
PROGRAMMAZIONE 2

11. Le gerarchie di tipi

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 soddisfa le proprietà del super-tipo
 - una istanza del sotto-tipo può avere maggiori vincoli di quella del super-tipo
- Questo non è sempre vero in Java

-
- Quella di sotto-tipo è una nozione semantica (specifica delle proprietà)
 - B è un sotto-tipo di A se e solo se un oggetto di B si può mascherare come un oggetto di A in tutti i possibili contesti
 - L'ereditarietà è una nozione di implementazione
 - creare una nuova classe evidenziando solo le differenze (il codice nuovo)

Principio di sostituzione

- Un oggetto del sotto-tipo può essere sostituito a 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 `AbstractSet` può lavorare correttamente su oggetti del tipo `TreeSet`
- Il sotto-tipo deve soddisfare le specifiche del super-tipo
- Astrazione via specifica per una famiglia di tipi
 - astraiano diversi sotto-tipi a quello che hanno in comune

Gerarchia di tipi: specifica

- Specifica del tipo superiore della gerarchia
 - come quelle che già conosciamo
 - l'unica differenza è che può essere parziale, e per esempio possono mancare i costruttori
- Specifica di un sotto-tipo
 - la specifica di un sotto-tipo è data relativamente a quella dei suoi super-tipi
 - non si definiscono nuovamente quelle parti delle specifiche del super-tipo che non cambiano
 - vanno specificati
 - i costruttori del sotto-tipo
 - i metodi “nuovi” forniti dal sotto-tipo
 - i metodi del super-tipo che il sotto-tipo ridefinisce

Gerarchia di tipi: implementazione

- Implementazione del super-tipo
 - può non essere implementato affatto
 - può avere implementazioni parziali, ovvero alcuni metodi sono implementati e altri no
 - può fornire informazioni a potenziali sotto-tipi dando accesso a variabili o metodi di istanza
 - ✓ che un “normale” utente del super-tipo non può vedere
- I sotto-tipi sono implementati come estensioni dell’implementazione del super-tipo
 - la *rep* degli oggetti del sotto-tipo contiene anche le variabili di istanza definite nell’implementazione del super-tipo
 - alcuni metodi possono essere ereditati
 - di altri il sotto-tipo può definire una nuova implementazione

Gerarchie di tipi in Java

- Attraverso l'ereditarietà
 - una classe può essere sotto-classe di un'altra (la sua super-classe) e implementare zero o più interfacce
- il super-tipo (classe o interfaccia) fornisce in ogni caso la specifica del tipo
 - le interfacce fanno solo questo
 - le classi possono anche fornire parte dell'implementazione

Gerarchie di tipi in Java

- I super-tipi sono definiti da
 - classi
 - interfacce
- Le classi possono essere
 - astratte (forniscono un'implementazione parziale del tipo)
 - ✓ non hanno oggetti
 - ✓ il codice esterno non può chiamare i loro costruttori
 - ✓ possono avere metodi astratti la cui implementazione è lasciata a qualche sotto-classe
 - concrete (forniscono un'implementazione piena del tipo)
- Le classi astratte e concrete possono contenere metodi finali
 - non possono essere reimplementati da sotto-classi

Gerarchie di tipi in Java

- Le interfacce definiscono solo il tipo (specifica) e non implementano nulla
 - contengono solo (le specifiche di) metodi
 - ✓ pubblici
 - ✓ non statici
 - ✓ astratti

Gerarchie di tipi in Java

- Una sotto-classe dichiara la super-classe che estende (e/o le interfacce che implementa)
 - ha tutti i metodi della super-classe con gli stessi nomi e signature
 - può implementare i metodi astratti e reimplementare gli altri (purché non final)
 - qualunque metodo sovrascritto deve avere signature identica a quella della super-classe
 - ma i metodi della sotto-classe possono sollevare meno eccezioni
- La rappresentazione di un oggetto di una sotto-classe consiste delle variabili di istanza proprie e di quelle della super-classe
 - quelle della super-classe non possono essere accedute direttamente se sono (come dovrebbero essere) dichiarate private
- Ogni classe che non ne estenda esplicitamente un'altra estende implicitamente Object

