Le gerarchie di tipi: implementazioni multiple e principio di sostituzione

Come si può utilizzare una gerarchia di tipi

- implementazioni multiple di un tipo
  - i sottotipi non aggiungono alcun comportamento nuovo
  - la classe che implementa il sottotipo implementa esattamente il comportamento definito dal supertipo
- il sottotipo estende il comportamento del suo supertipo
  - fornendo nuovi metodi
- dal punto di vista semantico, supertipo e sottotipo sono legati dal principio di sostituzione
  - che definisce esattamente il tipo di astrazione coinvolto nella definizione di gerarchie di tipo

Implementazioni multiple

- il tipo superiore della gerarchia
  - un'interfaccia
  - una classe astratta
- definisce una famiglia di tipi tale per cui
  - tutti i membri hanno esattamente gli stessi metodi e la stessa semantica
    - per esempio, ci potrebbero essere implementazioni sparse e dense dei polinomi, realizzate con sottoclassi
      - che forniscono l'implementazione di tutti i metodi astratti, in accordo con le specifiche del supertipo
      - in più hanno i costruttori
    - un programma potrebbe voler usare tutte e due le implementazioni
      - scegliendo ogni volta la più adeguata
  - gli oggetti dei sottotipi vengono dall'esterno tutti visti come oggetti dell'unico supertipo
  - dall'esterno si vedono solo i costruttori dei sottotipi

IntList

- il supertipo è una classe astratta
- usiamo i sottotipi per implementare i due casi della definizione ricorsiva
  - lista vuota
  - lista non vuota
  - la classe astratta ha alcuni metodi non astratti
    - comuni alle due sottoclassi
    - definiti in termini dei metodi astratti
  - la classe astratta non ha variabili di istanza e quindi nemmeno costruttori

Specifica del supertipo `IntList`

```
public abstract class IntList {
 // OVERVIEW: un IntList è una lista non modificabile di Integers.
 // Elemento tipico [x1,...,xn]
 public abstract Integer first () throws EmptyException;
  // EFFECTS: se this è vuoto solleva EmptyException, altrimenti
  // ritorna il primo elemento di this
 public abstract IntList rest () throws EmptyException;
  // EFFECTS: se this è vuoto solleva EmptyException, altrimenti
  // ritorna la lista ottenuta da this togliendo il primo elemento
 public abstract Iterator elements ();
 // EFFECTS: ritorna un generatore che produrrà tutti gli elementi di
 // this (come Integers) nell'ordine che hanno in this
 public abstract IntList addEl (Integer x);
  // EFFECTS: aggiunge x all'inizio di this
 public abstract int size ();
  // EFFECTS: ritorna il numero di elementi di this
 public abstract boolean repOk ();
 public String toString ();
 public boolean equals (IntList o);
}
```

Implementazione del supertipo `IntList`

```
public abstract class IntList {
 // OVERVIEW: un IntList è una lista non modificabile di Integers.
 // Elemento tipico [x1,...,xn]
 // metodi astratti
 public abstract Integer first () throws EmptyException;
 public abstract IntList rest () throws EmptyException;
 public abstract Iterator elements ();
 public abstract IntList addEl (Integer x);
 public abstract int size ();
 public abstract boolean repOk ();
 // metodi concreti
 public String toString () {....};
 public boolean equals (IntList o) {
  // confronta gli elementi usando elements
   }
}
```

- ✓ `toString` e `equals` sono implementate
  - utilizzando il generatore `elements`

Implementazione del sottotipo `EmptyIntList`

```
public class EmptyIntList extends IntList {
 public EmptyIntList () {}
 public Integer first () throws EmptyException
 { throw new EmptyException ("EmptyIntList.first"); }
 public IntList rest () throws EmptyException
 { throw new EmptyException ("EmptyIntList.rest"); }
 public Iterator elements () { return new EmptyGen(); }
 public IntList addEl (Integer x)
 {return new FullIntList(x);}
 public int size () {....}
 public boolean repOk () {....}
 static private class EmptyGen implements Iterator {
   EmptyGen () {}
   public boolean hasNext () { return false; }
   public Object next () throws NoSuchElementException {
        throw new NoSuchElementException("IntList.elements"); }      }

}
```

Implementazione del sottotipo `FullIntList`

```
public class FullIntList extends IntList {
 private int sz;
 private Integer val;
 private IntList next;
 public FullIntList (Integer x)
 {sz = 1; val = x; next = new EmptyIntList (  ); }
 public Integer first () {return val; }
 public IntList rest () { return next;  }
 public Iterator elements () { .... }
 public IntList addEl (Integer x)
 {FullIntList n = new FullIntList(x);
 n.next = this; n.sz = this.sz + 1; return n; }
 public int size () {....}
 public boolean repOk () {....}
}
```

Poly

- il supertipo è una classe astratta
- la specifica della classe astratta è quella solita
- usiamo i sottotipi per realizzare due diverse implementazioni
  - `DensePoly`
  - `SparsePoly`
  - la classe astratta ha alcuni metodi non astratti
    - comuni alle due sottoclassi
    - definiti in termini dei metodi astratti
  - la classe astratta ha variabili di istanza e quindi costruttori

