Programmazione a Oggetti

Astrazione sui Dati e Iteratori

Sommario

Astrazione sui dati

- *Specifica di classi*
- *Invariante di rappresentazione*
- *Funzione di astrazione*

Iteratori

- *inner classes*
- *astrazione sul controllo*

Astrazione sui dati: specifica

```
class C {
//OVERVIEW: description of the
//behaviour of the type's object

//costruttori e relativa specifica

//metodi e relativa specifica
}
```

Insiemi di Interi: specifica

```
public class IntSet{
//OVERVIEW: Intset sono mutable,
    illimitati insiemi di interi {x1,
    x2, … xn}
public IntSet()
//POST: inizializza this all'insieme
    vuoto
public void insert(int x)
//MODIFICA:this
//POST: inserisce x in this
//this_post = this + {x}
```

Insiemi di Interi: specifica

```
public void remove(int x){
//MODIFICA: this
//POST: rimuove x da this
//this_post = this - {x}
public boolean isIn(int x)
//POST: ritorna true se x è in this
public int size()
//POST:ritorna la cardinalità di this
public int choose() throws
    EmptyException
//POST:torna un elemento arbitrario
}
```

Osservazioni

Le specifiche sono scritte in un ***linguaggio informale****, che fa riferimento a un patrimonio comune di* ***concetti "ben noti".***

Dichiarano:

modifiche *allo stato dell'oggetto ricevente (osservare la notazione per distinguere i valori prima e dopo l'esecuzione di un metodo)*

eventuali precondizioni *di ciascun metodo*

postcondizioni *ed eventuali* ***eccezioni sollevate***

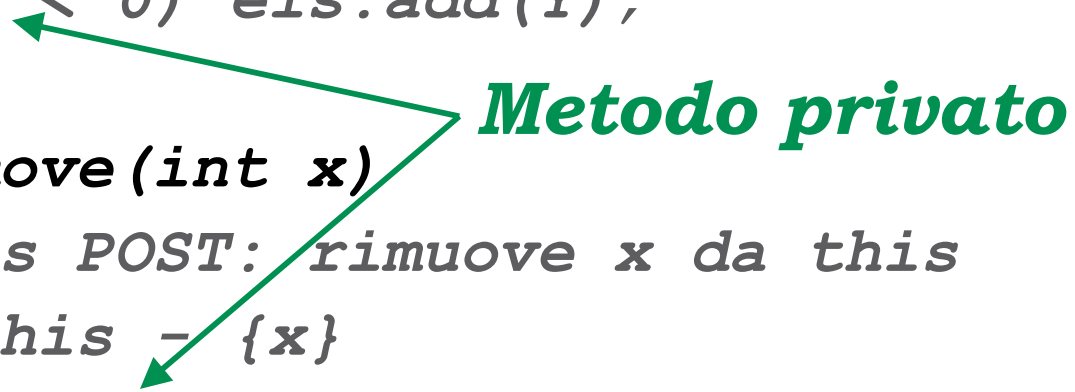
IntSet: implementazione

```
public class IntSet{
//OVERVIEW: Intset sono mutable,
    illimitati insiemi di interi
private Vector els;
// constructors
public IntSet(){
   //POST: inizializza this all'insieme
    vuoto
   els=new Vector();
}
```

Vector: classe di libreria per vettori estensibili: osservare che può memorizzare oggetti Object

IntSet: implementazione

```
public void insert(int x){
//MODIFICA: this. POST: inserisce x in this
//this_post = this + {x}
Integer y = new Integer(x);
if (getIndex(y) < 0) els.add(Y);
}
public void remove(int x)
//MODIFICA: this POST: rimuove x da this
//this_post = this - {x}
int i = getIndex(new Integer(x));
if (i<0) return;
els.set(i, els.lastElement());
els.remove(els.size()-1); }
```



IntSet: implementazione

```
private int getIndex(Integer x){
//POST: se x è in this, ritorna la sua
    posizione in els, altrimenti torna -1
for (int i=0; i<els.size(); i++)
    if (x.equals(els.get(i))) return i;
return -1;
}
```

Sarebbe interessante implementarlo ***facendo uso di eccezioni****, ma trattandosi di un metodo privato…*

IntSet: implementazione

```
public boolean isIn(int x){
/* POST: torna true se x è in this, altrimenti false */
return getIndex(new Integer(x))>=0;
}

public int size(){
/* POST: torna la cardinalità di this */
return els.size();}

public int choose() throws EmptyException{
/* POST: se this è vuoto solleva EmptyException,
 altrimenti sceglie un elemento arbitrario */
 if (els.size() == 0) throw new EmptyException();
    else return els.lastElement(); }
```

getIndex con Eccezioni

```
private int getIndex(Integer x)
    throws NoSuchElementException{
for (int i=0; i<els.size(); i++)
    if (x.equals(els.get(i))) return i;
throw NoSuchElementException;
}
```

chi lo invoca deve comportarsi di conseguenza:

```
public void insert(int x){
Integer y = new Integer(x);
try {getIndex(y);}
    catch (NoSuchElementExcetption)
    {els.add(y);}
}
```

Tassonomia dei "metodi"

creatori *generano nuovi oggetti "from scratch" (sono sostanzialmenti i costruttori)*

produttori *generano nuovi oggetti partendo da oggetti esistenti*

mutatori *modificano lo stato dell'oggetto ricevente*

osservatori *danno informazione sullo stato dell'oggetto ricevente*

Abstraction Function

*Scegliendo la rappresentazione, un implementatore ha in mente una relazione (**codifica**) dei dati astratti nei termini della rappresentazione.*

