



ASTRAZIONI SUI DATI : RAGIONARE SUI TIPI DI DATO ASTRATTI

1



Verifica & Validazione

- ✎ Proprietà dell'astrazione
 - modificabilità
 - categorie di operazioni
 - dimostrare proprietà dell'astrazione
- ✎ Dimostrare proprietà dell'implementazione
 - funzione di astrazione
 - invariante di rappresentazione
 - dimostrazione mediante induzione sui dati

2

Modificabilità 1



- ✦ i tipi non modificabili sono più “sicuri”
 - la condivisione di sottostrutture non crea problemi
 - Thread safe
- ✦ i tipi non modificabili sono spesso più inefficienti
 - la necessità di costruire spesso copie di oggetti può complicare la vita al garbage collector
- ✦ la scelta dovrebbe comunque tener conto delle caratteristiche dei concetti matematici o degli oggetti del mondo reale modellati dal tipo
 - gli interi non sono modificabili
 - gli insiemi sono modificabili
 - i conti correnti sono modificabili
 -

3

Modificabilità 2



- ✦ un tipo non modificabile può essere implementato utilizzando strutture modificabili
 - arrays, vectors, tipi record, tipi astratti modificabili

```
public class Poly {  
    // OVERVIEW: un Poly è un polinomio a  
    // coefficienti interi non modificabile  
    // esempio:  $c_0 + c_1*x + c_2*x^2 + \dots$   
    private int[] termini; // la rappresentazione  
    private int deg; // la rappresentazione
```

- ✦ attenzione comunque agli effetti laterali “nascosti”
 - un metodo può restituire la rappresentazione modificabile (esporre la rep)
 - un tipo non modificabile può contenere un tipo modificabile che può essere restituito da un metodo (e poi modificato)

4

Categorie di operazioni 1



creatori

- creano oggetti del loro tipo “dal nulla”
✓ sicuramente costruttori

```
public IntSet ()  
    // EFFECTS: inizializza this a vuoto
```

produttori

- prendono come argomenti oggetti del loro tipo e ne costruiscono altri. Possono essere costruttori o metodi

```
public Poly sub (Poly q) throws  
    NullPointerException  
    // EFFECTS: q=null solleva NullPointerException  
    // altrimenti ritorna this - q
```

5

Categorie di operazioni 2



Modificatori: modificano gli oggetti del loro tipo

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

Osservatori: prendono oggetti del loro tipo e restituiscono valori di altri tipi per ottenere informazioni sugli oggetti

```
public boolean isIn (int x)  
    // EFFECTS: se x appartiene a this ritorna  
    // true, altrimenti false  
public int coeff (int d)  
    // EFFECTS: ritorna il coefficiente del  
    // termine in this che ha come esponente d
```

6

Operazioni



- ✎ Quali e quante operazioni in una astrazione?
 - almeno un creatore
 - qualche produttore, se il tipo non è modificabile
 - qualche modificatore, se il tipo è modificabile
 - ✓ attraverso creatori e produttori (o modificatori) dovremmo essere in grado di generare tutti i valori astratti
 - qualche osservatore
- ✎ Come operare?
 - bilanciando efficienza dell'implementazione dei metodi e complessità della classe

7

Dimostrare proprietà dell'astrazione



- ✎ Aspetto significativo dello sviluppo di software "safe":
garantire proprietà delle astrazioni
 - Storicamente sviluppato per le astrazioni procedurali, ma più interessante per le astrazioni sui dati
- ✎ per dimostrare la proprietà dobbiamo utilizzare le specifiche
- ✎ Tecnica di dimostrazione: induzione strutturale sul tipo astratto
 - si dimostra che la proprietà vale sui valori astratti costruiti dai costruttori
 - si dimostra che se la proprietà vale prima allora vale anche dopo per ogni applicazione di modificatore o produttore

8

Una proprietà di IntSet



```
public class IntSet {  
    // OVERVIEW: un IntSet è 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
```

- ✎ Proprietà': per ogni IntSet la sua size è ≥ 0
- ✎ dobbiamo considerare il costruttore ed i due modificatori

9

Una proprietà di IntSet



- ✎ per ogni IntSet la sua size è ≥ 0
- ✎ per il costruttore

```
public IntSet ()  
    // EFFECTS: inizializza this a vuoto
```

- ✎ l'insieme vuoto ha cardinalità 0
- ✎ La proprietà vale per il costruttore.

