Uso di Asserzioni alla Hoare nei Linguaggi O.O.
Java Modeling Language

Gennaio 2002

NB: tutti gli esempi in JML **NON** sono stati testati quindi non garantisco sulla sintassi!

Scopo: descrivere in modo astratto le proprietà dei moduli (classi e interfacce nel caso di Java)

Principi generali: Meyer, Object-Oriented Software Construction, Prentice Hall, Seconda edizione, 1997

“Design by Contract”: cioè la relazione tra un modulo e i suoi clienti è un contratto con diritti e doveri da entrambe le parti

Precondizioni e postcondizioni associate a metodi

associare la precondizione P e la postcondizione R ad un metodo m significa che richiediamo che valga

$$\{P\}\ body_m\ \{R\}$$

associare la precondizione P e la postcondizione R ad un costruttore C significa che richiediamo che valga

$$\{default_C \wedge P\}\ body_C\ \{R\}$$

dove $default_C$ è l'asserzione che dice che ogni campo ha i valori di default (ed ogni campo di classe ha il valore con cui è stato inizializzato)

NB: per semplicità trascuriamo la possibilità di inizializzazione diretta dei campi

Esempio

```
class StackByArray extends Stack {
  final static int maxLength = 15;
  int[] elems;
  int length;
/*@ requires !isEmpty() (oppure length>0)
  @ ensures   \result == elems[\old(length-1)])
  @ && length == \old(length - 1)
  @ oppure length < maxLength
  @*/
  int pop () throws EmptyStackException {
    if (length == 0) throw new EmptyStackException();
    return elems[--length];
  }
/*@ requires length < maxLength
  @ ensures elems[length-1] = e && length == \old(length + 1)
  @ oppure !isEmpty() (oppure length>0)
  @*/
  void push (int e) throws FullStackException {
    if (length == maxLength) throw new FullStackException();
    elems[length++]=e;
  }
```

```
/*@ requires true si omette
  @ ensures   \result  == length==0
  @ pure @*/ final boolean isEmpty(){
    return length ==0;
  }

//@ ensures   isEmpty() && elems != null
  StackByArray () {
    elems = new int[maxLength];
  }
}
```

Nel contratto:

precondizione = obbligo per il cliente

postcondizione = obbligo per la classe

nel caso in cui la precondizione non sia soddisfatta il metodo può in principio “fare qualunque cosa”; meglio se prevede un’eccezione

Quindi:

- la violazione di una precondizione segnala un errore nel cliente
- la violazione di una postcondizione segnala un errore nella classe

Sintassi:

la precondizione è un'espressione booleana relativa allo stato iniiziale dell'oggetto: variabili di istanza e di classe, metodi "puri" come `isEmpty()` + i parametri

NB: per i costruttori la precondizione è relativa solo agli argomenti

la postcondizione è un'espressione booleana relativa allo stato finale dell'oggetto + lo stato iniziale dell'oggetto (usando `\old`) + i parametri + `\result` (se c'e' tipo di ritorno)

`\old`(e) = valore dell'espressione booleana e relativo allo stato iniziale

`pure` = nessun side effect sullo stato

Invarianti di classe

```
class StackByArray extends Stack {
/*@ invariant  length >=0 && length <= maxLength
  @ && elems != null
  @*/
final static int maxLength = 15;
  int[] elems = new int[maxLength];
  int length = 0;
  }
```

Le invarianti di classe corrispondono intuitivamente a definire gli stati “corretti”

la classe implementa un tipo di dato attraverso una “funzione di astrazione” Φ, nell’esempio

Φ: `StackByArray` $\rightarrow \mathbb{Z}^{\star}$

L’invariante determina gli stati su cui Φ è definita

Le invarianti di classe sono espressioni relative allo stato dell'oggetto (anche usando metodi "puri")

devono valere in ogni stato *osservabile* cioè
- dopo la creazione (chiamata di un costruttore)
- dopo la chiamata di ogni metodo *non privato*