Nel caso degli insiemi di interi, i vettori `[1,3]` *e* `[3,1]` *entrambi rappresentano l'insieme* {1,3}

*In questo caso la abstraction function è non iniettiva (**molti-a-uno**)*

Representation Invariant

La rappresentazione dei dati soddisfa ***una proprietà invariante*** *che l'implementazione dei metodi si impegna a mantenere e che è* ***precondizione implicita*** *di tutti i metodi*

Esempi:

un insieme è rappresentato da un vettore ***senza ripetizioni:***

```
for all i, j. i!=j => els[i]!=els[j]
```

un insieme è rappresentato da un vettore ***ordinato senza ripetizioni***

```
for all i<j els[i]<els[j]
```

Representation Invariant

La funzione:

```
public boolean repOK()
// POST: ritorna true se this
  soddisfa REP INV
```

può essere esplicitamente ***implementata*** *e invocata, ad esempio da programmi tester*

dovrebbe essere invocata da ogni metodo mutatore*, ma la cosa potrebbe essere troppo inefficiente (le chiamate vengono rimosse dopo il testing)*

Metodi ereditati da `Object`

Tutte le classi sono implicitamente sottoclassi di **`Object`**.

In particolare, vengono ereditati i metodi:

`boolean equals(Object o)`
l'implementazione di default testa l'identità tra oggetti

`Object clone()` *crea una copia dell'oggetto, ma di default crea* ***sharing*** *delle instance variable*

`String toString()` *l'implementazione di default produce una stringa tipo:*
`"Type name: hash code"`

equals

Per fare ***overriding*** *del metodo standard di* **`Object`** *e per* ***ragioni di efficienza*** *(la reflection è costosa) è opportuno ridefinire il metodo come segue:*

```
boolean equals(Object o){
  if (!(o istanceof IntSet))
        return false;
  return equals((IntSet) o);
}
boolean equals(IntSet s){… }
```

clone()

L'implementazione standard di **`clone()`** *è corretta per i tipi immutabili.* ***Va ridefinita per i mutabili.***

Osservate in questo caso, l'uso di un ***costruttore privato****:*

```
private IntSet(Vector v){
   els=v;}

public Object clone(){
  return new IntSet((Vector)
   els.clone());}
```

Somma elementi insiemi

```
public static int setSum(intSet s)
    throws NullPointerException{
int[] a = new int[s.size()];
int sum=0;
for (int i=0; i<a.length(); i++){
    a[i]=s.choose();
    sum += a[i];
    s.remove(a[i]);
}
for (int i=0; i <a.length(); i++){
    s.insert(a[i]);
return sum;
}
```

Dispendioso!!

Alternative

Arricchire l'interfaccia degli insiemi con una funzione:

```
public int[] members()
/* POST: torna un array con gli elementi in
   this */
```

Tornare un puntatore al `Vector` che memorizza gli elementi:

```
public Vector members()
/* POST: torna un riferimento a els */
{ return els;}
```

Pericoloso: espone rep!

Iteratori

Idea: *fornire un* ***meccanismo astratto*** *per enumerare gli elementi di una struttura dati,* ***senza conoscerne la rappresentazione***

```
for all elements of the set
do action
```

Esempio: sommare tutti gli elementi di un insieme

Interfaccia Iterator

```
public interface Iterator{
   public boolean hasNext();
   /* POST: torna true se ci sono altri
    elementi, else false */

public Object next()
      throws NoSuchElementException;
   /* MODIFICA: this  POST: torna il
    prossimo  elemento, rimuovendolo, se
    c'è altrimenti solleva
    un'eccezione*/
}
```

Uso di un oggetto Iterator

```
Iterator g = s.elements();

// loop controllato da hasNext()
while (g.hasNext()){
int x=((Integer)g.next()).intValue();
// use x }

// loop controllato da eccezioni:
try{
 while (true){
    int x=((Integer)g.next()).intValue();
// use x
}catch (NoSuchElementException e){}
```

Specifica di Iteratori

```
class intSet{
// as before

public iterator elements()
/* POST: torna un generatore che enumera gli
    elementi di this (come Integer)
   PREC: richiede che this non sia modificato
    mentre il generatore è in uso
*/

}
```

Esempio: iteratore per IntSet

```
class intSet{

public Iterator elements()
 {return new SetGen(this);}

private static class SetGen implements Iterator{
  private IntSet s;
  private int n;
  public SetGen(IntSet s){this.s=s; n=0;}
  public boolean hasNext(){
      return n==els.size();}
  public Object next() throws NoSuchElementException{
    if (n>s.els.size())
          throw new NoSuchElementException();
       else return s.els.get(n++);}
}
```

Implementazione di Iteratori

*Il comportamento di un iteratore viene definito attraverso una **classe privata** definita **dentro** (**inner class**) la classe che definisce la struttura dati*

*La classe privata **deve implementare l'interfaccia** `Iterator`*

*Il metodo che crea un iteratore, semplicemente **crea un nuovo oggetto di tale classe***

*Si possono definire **più iteratori** per la stessa struttura dati*

Inner class

Dentro una inner class ho accesso ai campi e metodi privati della classe contenente

Se una inner class è privata, non c'è modo di creare istanza fuori della classe contenente