10

Una proprietà di IntSet



☞ per ogni IntSet la sua size è ≥ 0

☞ per ogni modificatore

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

☞ se la proprietà vale prima dell'inserimento, vale anche dopo perché l'inserimento può solo incrementare la cardinalità

11

Una proprietà di IntSet



☞ per ogni IntSet la sua size è ≥ 0

☞ per ogni modificatore

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

☞ se la proprietà vale prima della rimozione, vale anche dopo perché la rimozione può ridurre la cardinalità solo se l'elemento era contenuto al momento della chiamata

12

Correttezza dell'implementazione



- ✦ Dimostrare che le implementazioni dei metodi soddisfano le rispettive specifiche
 - non possiamo utilizzare la metodologia appena vista
- ✦ l'implementazione utilizza la rappresentazione
 - nel caso di `IntSet`: **private Vector els**;
- ✦ le specifiche esprimono proprietà dell'astrazione
 - nel caso di `IntSet`

```
public boolean isIn (int x)
// EFFECTS: se x appartiene a this ritorna
// true, altrimenti false
```

- ✦ è necessario mettere in relazione tra loro due "insiemi di valori": valori astratti e quelli concreti

13

La funzione di astrazione



- ✦ la funzione di astrazione cattura l'intenzione del progettista nello scegliere una particolare rappresentazione
- ✦ la funzione di astrazione
$$\alpha : C \rightarrow A$$
porta da uno stato concreto
 - lo stato di un oggetto della definito in termini della sua rappresentazione a uno stato astratto
 - lo stato dell'oggetto astratto viene definito nella clausola OVERVIEW

```
public class IntSet {
// OVERVIEW: un IntSet è un insieme modificabile
// di interi di dimensione qualunque
private Vector els; // la rappresentazione
```

- ✦ La funzione di asrazione porta vettori in insiemi

14

La funzione di astrazione



- ☛ la funzione di astrazione può essere una funzione molti-a-uno

```
public class IntSet {  
  // OVERVIEW: un IntSet è un insieme modificabile  
  // di interi di dimensione qualunque  
  private Vector els; // la rappresentazione
```

- più stati concreti (vettori di interi) vengono portati nello stesso stato astratto (insieme)
 - ✓ $\alpha([1,2]) = \{1,2\}$
 - ✓ $\alpha([2,1]) = \{1,2\}$
- ☛ La funzione di astrazione deve sempre essere definita
 - perché è una parte importante delle decisioni relative all'implementazione
 - sintatticamente, viene inserita come commento all'implementazione, dopo le dichiarazioni di variabili di istanza che definiscono la rappresentazione
 - senza funzione di astrazione, non si può dimostrare la correttezza dell'implementazione

15

La funzione di astrazione



- ☛ per definire formalmente la funzione di astrazione dobbiamo avere una notazione per i valori astratti
- ☛ quando è necessario, forniamo (sempre nella OVERVIEW) la notazione per descrivere un tipico stato (valore) astratto
- ☛ nella definizione della funzione di astrazione, useremo la notazione di Java

16

La funzione di astrazione di IntSet

```
public class IntSet {
    // OVERVIEW: un IntSet è un insieme modificabile
    // di interi di dimensione qualunque
    // un tipico IntSet è {x1, ..., xn}
    private Vector els; // la rappresentazione
    // la funzione di astrazione
    //  $\alpha(c) = \{ c.els.get(i).intValue() \mid$ 
    //            $0 \leq i < c.els.size() \}$ 
}
```

17

La funzione di astrazione di Poly

```
public class Poly {
    // OVERVIEW: un Poly è un polinomio a
    // coefficienti interi non modificabile
    // un tipico Poly:  $c_0 + c_1*x + c_2*x^2 + \dots$ 
    private int[] termini; // la rappresentazione
    private int deg; // la rappresentazione
    // la funzione di astrazione
    //  $\alpha(c) = c_0 + c_1*x + c_2*x^2 + \dots$  tale che
    //  $c_i = c.termini[i]$  se  $0 \leq i < c.termini.length$ 
    //  $= 0$  altrimenti
}
```

- ☛ notare che il valore di deg non ha nessuna influenza sulla funzione di astrazione
 - è una informazione derivabile dall'array termini che utilizziamo nello stato concreto per questioni di efficienza

18

La funzione di astrazione ed il metodo toString



- se pensiamo a valori astratti rappresentati come stringhe
 - possiamo implementare la funzione di astrazione, che è esattamente il metodo toString
 - utile per stampare valori astratti

```
//  $\alpha(c) = \{ c.els.get(i).intValue() \mid$   
//  $0 \leq i < c.els.size() \}$ 
```

```
//  $\alpha(c) = c_0 + c_1*x + c_2*x^2 + \dots$  tale che  
//  $c_i = c.termini[i]$  se  $0 \leq i < c.termini.size()$   
//  $= 0$  altrimenti
```