Gerarchie di tipi in Java

- La super-classe può lasciare parti della sua implementazione accessibili alle sotto-classi
 - dichiarando metodi e variabili `protected`
 - ✓ implementazioni delle sotto-classi più efficienti
 - ✓ si perde l'astrazione completa, che dovrebbe consentire di reimplementare la super-classe senza influenzare l'implementazione delle sotto-classi
 - ✓ le entità `protected` sono visibili anche all'interno dell'eventuale package che contiene la super-classe
- Meglio interagire con le super-classi attraverso le loro interfacce pubbliche

Esempio: gerarchia con super-tipo classe concreta

- In cima alla gerarchia si trova una variante di IntSet
 - la solita, con in più il metodo subset
 - la classe non è astratta
 - fornisce un insieme di metodi che le sotto-classi possono ereditare, estendere o sovra-scrivere

Specifica del super-tipo

```
public class IntSet {
    // OVERVIEW: un IntSet e' un insieme modificabile
    // di interi di dimensione qualunque
    public IntSet( )
        // EFFECTS: inizializza this a vuoto
    public void insert(int x)
        // EFFECTS: aggiunge x a this
    public void remove(int x)
        // EFFECTS: toglie x da this
    public boolean isIn(int x)
        // EFFECTS: se x appartiene a this
        // ritorna true, altrimenti false
    public int size( )
        // EFFECTS: ritorna la cardinalità di this
    public boolean subset (IntSet s)
        // EFFECTS: se s e' un sottoinsieme di this
        // ritorna true, altrimenti false
}
```

Implementazione del super-tipo

```
public class IntSet {
    // OVERVIEW: un IntSet e' un insieme modificabile
    // di interi di dimensione qualunque
    private Vector els; // la rappresentazione
    public IntSet( ) { els = new Vector( ); }
    // EFFECTS: inizializza this a vuoto
    private int getIndex(Integer x) { ... }
    // EFFECTS: se x occorre in this ritorna la posizione
    // in cui si trova, altrimenti -1
    public boolean isIn(int x)
    // EFFECTS: se x appartiene a this ritorna true,
    // altrimenti false
    { return getIndex(new Integer(x)) >= 0; }
    public boolean subset(IntSet s) {
    // EFFECTS: se s e' un sottoinsieme di this
    // ritorna true, altrimenti false
    if (s == null) return false;
    for (int i = 0; i < els.size( ); i++)
        if (!s.isIn(((Integer) els.get(i)).intValue()))
            return false;
    return true;
    }
}
```

Un sotto-tipo: MaxIntSet

- Si comporta come IntSet
 - ma ha un metodo nuovo max, che ritorna l'elemento massimo nell'insieme
 - la specifica di MaxIntSet definisce solo quel che c'è di nuovo
 - ✓ il costruttore e il metodo max
 - tutto il resto della specifica viene ereditato da IntSet
- Perché non realizzare semplicemente un metodo max *stand alone* esterno alla classe IntSet?
 - facendo un sotto-tipo si riesce ad implementare max in modo più efficiente

Specifica del sotto-tipo

```
public class MaxIntSet extends IntSet {
    // OVERVIEW: un MaxIntSet è un sotto-tipo di IntSet che
    // lo estende con il metodo max
    public MaxIntSet( )
        // EFFECTS: inizializza this al MaxIntSet vuoto
    public int max( ) throws EmptyException
        // EFFECTS: se this è vuoto solleva EmptyException,
        // altrimenti ritorna l'elemento massimo in this
}
```

