
PROGRAMMAZIONE 2

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

Implementazioni multiple

Il **tipo superiore** della gerarchia definisce una famiglia di tipi in 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 **super-tipo**
- gli oggetti dei sotto-tipi vengono visti dall'esterno come oggetti dell'unico super-tipo
- **dall'esterno si vedono solo i costruttori dei sotto-tipi**

IntList

- Il **super-tipo** è una **classe astratta**
- Usiamo i sotto-tipi per implementare i due casi della **definizione ricorsiva**
 - **lista vuota**
 - **lista non vuota**
- La **classe astratta** ha alcuni metodi **non astratti**
 - **comuni alle due sotto-classi**
 - **definiti in termini dei metodi astratti**
- La **classe astratta** non ha variabili di istanza e quindi nemmeno costruttori

Super-tipo **IntList**

```
public abstract class IntList {
    //OVERVIEW: un IntList e' una lista modificabile
    //di Integer. 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: ritorna la lista ottenuta da this
        //aggiungendo x all'inizio di this
    public abstract int size( );
        // EFFECTS: ritorna il numero di elementi di this
}
```

La classe astratta non ha variabili di istanza e quindi nemmeno costruttori !!!!

Super-tipo **IntList**

```
public abstract class IntList {  
    // OVERVIEW: un e' una lista modificabile  
    // di Integer. Elemento tipico [x1,...,xn]  
  
    public String toString( ) {...}  
    public boolean equals(IntList o) {...}  
}
```

- Metodo **equals** overloaded
- I metodi concreti si possono implementare scorrendo la lista con i **metodi astratti (first, rest)**
- Completare per esercizio

Implementazione **EmptyIntList**

```
public class EmptyIntList extends IntList{  
//OVERVIEW: un EmptyIntList e' una lista vuota di Integer.  
// Elemento tipico: []  
  
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( ) {return 0;}  
}
```

Implementazione **FullIntList**

```
public class FullIntList extends IntList {
    // OVERVIEW: un e' una lista modificabile non vuota
    // di Integer. Elemento tipico [x1,...,xn]

    private int dim;
    private Integer val;
    private IntList next;

    public FullIntList(Integer x) {
        dim = 1; val = x;
        next = new( );
    }
    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( ) {return dim;}
}
```

Esercizio: da completare

- Completare l'implementazione della classe astratta
- Dare invariante e funzione di astrazione per **FullIntList** e **EmptyIntList**
- Scrivere un metodo statico **sum** che calcola la somma dei valori di una lista (il metodo ha parametro di tipo **IntList**)
- Si scrive guardando la specifica del super-tipo **IntList**

Un metodo stand alone

```
public class ProvaIntList{  
  
    public static int sum(IntList l) {  
        try{  
            int first= ((Integer) l.first( )).intValue();  
        } catch (EmptyException e)  
        {return 0;}  
        return first + sum(l.rest());  
    }  
}
```

Alternativamente si potrebbe usare **instance of** per i due sottotipi

Principio di sostituzione

- Un oggetto del sotto-tipo può essere sostituito da un oggetto del super-tipo senza influire sul comportamento dei programmi che utilizzano il tipo
 - i sotto-tipi supportano il comportamento del super-tipo
 - per esempio, un programma scritto in termini del tipo **IntList** può lavorare correttamente su oggetti del tipo **FullIntList** e **EmptyIntList**
- In questo caso il sotto-tipo soddisfa le specifiche del super-tipo (sono le stesse specializzate ai due sottocasi)
- In generale???

Sotto-tipo

- **B è un sotto-tipo di A:** “every object that satisfies interface B also satisfies interface A”
- **Obiettivo metodologico:** il codice scritto guardando la specifica di A opera correttamente anche se viene usata la specifica di B

Sotto-tipi e Principio di Sostituzione

- **B è un sotto-tipo di A: B** può essere sostituito per **A**
 - una istanza del sotto-tipo **B** soddisfa le proprietà del super-tipo **A**
 - una istanza del sotto-tipo **B** può avere **maggiori vincoli** di quella del super-tipo **A**

Principio di sostituzione (1)

La regola della segnatura

- gli oggetti del **sotto-tipo** devono avere tutti i metodi del **super-tipo**
- le **segnature** dei metodi del **sotto-tipo** devono essere **compatibili** con le segnature dei corrispondenti metodi del **super-tipo**

Principio di sostituzione (2)

La regola dei metodi

- le chiamate dei metodi del **sotto-tipo** devono comportarsi come le chiamate dei corrispondenti metodi del **super-tipo** (metodi sovrascritti)

La regola delle proprietà

- il **sotto-tipo** deve preservare tutte le proprietà che possono essere provate sugli oggetti del **super-tipo**

La regola della segnatura

- Se una chiamata è **type-correct** per il **super-tipo** lo è anche per il **sotto-tipo**
 - **garantita dal compilatore Java**
 - che permette che i metodi del sotto-tipo sollevino meno eccezioni di quelli del super-tipo
 - da Java 5 un metodo della sotto-classe può sovrascrivere un metodo della super-classe 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 esser garantite dal compilatore Java...
 - **dato che hanno a che fare con la specifica della semantica!**

