens if myother is a "client"?
      -- secret := FALSE; Current.myother("2"); secret := TRUE
    ensure ok_post("mycommand")
    end
  myother (s: STRING)
    require ok_pre("myother")
    do
      print("Executing myother " + s + "%N")
    ensure ok_post("myother")
    end
  ok_inv: BOOLEAN do print("Checking ok_inv!%N"); Result := secret; end
  ok_pre (w: STRING): BOOLEAN do print("Checking ok_pre @ " + w + "%N"); Result := True; end
  ok_post (w: STRING): BOOLEAN do print("Checking ok_post @ " + w + "%N"); Result := True; end
invariant ok_inv
end
```

Demo I

```
class GCD
create make
feature gcd (x: INTEGER; y: INTEGER): INTEGER
    require
      positive_parms: x >= 0 and y >= 0
      not_zero: x /= 0 or y /= 0
    local t: INTEGER
    do
      if x = 0 or y = 0 then Result := x.max (y)
      else
       from Result := x; t := y
        invariant
         positive_result: Result > 0
         positive_t: t > 0
         gcd_inv: mathgcd(x, y) = mathgcd(Result, t)
        until Result = t
        loop
         if Result > t then Result := Result - t
         else t := t - Result
         end
        variant  t.max(Result)
       end
    end
    ensure
      positive_ris: Result > 0
      dividex: x = 0 or else x.integer_remainder(Result) = 0
      dividey: y = 0 or else y.integer_remainder(Result) = 0
      Result = x.min(y) or else across ((Result+1).to_integer |..| x.min(y)) as ic
                               all (x.integer_remainder(ic.item) /= 0
                               or y.integer_remainder(ic.item) /= 0) end
                              end
```


Demo II

```
  make do
   print("Ris: " + gcd(126,294).out + " %N")
   print("Ris: " + gcd(0,294).out + " %N")
  end

  mathgcd(x,y: INTEGER):INTEGER do
      from Result := x.min(y)
      until y.integer_remainder(Result) = 0
        and then x.integer_remainder(Result) = 0
      loop
        Result := Result - 1
      end
    end
end
```


Procedurale vs. Dichiarativo

Spesso si scrivono le "stesse" cose due volte:

```
do
  balance := balance - x
```

```
ensure
  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 (is_a Cosa)
- 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


```
div (num: REAL, denom: REAL): REAL
 require
   denom /= 0
 deferred

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

Correttezza

Per ogni feature (pubblica) f:

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