- la specifica di MaxIntSet definisce solo quel che c'è di nuovo
 - ✓ il costruttore
 - ✓ il metodo max
- tutto il resto della specifica viene ereditato da IntSet

Implementazione di `MaxIntSet`

- Per evitare di generare ogni volta tutti gli elementi dell'insieme, memorizziamo in una variabile di istanza di `MaxIntSet` il valore massimo corrente
 - oltre ad implementare `max`
 - dobbiamo reimplementare `insert` e `remove` per tenere aggiornato il valore massimo corrente
 - sono i soli metodi per cui c'è overriding
 - tutti gli altri vengono ereditati da `IntSet`

Implementazione del sotto-tipo 1

```
public class MaxIntSet extends IntSet {
    // OVERVIEW: un MaxIntSet e' un sotto-tipo di IntSet
    // che lo estende con il metodo max
    private int mass;
        // l'elemento massimo, se this non e' vuoto
    public MaxIntSet( ) { super( ); }
        // EFFECTS: inizializza this al MaxIntSet vuoto
    ...
}
```

- Chiamata esplicita del costruttore del super-tipo
 - potrebbe in questo caso essere omessa
 - necessaria se il costruttore ha parametri
- Nient'altro da fare
 - perché mass non ha valore quando els è vuoto

Implementazione del sotto-tipo 2

```
public class MaxIntSet extends IntSet {
    // OVERVIEW: un MaxIntSet e' un sotto-tipo di
    // IntSet che lo estende con il metodo max

    private int mass;
        // l'elemento massimo, se this non e' vuoto
    ...
    public int max( ) throws EmptyException {
        // EFFECTS: se this e' vuoto solleva
        // EmptyException, altrimenti
        // ritorna l'elemento massimo in this
        if (size( ) == 0) throw
            new EmptyException("MaxIntSet.max");
        return mass;
    }
    ...
}
```

- Usa un metodo ereditato dal super-tipo (`size`)

Implementazione del sotto-tipo 3

```
public class MaxIntSet extends IntSet {
    // OVERVIEW: un MaxIntSet e' un sotto-tipo di
    // IntSet che lo estende con il metodo max

    private int mass;
        // l'elemento massimo, se this non e' vuoto
    ...
    public void insert (int x) {
        if (size( ) == 0 || x > mass) mass = x;
        super.insert(x);
    }
    ...
}
```

- Ha bisogno di usare il metodo insert del super-tipo
 - attraverso il prefisso super
- Per un programma esterno che usi il tipo MaxIntSet il metodo overridden insert del super-tipo non è accessibile

Implementazione del sotto-tipo 4

```
public class MaxIntSet extends IntSet {
    // OVERVIEW: un MaxIntSet e' un sotto-tipo di
    // IntSet che lo estende con il metodo max

    private int mass;
    // l'elemento massimo, se this non e' vuoto
    ...
    public void remove(int x) { ... }
```

- Problema: come possiamo riscrivere remove se non abbiamo accesso agli elementi del super-tipo? Lo discutiamo dopo!!
- Invariante di rappresentazione per MaxIntSet
 - // $I_{\text{MaxIntSet}}(c) = c.\text{size}() > 0 \implies$
 - // $(c.\text{mass} \text{ appartiene a } AF_{\text{IntSet}}(c) \ \&\&$
 - // per tutti gli x in $AF_{\text{IntSet}}(c)$, $x \leq c.\text{mass}$

AF di sotto-classi di una classe concreta

- Definita in termini di quella del super-tipo, nome della classe come indice per distinguerle
- Funzione di astrazione per `MaxIntSet`

```
// la funzione di astrazione è  
//  $AF_{MaxIntSet}(c) = AF_{IntSet}(c)$ 
```

- La funzione di astrazione è la stessa di `IntSet`: produce lo stesso insieme di elementi dalla stessa rappresentazione (els)
 - il valore della variabile `mass` non ha influenza sull'astrazione