La regola dei metodi

- Si può ragionare sulle chiamate dei metodi usando la **specifica** del super-tipo anche se viene eseguito il codice del sotto-tipo
- **Si è garantiti che va bene se i metodi del sotto-tipo hanno esattamente le stesse specifiche di quelli del super-tipo**
- **Come possono essere diverse?**
 - se la specifica nel super-tipo è **non-deterministica** (comportamento sotto-specificato) il sotto-tipo può avere una **specifica più forte che risolve (in parte) il non-determinismo**

La regola dei metodi

- In generale un **sotto-tipo** può **indebolire le pre-condizioni** e **rafforzare le post-condizioni**
- Per avere compatibilità tra specifiche del super-tipo e del sotto-tipo devono essere soddisfatte le regole
 - regola delle **pre-condizione**
 - $pre_{super} \implies pre_{sub}$
 - regola delle **post-condizione**
 - $pre_{super} \ \&\& \ post_{sub} \implies post_{super}$

Regola dei metodi: pre-condizione

- Ha senso indebolire le pre-condizioni
 - $pre_{super} \implies pre_{sub}$
- Infatti il codice che utilizza il metodo è scritto per usare il **super-tipo**
 - ne verifica la pre-condizione pre_{super}
 - verifica anche la pre-condizione del sotto-tipo pre_{sub}

Esempio: pre-condizione

- un metodo di **IntSet**

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

- potrebbe essere ridefinito in un sotto-tipo

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

Regola dei metodi: post-condizioni

- Ha senso rafforzare le post-condizioni
 - $pre_{super} \ \&\& \ post_{sub} \ ==> \ post_{super}$
- Dato che il codice che utilizza il metodo è scritto per usare il **super-tipo**
 - assume come effetti quelli specificati nel **super-tipo**
 - gli effetti del metodo del **sotto-tipo** includono comunque quelli del **super-tipo** (se la chiamata soddisfa la **pre-condizione più forte**)

Esempio: post-condizione

- un metodo di **IntSet**

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

- potrebbe essere ridefinito in un sotto-tipo

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

Varrebbe senza la pre-super?

Regola dei metodi: violazioni

- Consideriamo `insert` di `IntSet`

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

- Supponiamo di definire un sotto-tipo di `IntSet` con la seguente specifica di `insert`

```
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 super-tipo è ancora valido quando gli oggetti appartengono al sotto-tipo
- 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 super-tipo
 - un tipo astratto può avere un numero infinito di proprietà
- Proprietà invarianti
- Proprietà di evoluzione

Regola delle proprietà

- Per mostrare che un **sotto-tipo** soddisfa la regola delle proprietà dobbiamo mostrare che preserva le proprietà del **super-tipo**
- Per **proprietà invarianti**
 - che tutti i metodi (anche quelli nuovi, inclusi i costruttori) del sotto-tipo preservano l'**invariante**
 - bisogna provare che creatori e produttori del sotto-tipo stabiliscono l'**invariante** (solita induzione sul tipo)
- Per le **proprietà di evoluzione**
 - bisogna mostrare che ogni metodo del sotto-tipo le preserva

Una proprietà invariante

- Il tipo **FatSet** è caratterizzato dalla proprietà che i suoi insiemi 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. Si può provare che gli oggetti **FatSet** hanno dimensione maggiore di 0?

Una proprietà invariante

- Consideriamo il sotto-tipo **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 sotto-tipo legale di **FatSet**
 - perché il suo extra metodo può svuotare l'insieme, e
 - La proprietà del super-tipo non sarebbe conservato

Una proprietà di evoluzione

- 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 sotto-tipo di **SimpleSet** perché il metodo **remove** non conserva la proprietà

Un esempio conclusivo

```
interface Rectangle {  
    // effects: thispost.width = w, thispost.height = h  
    void setSize(int w, int h);  
}
```

```
interface Square implements Rectangle { ... }
```

Quale è la scelta ottimale per la specifica di `setSize` per `Square`?

- `// requires: w = h`
`// effects: ... come sopra`
`void setSize(int w, int h);`
- `// effects: esattamente come sopra unico parametro`
`void setSize(int edgeLength);`
- `// effects: come sopra`
`// throws BadSizeException if w != h`
`void setSize(int w, int h) throws BadSizeException;`

Figure geometriche

- **Square** non è un sotto-tipo di **Rectangle**
 - gli oggetti di **Rectangle** hanno variabili di istanza (base e altezza) indipendenti
 - **Square** viola questa proprietà
- **Rectangle** non è sotto-tipo di **Square**
 - **Square** base e altezza sono identici
 - **Rectangle** viola questa proprietà
- La nozione di sotto-tipo non è sempre quella suggerita dall'intuizione
- Possibile soluzione
 - aggiungere **Shape**