19

toString per IntSet



```
//  $\alpha(c) = \{ c.els.get(i).intValue() \mid$   
//  $0 \leq i < c.els.size() \}$ 
```

```
public String toString ()  
{String s = "{";  
for (int i = 0; i < els.size() - 1; i++) {  
s = s + els.get(i).toString() + ","; }  
if (els.size() > 0) {  
s = s + els.get(els.size() - 1).toString(); }  
s = s + "}";  
return (s);}
```

20

Verso l'invariante di rappresentazione



- ✎ non tutti gli stati concreti “rappresentano” correttamente uno stato astratto

```
public class IntSet {  
    // OVERVIEW: un IntSet è un insieme modificabile  
    // di interi di dimensione qualunque  
    // un tipico IntSet è {x1, ..., xn}  
    private Vector els; // la rappresentazione  
    // la funzione di astrazione  
    //  $\alpha(c) = \{ c.els.get(i).intValue() \mid 0 \leq i < c.els.size() \}$   
}
```

- ✎ il vettore els potrebbe contenere più occorrenze dello stesso elemento
 - questo potrebbe anche essere coerente con la funzione di astrazione
 - ma non rispecchierebbe la nostra scelta di progetto riflessa nell'implementazione dei metodi

21

L'invariante di rappresentazione



- ✎ l'invariante di rappresentazione (rep invariant) è un predicato

$I : C \rightarrow \text{boolean}$

che è verificato solo per gli stati concreti che sono rappresentazioni legittime di uno stato astratto

- ✎ l'invariante di rappresentazione, insieme alla funzione di astrazione, riflette le scelte relative alla rappresentazione
 - deve essere inserito nella documentazione della implementazione come commento, insieme alla funzione di astrazione
- ✎ la funzione di astrazione è una funzione parziale definita solo per stati concreti che soddisfano l'invariante

22

L'invariante di rappresentazione di IntSet



```
public class IntSet {
    // OVERVIEW: un IntSet è un insieme modificabile
    // di interi di dimensione qualunque
    // un tipico IntSet è {x1, ..., xn}
    private Vector els; // la rappresentazione
    // la funzione di astrazione:
    //  $\alpha(c) = \{ c.els.get(i).intValue() \mid 0 \leq i < c.els.size() \}$ 
    // l'invariante di rappresentazione:
    //  $!c = c.els \neq \text{null} \text{ e}$ 
    // per ogni intero  $i$ ,  $c.els.get(i)$  è un Integer
    // e per tutti gli interi  $i, j$ , tali che
    //  $0 \leq i < j < c.els.size()$ ,
    //  $c.els.get(i).intValue() \neq c.els.get(j).intValue()$ 
    🦋 il vettore non deve essere null
    🦋 gli elementi del vettore devono essere Integer
        ◦ assunti soddisfatti in  $\alpha$ 
    🦋 tutti gli elementi sono distinti
```

23

Una diversa implementazione per IntSet



```
public class IntSet {
    // OVERVIEW: un IntSet è un insieme
    // modificabile
    // di interi di dimensione qualunque
    // un tipico IntSet è {x1, ..., xn}
    private boolean[100] els;
    private Vector altriels;
    private int dim;
    🦋 L'appartenenza di un elemento  $n$  compreso tra 0 e 99 viene
    realizzata mettendo a true  $els[n]$ 
    🦋 gli elementi maggiori di 99 sono inseriti nel vettore
    altriels gestito come nell'implementazione precedente
    🦋 dim contiene esplicitamente la cardinalità
        ◦ che sarebbe complessa da calcolare a partire da els
    🦋 implementazione sensata solo se la maggior parte degli
    elementi sono compresi nell'intervallo 0-99
```

24

Una diversa implementazione per IntSet



```
public class IntSet {
    // OVERVIEW: un IntSet è un insieme modificabile
    // di interi di dimensione qualunque
    // un tipico IntSet è {x1, ..., xn}
    private boolean[100] els;
    private Vector altriels;
    private int dim;
    // la funzione di astrazione:
    //  $\alpha(c) = \{ c.altriels.get(i).intValue() \mid$ 
    //    $0 \leq i < c.altriels.size() \} \cup$ 
    //    $\{j \mid 0 \leq j < 100 \text{ e } c.els[j] \}$ 
}
```