IR di sotto-classi di una classe concreta

- Invariante di rappresentazione per `MaxIntSet`

```
// IMaxIntSet (c) = c.size() > 0 ==>  
// (c.mass appartiene a AFIntSet(c) &&  
// per tutti gli x in AFIntSet(c), x <= c.mass)
```

- L'invariante non include (e dunque non utilizza) l'invariante di `IntSet` perché tocca all'implementazione di `IntSet` preservarlo
 - le operazioni di `MaxIntSet` non possono interferire perché operano sulla IR del super-tipo solo attraverso i metodi pubblici
- Usa la funzione di astrazione del super-tipo

repOk di sotto-classi di una classe concreta

- Invariante di rappresentazione per `MaxIntSet`

```
//  $I_{\text{MaxIntSet}}(c) = c.\text{size}() > 0 \implies$   
//  $(c.\text{mass}$  appartiene a  $\text{AF}_{\text{IntSet}}(c)$  &&  
// per tutti gli  $x$  in  $\text{AF}_{\text{IntSet}}(c)$ ,  $x \leq c.\text{mass}$ )
```

- L'implementazione di `repOk` deve verificare l'invariante della super-classe perché la correttezza di questo è necessaria per la correttezza dell'invariante della sotto-classe

E se il super-tipo espone la rappresentazione?

- Il problema della remove si potrebbe risolvere facendo vedere alla sotto-classe la rappresentazione della super-classe
 - dichiarando `els protected` nell'implementazione di `IntSet`
- in questo caso, l'invariante di rappresentazione di `MaxIntSet` deve includere quello di `IntSet`
 - perché l'implementazione di `MaxIntSet` potrebbe violarlo

```
//  $I_{\text{MaxIntSet}}(c) = I_{\text{IntSet}}(c) \ \&\& \ c.\text{size}() > 0 \implies$   
//  $(c.\text{mass} \text{ appartiene a } AF_{\text{IntSet}}(c) \ \&\&$   
//  $\text{per tutti gli } x \text{ in } AF_{\text{IntSet}}(c), \ x \leq c.\text{mass})$ 
```

Classi astratte come super-tipi

- Implementazione parziale di un tipo
- Può avere variabili di istanza e uno o più costruttori
- Non ha oggetti
- I costruttori possono esser chiamati solo dalle sotto-classi per inizializzare la parte di rappresentazione della super-classe
- Può contenere metodi astratti (senza implementazione)
- Può contenere metodi regolari (implementati)
 - questo evita di implementare più volte i metodi quando la classe abbia più sotto-classi e permette di dimostrare più facilmente la correttezza
 - l'implementazione può utilizzare i metodi astratti
 - ✓ la parte generica dell'implementazione è fornita dalla super-classe
 - ✓ le sotto-classi forniscono i dettagli

Può convenire che `IntSet` sia astratta

- Vogliamo definire (come sotto-tipo di `IntSet`) il tipo `SortedIntSet`
 - un nuovo metodo `subset` (overloaded) per ottenere una implementazione più efficiente quando l'argomento è di tipo `SortedIntSet`
- Vediamo la specifica di `SortedIntSet`

Specifica del sotto-tipo

```
public class SortedIntSet extends IntSet {
    // OVERVIEW: un SortedIntSet e' un sotto-tipo di IntSet
    // che lo estende con i metodi max e subset e in cui
    // gli elementi sono accessibili in modo ordinato
    public SortedIntSet( )
        // EFFECTS: inizializza this all'insieme vuoto
    public int max( ) throws EmptyException
        // EFFECTS: se this e' vuoto solleva EmptyException,
        // altrimenti ritorna l'elemento massimo in this
    public boolean subset(Intset s)
        // EFFECTS: se s e' un sottoinsieme di this
        // ritorna true, altrimenti false
}
```

