Programmazione per contratto

---

pre-condizioni e post-condizioni

Una corretta definizione dell'interfaccia di una classe prevede la stesura di una sorta di contratto tra lo sviluppatore della classe e l'utilizzatore della classe. Per ogni funzionalità fornita, il contratto stabilisce quali sono le *pre-condizioni* che l'utilizzatore deve soddisfare per potere invocare la funzionalità e quali sono le *post-condizioni* che l'implementatore deve garantire in seguito all'esecuzione della funzionalità.

Tra le pre-condizioni e le post-condizioni sono sempre incluse le invarianti di classe sugli oggetti che sono acceduti (in lettura e/o scrittura) durante l'implementazione della funzionalità.

Il contratto si specifica in questo modo:

$$ \text{pre-condizioni } \to \text{ post-condizioni} $$

Si noti che se le pre-condizioni NON sono valide (cioè sono false), allora l'implicazione del contratto è vera a prescindere dalla validità o meno delle post-condizioni. Ovvero, se l'utente non soddisfa una pre-condizione l'implementatore non ha alcun obbligo. Spesso si preferisce rendere esplicite le condizioni sulle invarianti di classe, scrivendo il contratto nella forma: $$ \text{pre-condizioni AND invarianti } \to \text{ post-condizioni AND invarianti} $$ In questo caso, pre-condizioni e post-condizioni sono intese "al netto" delle invarianti di classe. Per esempio, nel caso dell'operatore di divisione tra oggetti Razionale:

```
Razionale operator/(const Razionale& x, const Razionale& y) {
  assert(x.check_inv() && y.check_inv()); // invarianti in ingresso
  assert(y != Razionale(0)); // pre-condizione "al netto" delle invarianti

  Razionale res = x;
  res /= y;

  // invarianti in uscita (omesso per x e y, perché costanti)
  assert(res.check_inv());
  return res;
}
```

La pre-condizione "al netto delle invarianti" dice che l'oggetto y deve essere diverso dal Razionale zero.

Analogamente, la post-condizione "al netto delle invarianti" richiede che il valore restituito sia effettivamente il risultato della divisione di x per y: i controlli sulle post-condizioni sono spesso difficili se non impossibili da automatizzare all'interno della classe e quindi si codificano direttamente nei test, indicando i risultati attesi.

Torna all'indice

Contratti narrow

Un operatore di divisione come specificato sopra è un esempio di "contratto narrow" (stretto): nei contratti narrow, l'implementatore si impegna a fornire la funzionalità solo quando ha senso farlo, cioè quando i valori forniti in input sono legittimi; l'onere di verificare tale legittimità è lasciato all'utilizzatore.

I contratti di tipo narrow sono molto comuni in C++, sia nel linguaggio in senso stretto (per esempio, quando si accede ad un elemento di un array, l'onere di controllare la validità dell'indice è a carico del programmatore), sia a livello di libreria standard (per esempio, spetta all'utente controllare che un `std::vector` non sia vuoto prima di eliminare l'ultimo elemento usando il metodo pop_back).

Torna all'indice

Contratti wide

Un caso ben diverso si verifica nel caso dei "contratti wide" (ampi), nei quali l'onere di verificare la legittimità delle invocazioni ricade sull'implementatore. Scegliere un contratto wide equivale quindi a spostare alcuni elementi del contratto dal lato della pre-condizione al lato della post-condizione.

Esempio di operatore di divisione con contratto wide:

```
Razionale operator/(const Razionale& x, const Razionale& y) {
 assert(x.check_inv() && y.check_inv()); // invarianti in ingresso

 // il contratto wide prevede, come post-condizione, che nel caso
 // y sia zero venga lanciata una opportuna eccezione
 if (y == Razionale(0))
  throw DivByZero();

 Razionale res = x;
 res /= y;

 // invarianti in uscita (omesso per x e y, perché costanti)
 assert(res.check_inv());
 return res;
}
```

Notare come, in caso di contratto wide, il controllo che y sia diverso da zero deve essere esplicito (NON può essere implementato come asserzione, perché deve essere presente anche nel codice compilato in modalità di produzione); inoltre, anche se la condizione viene controllata all'inizio dell'implementazione dell'operatore, si tratta di una *post* condizione, perché in questo caso l'utente può legittimamente pretendere di ottenere l'eccezione DivByZero quando y è uguale a zero.

I contratti wide sono quindi più onerosi (sia in termini di efficienza che in termini di codice da scrivere) per l'implementatore della classe.

Torna all'indice

I contratti per il linguaggio C++ e la libreria standard

Ogni volta che lo standard descrive una funzionalità del linguaggio o della libreria standard, ne viene descritto (a volte implicitamente) il contratto, in termini di pre-condizioni e post-condizioni. In particolare, sono classificati i comportamenti (behaviors) che una specifica implementazione del linguaggio è tenuta a rispettare, distinguendo tra le seguenti categorie:

1. Comportamento specificato (specified behavior)

   Il comportamento è descritto dallo standard; ogni implementazione è tenuta a conformarsi alla prescrizione, come specificata.

2. Comportamento definito dall'implementazione (implementation-defined)

   Ogni implementazione può scegliere come realizzare una determinata funzionalità, con l'obbligo di documentare la scelta fatta; per esempio, la scelta della dimensione di ognuno dei tipi interi standard, oppure la scelta se il tipo "char" (plain, cioè senza l'uso di signed o unsigned) abbia o meno il segno.

3. Comportamento non specificato (unspecified behavior)

   Comportamento che dipende dall'implementazione, che però non è tenuta a documentare la scelta fatta (e nemmeno a rifare la stessa scelta in contesti diversi); ad esempio, l'ordine di valutazione delle sottoespressioni è non specificato (caso particolare, l'ordine di valutazione degli argomenti in una chiamata di funzione).

4. Comportamento non definito (undefined behavior)

   Comportamento ottenuto a causa della violazione di una pre-condizione, in seguito alla quale l'implementazione non ha più nessuna prescrizione da seguire e quindi potrebbe comportarsi in modo totalmente arbitrario; esempi: indicizzazione di un array al di fuori dei limiti; tentativo di scrivere su di un oggetto definito const; overflow sui tipi interi con segno, ecc. Viene spesso indicato con la sigla UB e, per sottolinearne la totale arbitrarietà descritto,

colloquialmente, con la possibilità "to make demons fly out of your nose".

Esiste anche il locale-specific behavior, che dipende da convenzioni, cultura o linguaggio; per esempio, il valore dei caratteri del cosiddetto "execution character set".

Torna all'indice