25

Una diversa implementazione per IntSet



```
public class IntSet {
    // OVERVIEW: un IntSet è un insieme modificabile
    // di interi di dimensione qualunque
    // un tipico IntSet è {x1, ..., xn}
    private boolean[100] els;
    private Vector altriels;
    private int dim;
    // l'invariante di rappresentazione:
    //  $!(c) = c.els \neq \text{null}$  e
    //  $c.altriels \neq \text{null}$  e
    //  $els.length = 100$  e
    // per ogni intero  $i$ ,
    //  $c.altriels.get(i)$  è un Integer,
    //  $c.altriels.get(i).intValue()$  non appartiene
    // all'intervallo 0-99, e
    // per tutti gli interi  $i, j$ , tali che
    //  $0 \leq i < j < c.altriels.size()$ ,
    //  $c.altriels.get(i).intValue() \neq$ 
    //  $c.altriels.get(j).intValue()$  e
    //  $c.dim = c.altriels.size() + \text{conta}(c.els, 0)$ 
}
```

26

Funzione ausiliaria di rep invariant



```
conta(a,i) = if (i >= a.length) return 0;
             else if (a[i]) return (1 + conta(a, i-1))
             else return (conta(a, i-1))
```

27

L'invariante di rappresentazione di Poly



```
public class Poly {
  // OVERVIEW: un Poly è un polinomio a
  // coefficienti interi non modificabile
  // un tipico Poly:  $c_0 + c_1*x + c_2*x^2 + \dots$ 
  private int[] termini; // la rappresentazione
  private int deg; // la rappresentazione
  // la funzione di astrazione
  //  $\alpha(c) = c_0 + c_1*x + c_2*x^2 + \dots$  tale che
  //  $c_i = c.termini[i]$  se  $0 \leq i < c.termini.size()$ 
  //  $= 0$  altrimenti
  // l'invariante di rappresentazione:
  //  $!(c) = c.termini \neq \text{null}$  e
  //  $c.termini.length \geq 1$  e
  //  $c.deg = c.termini.length-1$  e
  //  $c.deg > 0 \implies c.termini[deg] \neq 0$ 
}
```

28

L'invariante di rappresentazione può essere implementato



- ✦ il metodo **repOk** che verifica l'invariante può essere fornito dalla astrazione sui dati
 - pubblico perché deve poter essere chiamato da fuori della sua classe, ma non è proprio essenziale
- ✦ ha sempre la seguente specifica

```
public boolean repOk ()  
    // EFFECTS: ritorna true se il rep invariant  
    // vale per this, altrimenti ritorna false
```

29

repOK



- ✦ può essere usato da programmi di test per verificare se una implementazione preserva l'invariante
- ✦ può essere usato dentro le implementazioni di costruttori e metodi
 - creatori, modificatori e produttori dovrebbero chiamarlo prima di ritornare per assicurarsi che per l'oggetto costruito o modificato vale l'invariante
 - ✓ per esempio, dovrebbero chiamarlo insert e remove di IntSet, se l'invariante non vale si può sollevare FailureException
- ✦ in alternativa, l'implementazione è accompagnata da una *dimostrazione formale*, che ci garantisce che tutti i metodi preservano l'invariante (vedi dopo)
- ✦ la verifica "dinamica" via repOK diventa inutile!

30

repOK per Poly



```
public class Poly {
    private int[] termini; // la rappresentazione
    private int deg; // la rappresentazione
    // l(c) = c.termini != null e
    // c.termini.length >= 1 e
    // c.deg = c.termini.length-1 e
    // c.deg > 0 ==> c.termini[deg] != 0

    public boolean repOk() {
        if (termini == null || deg != termini.length - 1
            || termini.length == 0) return false;
        if (deg == 0) return true;
        return termini[deg] != 0; }
}
```

31

repOK per IntSet



```
public class IntSet {
    private Vector els; // la rappresentazione
    // l(c) = c.els != null e
    // per ogni intero i, c.els.get(i) è un Integer
    // e per tutti gli interi i,j, tali che
    // 0 <= i < j < c.els.size(),
    // c.els.get(i).intValue() !=
    // c.els.get(j).intValue()

    public boolean repOk() {
        if (els == null) return false;
        for (int i = 0; i < els.size(); i++) {
            Object x = els.get(i);
            if (!(x instanceof Integer)) return false;
            for (int j = i + 1; j < els.size(); j++)
                if (x.equals(els.get(j))) return false; }
        return true; }
}
```

