AA 2016-2017

12. Gerarchie di tipi: implementazioni multiple e principio di sostituzione

Implementazioni multiple

- Il tipo superiore della gerarchia definisce una famiglia di tipi tale per cui
  - tutti i membri hanno esattamente gli stessi metodi e la stessa semantica che forniscono l'implementazione di tutti i metodi astratti, in accordo con le specifiche del supertipo
  - gli oggetti dei sottotipi vengono visti dall'esterno 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

supertipo IntList


```
public abstract class IntList {
    // OVERVIEW: un IntList e' una lista modificabile
    // di Integers.
    // Elemento tipico [x1,...,xn]
    public abstract Integer first( ) throws EmptyException;
        // EFFECTS: se this e' vuoto solleva EmptyException,
        // altrimenti ritorna il primo elemento di this
    public abstract IntList rest( ) throws EmptyException;
        // EFFECTS: se this e' vuoto solleva EmptyException,
        // altrimenti  ritorna la lista ottenuta da this
        // togliendo il primo elemento
    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 `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 IntList addEl (Integer x) {
    return new FullIntList(x);
  }
  public int size( ) {...}
  public boolean repOk( ) {...}
}
```

Implementazione `FullIntList`


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

Principio di sostituzione

- Un oggetto del sottotipo può essere sostituito da 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 IntList può lavorare correttamente su oggetti del tipo FullIntList
- 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
- NB: le regole riguardano la semantica!

Regola della segnatura

- Se una chiamata è type-correct per il supertipo lo è anche per il sottotipo
  - garantita dal compilatore Java
    - ✓che permette che i metodi del sottotipo sollevino meno eccezioni di quelli del supertipo
    - ✓da Java 5 un metodo della sottoclasse può sovrascrivere un metodo della superclasse con la stessa firma fornendo un return type più specifico.
      - docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.4.8.3
    - ✓le altre due regole non possono essere garantite dal compilatore Java…
    - ✓dato che hanno a che fare con la specifica della semantica!

Covariante vs. controvariante

- Una sottoclasse può riscrivere un metodo restituendo come risultato un valore di un sottotipo di quello previsto dal metodo della superclasse (covariant return type)
- Una nozione più liberale potrebbe avere argomenti contravarianti
  - un tipo è detto covariante se mantiene l'ordinamento dato dalla gerarchia dei tipi
  - contravariante se inverte l'ordinamento
  - invariante se non dipende dall'ordinamento

Classi vs. Tipi

- Una classe definisce il comportamento degli oggetti (istanza)
  - il meccanismo di ereditarietà può modificare i comportamenti riscrivendo i metodi
- Un tipo definisce i comportamenti in termini del tipo dei metodi
- Sono concetti differenti e devono essere usati in modo coerente
  - Java/C# confondono le due nozioni: il nome di una classe è il tipo degli oggetti
  - nella pratica questa confusione è utile ma si deve tenere a mente che sono due nozioni differenti

Regola dei metodi

- Si può ragionare sulle chiamate dei metodi usando la specifica del supertipo anche se viene eseguito il codice del sottotipo
- Si è garantiti 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 non-deterministica (comportamento sottospecificato) il sottotipo può avere una specifica più forte che risolve (in parte) il non-determinismo

Regola dei metodi

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

Regola dei metodi

- Ha senso indebolire la precondizione
  - pre $_{super}$ ==> pre $_{sub}$

  perché il codice che utilizza il metodo è scritto per usare il supertipo
  - ne verifica la pre-condizione
  - verifica anche la pre-condizione del metodo del sottotipo
- Esempio: un metodo in IntSet

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

potrebbe essere ridefinito in un sottotipo

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

Regola dei metodi

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

  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 pre-condizione più forte)
- Esempio: un metodo in IntSet

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

potrebbe essere ridefinito in un sottotipo

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

Regola dei metodi: violazioni

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

Regola delle proprietà

- Il ragionamento sulle proprietà degli oggetti basato sul supertipo è ancora valido quando gli oggetti appartengono al sottotipo
- Sono 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 può avere un numero infinito di proprietà
- Proprietà invarianti
  - un FatSet non è mai vuoto
- Proprietà di evoluzione
  - il numero di caratteri di una String non cambia

Regola delle proprietà

- Per mostrare che un sottotipo soddisfa la regola delle proprietà dobbiamo mostrare che preserva le proprietà del supertipo
- 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
- Per le proprietà di evoluzione
  - bisogna mostrare che ogni metodo del sottotipo le preserva

Regola delle proprietà: una proprietà invariante

- Il tipo `FatSet` è caratterizzato dalla proprietà che i suoi oggetti non sono mai vuoti

```
// OVERVIEW: un FatSet e' un insieme modificabile di interi
// la cui dimensione e' sempre almeno 1
```

- Assumiamo che `FatSet` non abbia un metodo `remove`, ma invece abbia un metodo `removeNonEmpty`

```
public void removeNonEmpty (int x)
    // EFFECTS: se x e' in this e this contiene altri elementi
    // rimuovi x da this
```

e abbia un costruttore che crea un insieme con almeno un elemento

Possiamo provare che gli oggetti `FatSet` hanno dimensione maggiore di 0?

Regola delle proprietà: una proprietà invariante

- Consideriamo il sottotipo ThinSet che ha tutti i metodi di FatSet con identiche specifiche e in aggiunta il metodo

```
public void remove(int x)
    // EFFECTS: rimuove x da this
```

- ThinSet non è un sottotipo legale di FatSet
  - perché il suo extra metodo può svuotare l'oggetto, e
  - l'invariante del supertipo non sarebbe conservato

Regola delle proprietà: una proprietà di evoluzione (non modificabilità)

- Il tipo SimpleSet ha i due soli metodi insert e isIn
  - gli oggetti di SimpleSet possono solo crescere in dimensione
  - IntSet non può essere un sottotipo di SimpleSet perché il metodo remove non conserva la proprietà