NON devono valere "continuamente"

vengono aggiunte implicitamente alla precondizione e postcondizione di ogni metodo

cioè se ho l'invariante *INV* in C questa vale se valgono le seguenti
- $\{P \wedge \mathit{INV}\}\ \mathit{body}_m\ \{R \wedge \mathit{INV}\}$ per ogni metodo m
- $\{P \wedge \mathit{default}_C\}\ \mathit{body}_C\ \{R \wedge \mathit{INV}\}$ per ogni C costruttore

NB: quindi le invarianti potrebbero sempre essere eliminate, ma è meglio utilizzarle per il loro significato intuitivo

inoltre esprimono proprietà che devono valere anche nelle classi eredi, quindi anche per i metodi che verranno aggiunti successivamente $\Rightarrow$ controllo su inheritance

Invarianti e metafora del contratto: specificare *INV* significa che per ogni metodo la specifica diventa

$\{P \wedge \mathit{INV}\}\ \mathit{body}_m\ \{R \wedge \mathit{INV}\}$

- aggiungere *INV* alla precondizione significa una restrizione maggiore per il cliente

- aggiungere *INV* alla postcondizione significa un requisito aggiuntivo da realizzare per il body

Invarianti e costruttori

Un’invariante di classe esprime proprietà che le istanze devono soddisfare in ogni stato stabile

se gli stati ottenuti con i valori di default non la soddisfano occorre prevedere opportuni costruttori

quindi: ruolo dei costruttori è anche garantire che l’invariante di classe sia soddisfatta

```
class Rectangle {
//@invariant length >= 0 && width >= 0
int length , width;
//@ requires l >= 0 && w >= 0
  Rectangle (int l, int w) { ... }
/*@ requires l >= 0
  @ ensures length = l && width = \old(width)
  @*/
  setLength (int l) {
    length = l;
  }
/*@ requires l >= 0
  @ ensures width = w && length = \old(length)
  @*/
  void setWidth (int w) {
    width = w;
  }
/*@  ensures \result == length * width
  @ pure *@/ int area () {
    return width * length;
  }
}
```

Problema: poichè le asserzioni sono definite sullo stato, spesso risultano legate alla particolare implementazione

ad esempio non potrei scriverle in un'interfaccia dove compaiono solo metodi

per avere una specifica "più astratta" si possono aggiungere dei metodi puri (osservazioni) in più utilizzati solo a scopo di specifica: si usa la parola chiave `model`

```
interface Stack {
/*@ invariant length() >= 0 &&
  @ (\forall int i ; 0 <= 1 && i<= length(); getElem(i) != null)
  @ model int getElem(int i) NB:restituisce i-esimo elemento (1=top)
  @ model int length()
  @
  @ requires length() > 0
  @ ensures \result == \old(getElem(1)) &&
  @ length() = \old(length()) - 1
  @ &&(\forall int i; 1 <= i && i <= length();
  @ getElem(i)=\old(getElem(i+1))
  @/*
  int pop () throws EmptyStackException;
/*@ ensures getElem(1) = e && length() = \old(length()) + 1
  @ &&(\forall int i ; 1 < i && i <= length();
  @ getElem(i) =\old(getElem(i-1))
  @/*
  void push (int e);
/*@ ensures \result == (length() == 0)
  @ pure @/* boolean isEmpty();
}
```

Si possono utilizzare campi e metodi `model` anche a valori in tipi non Java ma JML (libreria di “tipi puri”):

```
//@ model import edu.iastate.cs.jml.models.*;
interface Stack {
/*@ model JMLObjectSequence abs;
  @ initially abs != null and abs.isEmpty()
  @ invariant abs != null
  @
  @ requires !isEmpty()
  @ ensures \result == abs.first() &&
  @ abs.equals(\old(abs.trailer()))
  @/*
  int pop () throws EmptyStackException;

//@ ensures abs.equals(\old(abs.insertFront(e)))
  void push (int e);
/*@ ensures \result ==abs.isEmpty()
  @ pure @/* boolean isEmpty();
}
```

Nell’esempio la funzione di osservazione è esattamente la funzione di astrazione

Poi in una particolare implementazione andrò a definire come sono fatte le funzioni di osservazione e controllerò se le asserzioni sono verificate

Per i campi si usa clausola `represents`:

```
represents model field <- expression
```

Per i metodi: da verificare sul manuale