Implementazione del supertipo `Poly`

```
public abstract class Poly {
 protected int deg; // il grado
 protected Poly (int n) {deg = n; }
 public abstract int coeff (int d);
 public abstract Poly add (Poly q) throws NullPointerException;
 public abstract Poly mul (Poly q) throws NullPointerException;
 public abstract Poly minus ();
 public abstract Iterator terms ();
 public abstract boolean repOk ();
 public int degree () {return deg;}
 public Poly sub (Poly p) {return add(p.minus());}
 public String toString (){...}
 public boolean equals (Poly p)
  {if (p == null || deg != p.deg) return false;
   Iterator tg = terms(); Iterator pg = p.terms();
   while (tg.hasNext())
     {int tx = ((Integer) tg.next( )).intValue(  );
     int px = ((Integer) pg.next( )).intValue(  );
     if (tx != px || coeff (tx) != p. coeff (px) return false); }
   return true; }
}
```

le sottoclassi di `Poly`

- nelle operazioni si decide quale delle due rappresentazioni conviene usare
  - l'add di `DensePoly` lascia a `SparsePoly` la gestione del caso in cui i due oggetti sono diversi
- possono essere necessarie conversioni fra le due rappresentazioni
  - ed altri problemi simili

Implementazione del sottotipo DensePoly 1

```
public class DensePoly extends Poly {
 private int[] trms; // coefficienti fino a deg
 public DensePoly () { super(0); trms = new int[1]; }
 public DensePoly (int c, int n) throws NegExpException {...    }
 private DensePoly (int n) { super (n); trms = new int [n+1] ; }
 ....
 public Poly add (Poly q) throws NullPointerException
 {if (q instanceof SparsePoly) return q. add (this) ;
  DensePoly la, sm;
  if (deg > q.deg) (la = this; sm = (DensePoly) q; }
    else { la = (DensePoly) q; sm = this; }
  int newdeg = la.deg;
  if (sm.deg == la.deg)
    for int k = sm.deg; k > 0; k--) if (sm.trms[k] + la.trms[k] != 0) break;
    else newdeg--;
    DensePoly r = new DensePoly(newdeg);
   int i;
   for (i = 0; i <= sm.deg && i <= newdeg; i++)
     r.trms[i] = sm.trms[i] + la.tms[i];
   for (int j = i; j <= newdeg; j ++) r. trms [j] = la. trms [j];
   return r; }
}
```

Principio di sostituzione

- un oggetto del sottotipo può essere sostituito ad un oggetto del supertipo senza influire sul comportamento dei programmi che utilizzano il tipo
  - i sottotipi supportano il comportamento del supertipo
  - per esempio, un programma scritto in termini del tipo `Poly` può lavorare correttamente su oggetti del tipo `DensePoly`
- il sottotipo deve soddisfare le specifiche del supertipo
- astrazione via specifica per una famiglia di tipi
  - astraiamo diversi sottotipi a quello che hanno in comune
    - la specifica del loro supertipo

Principio di sostituzione

- ✓ devono essere supportate
  - la regola della segnatura
    - gli oggetti del sottotipo devono avere tutti i metodi del supertipo
    - le segnature dei metodi del sottotipo devono essere compatibili con le segnature dei corrispondenti metodi del supertipo
  - la regola dei metodi
    - le chiamate dei metodi del sottotipo devono comportarsi come le chiamate dei corrispondenti metodi del supertipo
  - la regola delle proprietà
    - il sottotipo deve preservare tutte le proprietà che possono essere provate sugli oggetti del supertipo
- ✓ tutte le regole riguardano solo le specifiche!

Regola della segnatura

- ✓ se una chiamata è type-correct per il supertipo lo è anche per il sottotipo
  - garantita dal compilatore Java
    - che permette solo che i metodi del sottotipo sollevino meno eccezioni di quelli del supertipo
    - potrebbe essere più liberale
      - il tipo ritornato dal metodo del sottotipo potrebbe essere un sottotipo
  - le altre due regole non possono essere garantite dal compilatore Java
    - hanno a che fare con la specifica della semantica! ...

Regola dei metodi 1

- ✓ si può ragionare sulle chiamate dei metodi usando la specifica del supertipo anche se viene eseguito il codice del sottotipo
- ✓ garantito che va bene se i metodi del sottotipo hanno esattamente le stesse specifiche di quelli del supertipo
- ✓ come possono essere diverse?
  - se la specifica nel supertipo è nondeterministica (comportamento sottospecificato) il sottotipo può avere una specifica più forte che risolve (in parte) il nondeterminismo

Regola dei metodi 2

- ✓ il metodo `elements` di `IntSet` non assume un ordine degli elementi dell'insieme

```
public class IntSet {
 public Iterator elements ()
 // EFFECTS: ritorna un generatore che produrrà tutti gli elementi di
 // this (come Integers) ciascuno una  sola volta, in ordine arbitrario
```

