Scritto del corso di Metodologie di Programmazione

canale P-Z 24 Giugno 2010

Nome e Cognome_____

- ☐ Secondo esonero (solo esercizi 3, 4 e 5; tempo 1 ora e 30')
- ☐ Scritto completo (tutti gli esercizi; tempo 2 ore)

Esercizio 1 (5 punti). Si consideri il seguente metodo:

```
int fib (int n) {
      if (n < 2) return 1;
      return fib(n-1) + fib(n-2);
}
```

Si fornisca una possibile specifica per il metodo in questione. Si scriva inoltre un metodo per il calcolo dell'n-esimo numero di Fibonacci che soddisfi la seguente specifica:

```
throws:   NegativeInputArgument
returns:  l'n-esimo numero di Fibonacci
```

Esercizio 2 (10 punti). Si consideri il seguente programma:

```
int raddoppia (int n) {//PREC: n >= 0, POSTC: restituisce 2n
      int i = 0, j = 0;
      while (i < n) {//INV: ????
            i++;
            j += 2;
      }
      return j;
}
```

Si dimostri la correttezza del metodo, fornendo un'opportuna invariante di ciclo (che descriva i valori assunti da i e j all'inizio della k-esima iterazione del while) e dimostrando per induzione sul numero di iterazioni del while che tale invariante è verificata ad ogni iterazione. Da ciò si concluda che il valore finale di j (cioè, il valore restituito dalla funzione) è 2n.

Esercizio 3 (6 punti). Si supponga di avere una classe Java che descrive una struttura dati generica `OrderedCollection<A>` che permette di inserire/rimuovere oggetti in coda, in testa, prima o dopo un oggetto dato. La classe `OrderedCollection<A>` implementa la seguente interfaccia:

```
interface OrderedCollectionSpec<A>{
      A first() trhows EmptyCollectionException;
      A last() trhows EmptyCollectionException;
      void addFirst(A x);
      void addLast(A x);
      void addBefore (A x) trhows NoSuchElementException;
      void addAfter (A x) trhows NoSuchElementException;
      A removeFirst()trhows EmptyCollectionException;
      A removeLast() trhows EmptyCollectionException;
      boolean isEmpty();
}
```

Usare la classe `OrderedCollection<A>` per definire le strutture dati `Stack<A>` e `Queue<A>` che implementano rispettivamente le interfacce `StackSpec<A>` e `QueueSpec<A>` sotto descritte.

Rispondere inoltre alla seguente domanda, motivando la risposta: è opportuno che le classi `Stack<A>` e `Queue<A>` siano sottoclassi di `OrderedCollection<A>`?

```
interface StackSpec<A>{
      void push(A x);
      A pop(A x) throws EmptyStackException;
      A top() throws EmptyStackException;
      boolean isEmpty();
}
interface QueueSpec<A>{
      void enqueue(A x);
      A dequeue() throws EmptyQueueException;
      A first()throws EmptyQueueException;
}
```

Esercizio 4 (6 punti). Si supponga di avere le seguenti creazioni di oggetti e invocazioni di metodi, con il relativo output prodotto. Scrivere il codice JAVA delle classi `A`, `B` e `C`.
[**Nota**: ci sono molte soluzioni, ma saranno apprezzate quelle con il minor numero di definizioni di metodo]

Sequenza di istruzioni:	Output
`public class TestABC{`	
`public static void main(String args[]){`	
`A aa = new A();`	`sono nato A e sono 1`
`A ab = new B(2);`	`A sono nato e 2 sono` `sono nato B e sono 2`
`B bb = new B(5);`	`A sono nato e 5 sono` `sono nato B e sono 5`

`B bc = new C(11);`	`A sono nato e 11 sono`
	`sono nato B e sono 11`
	`sono nato C e sono 11`

`aa.m();`	`1`
`ab.m();`	`4`
`bb.m();`	`10`
`bc.m();`	`22`
`aa.mm(aa);`	`2`
`aa.mm(ab);`	`3`
`aa.mm(bb);`	`6`
`aa.mm(bc);`	`12`
`bb.mm(aa);`	`6`
`bb.mm(ab);`	`7`
`bb.mm(bb);`	`20`
`bb.mm(bc);`	`32`
`bc.mm(aa);`	`12`
`bc.mm(ab);`	`13`
`bc.mm(bb);`	`48`
`bc.mm(bc);`	`66`
`}`	

Esercizio 5 (**6 punti**). Siano date due classi che implementano gli alberi binari, `EmptyTree` e `NonEmptyTree`, di cui sotto diamo definizione delle variabili di istanza e costruttori. Scrivere un metodo (e la inner class che implementa l'interfaccia `Iterator`)

```
public Iterator preOrder()
```

che genera un iteratore che restituisce gli elementi dell'albero nella sequenza corripondente alla visita preorder (cioè prima radice, poi gli elementi del sottoalbero sinistro e poi gli elementi del sottoalbero destro).
[**Suggerimento**: nel momento della creazione dell'iteratore, caricare gli elementi dell'albero in una struttura dati lineare in un ordine opportuno.]

```
interface BinTree {
  int depth();
  int howManyNodes();
  boolean isEmpty();
  BinTree search(Object o);
  boolean balanced();
  boolean Nbalanced();
}
class NonEmptyTree implements BinTree{
  private Object info;
  private BinTree left;
  private BinTree right;
public NonEmptyTree(Object info)
  { this.info = info;
    left = new NonEmptyTree();
    right = new NonEmptyTree;
  }
public NonEmptyTree(Object info, BinTree left, BinTree right)
  { this.info = info;
    this.left = left;
    this.right = right;
   }
      …
}
```

```
class EmptyTree implements BinTree{
      …
      }
```