- La rappresentazione degli oggetti di tipo SortedIntSet potrebbe utilizzare una lista ordinata
 - non serve più a nulla la variabile di istanza ereditata da IntSet
 - il vettore els andrebbe eliminato da IntSet
 - senza els, IntSet non può avere oggetti e quindi deve essere astratta

IntSet come classe astratta

- Specifiche uguali a quelle già viste
- Dato che la parte importante della rappresentazione (gli elementi dell'insieme) non è definita qui, sono astratti i metodi insert, remove e repOk
- isIn, subset e toString sono implementati in termini del metodo astratto elements
- Teniamo traccia nella super-classe della dimensione con una variabile intera dim
 - che è ragionevole sia visibile dalle sotto-classi (protected)
 - la super-classe non può nemmeno garantire le proprietà di dim
 - ✓ il metodo repOk è astratto
- Non c'è funzione di rappresentazione
 - tipico delle classi astratte, perché la vera implementazione è fatta nelle sotto-classi

IntSet come classe astratta

```
public abstract class IntSet {
    protected int dim; // la dimensione

    // costruttore
    public IntSet( ) { dim = 0; }

    // metodi astratti
    public abstract void insert(int x);
    public abstract void remove(int x);
    public abstract boolean repOk( );
    public abstract Vector elements( );

    // metodi
    public boolean isIn (int x)
    // implementazione
    public int size( ) { return dim; }
    // implementazioni di subset e toString
}
```

SortedIntSet

```
public class SortedIntSet extends IntSet {
    private OrderedIntList els; // la rappresentazione
    // la funzione di astrazione:
    // a(c) = {c.els[1] , . . . , c.els[c.dim]}
    // l'invariante di rappresentazione:
    // I(c) = c.els != null && c.dim = c.els.size()

    // costruttore
    public SortedIntSet( ) { els = new OrderedIntList(); }
    // metodi
    public int max( ) throws EmptyException {
        if (dim == 0) throw new EmptyException("SortedIntSet.max");
        return els.max( );
    }

    // implementazione di insert, remove e repOk
    ...
}
```

- La funzione di astrazione va da liste ordinate a insiemi
 - gli elementi della lista si accedono con la notazione []
- L'invariante di rappresentazione pone vincoli su tutte e due le variabili di istanza (anche quella ereditata)
 - els è assunto ordinato (perché così è in OrderedIntList)
- Si assume esistano per OrderedIntList anche size e max

SortedIntSet

```
public class SortedIntSet extends IntSet {
    private OrderedIntList els; // la rappresentazione
    // la funzione di astrazione:
    //  $a(c) = \{c.els[0], \dots, c.els[c.(dim-1)]\}$ 
    // l'invariante di rappresentazione:
    //  $I(c) = c.els \neq null \ \&\& \ c.dim = c.els.size()$ 
    ...
    public boolean subset(IntSet s) {
        try { return subset((SortedIntSet) s); }
        catch (ClassCastException e) { return super.subset(s); }
    }
    public boolean subset(SortedIntSet s)
        // qui si approfitta del fatto che smallToBig
        // di OrderedIntList
        // ritorna gli elementi in ordine crescente
    }
}
```

Gerarchie di classi astratte

- Anche le sotto-classi possono essere astratte
- Possono continuare a elencare come astratti alcuni dei metodi astratti della super-classe
- Possono introdurre nuovi metodi astratti

Ereditarietà multipla

- Una classe può estendere soltanto una classe
- Ma può implementare una o più interfacce
- Si riesce così a realizzare una forma di ereditarietà multipla
 - nel senso di super-tipi multipli
 - anche se niente di implementato è ereditato dalle interfacce

```
public class SortedIntSet extends IntSet
    implements SortedCollection { ... }
```

SortedIntSet è sotto-tipo sia di IntSet che di SortedCollection

Discussione finale: quadrati vs. rettangoli

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

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

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

Discussione finale

- **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
 - trasformarli in oggetti immutabili