- ✓ il metodo **`elements`** di **`SortedIntSet`** assume l'ordine crescente

```
public class SortedIntSet {
 public Iterator elements ()
 // EFFECTS: ritorna un generatore che produrrà tutti gli elementi di
 // this (come Integers) ciascuno una  sola volta, in ordine crescente
```

Regola dei metodi 3

- ✓ in generale un sottotipo può indebolire le precondizioni e rafforzare le post condizioni
- ✓ per avere compatibilità tra le specifiche del supertipo e quelle del sottotipo devono essere soddisfatte le regole
  - regola delle precondizione
    - $pre_{super} \Longrightarrow pre_{sub}$
  - regola delle postcondizione
    - $pre_{super} \;\&\&\; post_{sub} \Longrightarrow post_{super}$

Regola dei metodi 4

- ✓ indebolire la precondizione
  - $pre_{super} \Longrightarrow pre_{sub}$

  ha senso, perché il codice che utilizza il metodo è scritto per usare il supertipo
  - ne verifica la precondizione
  - verifica anche la precondizione del metodo del sottotipo
- ✓ esempio: un metodo in `IntSet`

```
public void addZero (  )
// REQUIRES: this non è vuoto
// EFFECTS: aggiunge 0 a this
```

potrebbe essere ridefinito in un sottotipo

```
public void addZero (  )
// EFFECTS: aggiunge 0 a this
```

Regola dei metodi 5

- ✓ rafforzare la post condizione
  - pre $_{super}$ && post $_{sub}$ ==> post $_{super}$

  ha senso, perché il codice che utilizza il metodo è scritto per usare il supertipo
  - assume come effetti quelli specificati nel supertipo
  - gli effetti del metodo del sottotipo includono comunque quelli del supertipo (se la chiamata soddisfa la precondizione più forte)
- ✓ esempio: un metodo in `IntSet`

```
public void addZero (  )
// REQUIRES: this non è vuoto
// EFFECTS: aggiunge 0 a this
```

potrebbe essere ridefinito in un sottotipo

```
public void addZero (  )
// EFFECTS: se this non è vuoto aggiunge 0 a this altrimenti aggiunge 1 a this
```

Regola dei metodi 6

```
public class IntSet {
 public Iterator elements ()
 // EFFECTS: ritorna un generatore che produrrà tutti gli elementi di
 // this (come Integers) ciascuno una  sola volta, in ordine arbitrario

public class SortedIntSet extends IntSet {
 public Iterator elements ()
 // EFFECTS: ritorna un generatore che produrrà tutti gli elementi di
 // this (come Integers) ciascuno una  sola volta, in ordine crescente
```

- ✓ entrambi i metodi hanno precondizione `true`
- ✓ la postcondizione del metodo del sottotipo
  - gli elementi sono generati in ordine crescente

  implica la postcondizione del metodo del supertipo

Regola dei metodi: violazioni 1

- ✓ consideriamo `insert` in `IntSet`

```
public class IntSet {
 public void insert (int x)
  // EFFECTS: aggiunge x a this
```

- ✓ supponiamo di definire un sottotipo di `IntSet` con la seguente specifica di `insert`

```
public class StupidoIntSet extends IntSet {
public void insert (int x)
  // EFFECTS: aggiunge x a this se x è pari, altrimenti non fa nulla
```

Regola dei metodi: violazioni 2

- ✓ consideriamo `addEl` in `OrderedIntList`

```
public class OrderedIntList {
 public void addEl (int x) throws DuplicateException
  // EFFECTS: aggiunge x a this se non c'è già, altrimenti solleva
  // l'eccezione
```

- ✓ supponiamo di definire un sottotipo di `OrderedIntList` con la seguente specifica di `addEl`

```
public void addEl (int x)
  // EFFECTS: aggiunge x a this se non c'è già, altrimenti non fa nulla
```

- ✓ non è un problema la differenza della segnatura
- ✓ ma c'è un problema con la regola dei metodi
  - se l'elemento c'è già, i due metodi hanno comportamento diverso
  - e quello del sottotipo fa "meno cose"

Regola delle proprietà 1

- ✓ il ragionamento sulle proprietà degli oggetti basato sul supertipo è ancora valido quando gli oggetti appartengono al sottotipo
- ✓ proprietà degli oggetti
  - non proprietà dei metodi
- ✓ da dove vengono le proprietà degli oggetti?
  - dal modello del tipo di dato astratto
    - le proprietà degli insiemi matematici, etc.
    - le elenchiamo esplicitamente nell'overview del supertipo
      - un tipo astratto potrebbe avere un numero infinito di proprietà
- ✓ un esempio, proprietà invarianti
  - un `FatSet` non è mai vuoto

Regola delle proprietà 2

- ✓ per mostrare che un sottotipo soddisfa la regola delle proprietà dobbiamo mostrare che preserva le proprietà del supertipo
- ✓ per esempio, per le proprietà invarianti
  - bisogna provare che creatori e produttori del sottotipo stabiliscono l'invariante (solita induzione sul tipo)
  - che tutti i metodi (anche quelli nuovi, inclusi i costruttori) del sottotipo preservano l'invariante