

Design by Contract di David Meyer

La nozione di contratto

Supponiamo di avere del codice strutturato come segue:

```
my_task is
do
  subtask 1 ;
  subtask2 ;
  subtask_n ;
end
-- my_task
```

Come implementatore del codice `my_task`, una decisione diventa fondamentale: dovrei implementare le subtask in maniera locale oppure gestirla esternamente (contract it out)?

Concretamente questo significa se implementare il codice all'interno del corpo di `my_task`, o creare delle routine secondarie accedervi (richiedendo i permessi) e implementare la task i-esima tramite la chiamata a quella routine. Generalmente la seconda scelta è giustificata da 2 motivi principali:

- Gestire la dimensione della routine principale, evitando così che risulti troppo estesa (quindi limitando la sua complessità)
- Cercare di trarre vantaggio di routine 'comuni' (richiamabili da più task), soprattutto quando esse sono già disponibili

Secondo Meyer l'esperienza di un programmatore software è fondamentale nella scelta della gestione delle routine.

Quando scegliamo di delegare l'implementazione della task ad una routine esterna, dobbiamo assicurarci che colui a cui deleghiamo l'implementazione della routine, concretamente questo è possibile solo tramite una stipula del contratto.

Un contratto, come si può intuire, ha il ruolo di proteggere entrambe le parti

1. Il cliente mettendo nero su bianco quanto deve essere fatto
2. Il contractor che ha così in testa quali sono i requisiti minimi/accettabili, un contractor così non può essere ritenuto responsabile di mansioni 'fuori-contratto'

Le asserzioni: contrattare nel mondo software

Secondo Mayer il modo migliore per produrre un software robusto è documentare le varie chiamate alla routine, o se dovesse risultare troppo macchinoso, quantomeno esplicitare gli obblighi di ogni chiamata (ovvero ciò che deve necessariamente garantire).

Tali condizioni prendono il nome di asserzioni: pre-condizioni e post-condizioni sulle routine individuali; invarianze di classe sulle routine di un determinata classe.

Pre-condizioni e post-condizioni

Nella specifica dei termini di un contratto software, dovremmo associare una pre-condizione ed una post-condizione ad ogni routine.

Le precondizioni esprimono i requisiti che ogni chiamata deve soddisfare in modo tale che risulti corretta, le post-condizioni esprimono le proprietà che vengono assicurate in modo tale che risultino corrette.

Nei linguaggi Eiffel-like la sintassi per la dichiarazione delle asserzioni è esplicita e potrebbe risultare così:

```
routine_name (argument declarations) is
    -- Header comment
    require
        precondition
    do
        routine body, i.e. instructions
    ensure
        postcondition
    end -- routine_name
```

Una possibile implementazione di questo codice:

```

put (element: T, key: STRING) is
    -- Insert element with key key
  require
    count < capacity
  do
    ... “Insertion algorithm” ...
  ensure
    count <= capacity;
    item (key) = element;
    count = old count + 1
  end -- put

```

Un possibile contratto stipulato tra cliente e contractor potrebbe essere del tipo:

	Obligations	Benefits
Client	Call <i>put</i> only on a non-full table	Get modified table in which <i>x</i> is associated with <i>key</i>
Contractor	Insert <i>x</i> so that it may be retrieved through <i>key</i>	No need to deal with the case in which table is full before insertion.

Vediamo da questa semplice tabella come ad entrambe le parti viene assegnata una certa responsabilità, ed entrambi esplicitano quali sono le loro richieste e i loro doveri.

Viene da porsi alcune domande, innanzitutto cosa succede se la precondition non è soddisfatta? In questo caso vediamo che il cliente chiama la put su una tabella già satura, il contractor non ha nessun obbligo di produrre un output corretto in quando i requisiti del contratto non sono stati rispettati in primo luogo dal cliente.

Inoltre, considerando che la presenza di errori nell'output non viene eliminata alla stipula del contratto, una routine dovrebbe poter gestire tutti i possibili input? Secondo Mayer no; ciò che deve essere ottimizzato non è la quantità di possibili errori non gestiti dal contratto, ma l'architettura più semplice che renda funzionante il software.


Note

The matter of who should deal with abnormal values is essentially a pragmatic decision about division of labor: the best solution is the one that achieves the simplest architecture. If every routine checked for every possible error in its calls, no useful work would ever be performed.


Who sould check?

Chi dovrebbe essere l'incaricato a gestire determinata condizioni? Il cliente o il contractor?

Secondo Mayer non c'e' una regola rigida in merito, ma comunque offre uno spunto da utilizzare come linea guida

 **Either the condition is part of the precondition, and must be guaranteed by the client; or it is not stated in the precondition, in which case the supplier must handle it.**

Inoltre qualcosa che personalmente ricollego al principio di Single Responsibility, (uno dei principi SOLID della progettazione software).

 **We never ask both client and supplier to be responsible for a consistency condition.**

Invarianza di classe e correttezza di classe

Seppure i concetti di pre-condizione e post-condizione sono comuni ad ogni linguaggio di programmazione, quello di invarianza di classe è tipico del linguaggio orientato agli oggetti, che in qualche modo ci dice qualcosa sulla correttezza della classe.

Un'invarianza di classe è una proprietà che deve essere soddisfatta da tutte le istanze della classe, ad esempio, riprendendo l'esempio che è stato fatto in precedenza su TABLE, un'invarianza di classe presuppone che tutte le tabelle devono rispettare la seguente:

```
0 ≤ count ≤ capacity
```

Anche in questo caso Eiffel ha una keyword per definire questa asserzione:

```
class TABLE [T] feature  
    ... Attribute and routine declarations for  
        put, item, delete, count, capacity, ...  
invariant  
    0 <= count <= capacity  
end -- class TABLE
```

Le proprietà che caratterizzano un'invarianza di classe sono:

- Deve essere soddisfatta da tutte le istanze della classe
- Deve essere preservata da ogni routine esterna

Gestire situazioni anomale

In un software che può essere considerato corretto ogni chiamata si dovrebbe trovare in uno dei seguenti stati:

- *Dimostrabile*
- *Protetta*

Questo viene garantito da assunzioni a priori o a posteriori, ma molto spesso non è sufficiente nella gestione di situazioni anomale, ad esempio nei casi in cui:

- operazioni la cui applicabilità viene valutata solo in fase di esecuzione
- Operazione con un bassa probabilità di fallimento
- Software con fault tolerance

Un modo per gestire situazioni anomale è tramite l'utilizzo di eccezioni

*if something_wrong then
 raise an_exception
end;
further_processing*

Mayer dichiara tre leggi sulla gestione delle eccezioni:

Prima Legge

First law of exception handling: There are only two ways a routine call may terminate: either the routine fulfils its contract, or it fails to fulfil it.

Seconda Legge

Second law of exception handling: A routine's failure must always cause an exception in the execution of the routine's caller

Terza Legge

Third law of exception handling: There are only two ways a routine may react as a result of an exception (that is to say, after a first strategy to fulfil its contract has not worked):

- Put back the objects in a stable state, and make a new attempt, using the same or another strategy (resumption.)
- Put back the objects in a stable state, give up on the contract, and report failure to the caller by triggering an exception (organized panic.)