SOLUZIONI:

Es. 1: Una specifica è la seguente:

```
requires: n >= 0
returns:  l'n-esimo numero di Fibonacci
```

Un'implementazione che soddisfa la specifica data dal testo è invece la seguente:

```
class NegativeInputArgument extends Exception {}

int fib throws NegativeInputArgument (int n) {
      if (n < 0) throw new NegativeInputArgument();
      if (n < 2) return 1;
      return fib(n-1) + fib(n-2);
}
```

Es. 2: L'invariante è la seguente:

```
INV: i = k-1 AND j = 2(k-1)
```

Dimostriamo che, all'inizio della k-esima iterazione del while, l'invariante vale. Procediamo per induzione su k.

- *Base* (k = 1): vera, essendo i e j inizializzati a 0
- *Induzione* (vera per k, da dimostrare per k+1): Per ipotesi induttiva, all'inizio della k-esima iterazione, i = k-1 e j = 2(k-1). Nel corso della k-esima iterazione, i viene incrementato di 1 e j di 2; quindi, al termine della k-esima iterazione (e quindi all'inizio della (k+1)-esima), i = k e j = 2k, come voluto.

Infine, basta osservare che, per quanto abbiamo appena mostrato, al termine dell'iterazione n-esima (quando il ciclo termina) j = 2n, e quindi la correttezza del metodo.

Es. 3: Non era opportuno definire classi `Stack<A>` e `Queue<A>` come sottoclassi di `OrderedCollection<A>` in quanto così facendo si ereditano metodi che non fanno parte dell'interfaccia di `Stack<A>` e `Queue<A>`. Era corretto che tali classi avessero al loro interno una variabile di istanza di tipo `OrderedCollection<A>` e la usassero per implementare i loro metodi. Ecco una possibile soluzione:

```
class Stack<A> implements StackSpec<A>{
      private OrderedCollection<A> o;
      public Stack<A>(){o=new OrderedCollection<A>();}
      public void push(A x){o.addFirst(x);}
      public A pop(A x) throws EmptyStackException{
            try { return o.removeFirst();
                  } catch (EmptyCollectionException e){
                              throws new EmptyStackException();}
         }
      public A top() throws EmptyStackException{
            try { return o.first();
                  } catch (EmptyCollectionException e){
                              throws new EmptyStackException();}
        }
      public boolean isEmpty(){return o.first}
}


class Queue<A> implements QueueSpec<A>{
      public void enqueue(A x) {o.addLast(x);}
      A dequeue() throws EmptyQueueException{
            try { return o.removeFirst();
                  } catch (EmptyCollectionException e){
                              throws new EmptyQueueException();}
         }
```

```
      A first() throws EmptyQueueException{
            try { return o.first();
                  } catch (EmptyCollectionException e){
                              throws new EmptyQueueException();}
        }
}
```

Es. 4: C'erano molte soluzioni, compresa quella banale di definire una versione dei metodi `m()` e `mm()` per ogni possibile coppia di tipi che sono presenti nei test presentati.
Ecco una possibile soluzione:

```
class A{
      protected int x;
      public A() {x=1; print("sono nato A e sono "+x);}
      public A(int y) {x=y; print("A sono nato e "+x+" sono");}
      protected void print(String s){System.out.println(s);}
      public void m() {print(""+x);}
      public void mm(A a) {print(""+(x+a.x));}
}

class B extends A{
      public B(int x){super(x); print("sono nato B e sono "+x);}
      public void m() {print(""+2*x);}
      public void mm(B a) {print(""+(2*(x+a.x)));}
}

class C extends B{
      public C(int x){super(x); print("sono nato C e sono "+x);}
      public void mm(B a) {print(""+(3*(x+a.x)));}
}
```

Es. 5: Era necessario definire una inner class dentro la classe `NonEmptyTree`. Il costruttore di tale inner class memorizza in una struttura lineare gli elementi dell'albero nel corretto ordine, dopo di che vanno implementati i metodi `hasNext()` e `next()`. Qui viene usata la struttura dati `Vector`, ma non padronneggiandola, era sufficiente per esempio usare le `OrderedCollection` dell'esercizio 3.
Ho per la verità scoperto che l'implementazione dell'interfaccia `Iterator` richiede anche l'implementazione del metodo `remove()`.

```
public  Iterator preOrder() {return new GenPreorder(this);}

private static class GenPreorder implements Iterator{
      private Vector p;
      int i = 0;
      GenPreorder(BinTree b){
            p = new Vector();
            if (!b.isEmpty()) preOrderAux(b, p);
            }
      private void preOrderAux(BinTree b, Vector p){
            if (b.isEmpty()) return;
            p.add(((NonEmptyTree) b).info);
            preOrderAux(((NonEmptyTree) b).left, p);
            preOrderAux(((NonEmptyTree) b).right, p);
            }
      public boolean hasNext(){ return i<p.size();}
      public Object next() throws NoSuchElementException {
              if (i>=p.size()) throw new NoSuchElementException();
              return p.get(i++);
            }
      public void remove(){}
      }
```