32

Correttezza di una implementazione



- ✦ Invece di “eseguire” repOk (controllo dinamico), possiamo dimostrare formalmente che, ogniqualvolta un oggetto del nuovo tipo è manipolato all'esterno della classe, esso soddisfa l'invariante
 - induzione sul tipo di dato
- ✦ Dobbiamo poi dimostrare, per ogni metodo, che l'implementazione soddisfa la specifica
 - usando la funzione di rappresentazione

33

Verifica del rep invariant



- ✦ (base) dimostriamo che l'invariante vale per gli oggetti restituiti dai costruttori
- ✦ (passo induttivo) dimostriamo che vale per tutti i metodi (produttori e modificatori)
 - assumiamo che l'invariante valga per **this** e per tutti gli argomenti del tipo
 - dimostriamo che vale quando il metodo ritorna
 - ✓ per **this**
 - ✓ per tutti gli argomenti del tipo
 - ✓ per gli oggetti del tipo ritornati
- ✦ induzione sul numero di invocazioni di metodi usati per produrre il valore corrente dell'oggetto
 - la base dell'induzione è fornita dai costruttori

34

Correttezza di IntSet



```
public class IntSet {
    private Vector els; // la rappresentazione
    // I(c) = c.els != null e
    // per ogni intero i, c.els.get(i) è un Integer
    // e per tutti gli interi i,j, tali che
    //     0 <= i < j < c.els.size(),
    // c.els.get(i).intValue() !=
    //     c.els.get(j).intValue()

    public IntSet ()
    {els = new Vector();}
```

- ✎ il costruttore soddisfa l'invariante perché restituisce un Vector vuoto

35

Correttezza di IntSet



```
public class IntSet {
    private Vector els; // la rappresentazione
    // I(c) = c.els != null e
    // per ogni intero i, c.els.get(i) è un Integer
    // e per tutti gli interi i,j, tali che
    //     0 <= i < j < c.els.size(),
    // c.els.get(i).intValue() !=
    //     c.els.get(j).intValue()
    public void insert (int x)
    {Integer y = new Integer(x);
    if (getIndex(y) < 0) els.add(y); }
    private int getIndex (Integer x)
    // EFFECTS: se x occorre in this ritorna la
    // posizione in cui si trova, altrimenti -1
```

- ✎ il metodo insert soddisfa l'invariante perché aggiunge x a this solo se x non è già in this

36

Correttezza di IntSet



```
public class IntSet {
    private Vector els; // la rappresentazione
    // I(c) = c.els != null e
    // per ogni intero i, c.els.get(i) è un Integer
    // e per tutti gli interi i,j, tali che
    // 0 <= i < j < c.els.size(),
    // c.els.get(i).intValue() !=
    // c.els.get(j).intValue()
    public void remove (int x)
    {int i = getIndex(new Integer(x));
    if (i < 0) return;
    els.set(i, els.lastElement());
    els.remove(els.size() - 1);}
}
```

- ✎ il metodo `remove` soddisfa l'invariante perché rimuove `x` da `this` solo se `x` è in `this`

37

Correttezza di Poly



```
public class Poly {
    private int[] termini; // la rappresentazione
    private int deg; // la rappresentazione
    // I(c) = c.termini != null e
    // c.termini.length >= 1 e
    // c.deg = c.termini.length-1 e
    // c.deg > 0 ==> c.termini[deg] != 0
    public Poly ()
    {termini = new int[1]; deg = 0; }
}
```

- ✎ il primo costruttore soddisfa l'invariante perché restituisce un `Array` di un elemento e `deg = 0`

38

Correttezza di Poly



```
public class Poly {
    private int[] termini; // la rappresentazione
    private int deg; // la rappresentazione
    // l(c) = c.termini != null e
    // c.termini.length >= 1 e
    // c.deg = c.termini.length-1 e
    // c.deg > 0 ==> c.termini[deg] != 0
    public Poly (int c, int n) throws NegativeExponentExc
    if (n < 0) throw new NegativeExponentExc ("Poly(int,int)
        constructor");
    if (c == 0)
        {termini = new int[1]; deg = 0; return; }
    termini = new int[n+1];
    for (int i = 0; i < n; i++) termini[i] = 0;
    termini[n] = c; deg = n; }
}
```

- ✎ il secondo costruttore soddisfa l'invariante perché testa esplicitamente il caso $c=0$

39

Correttezza di Poly



```
public class Poly {
    private int[] termini; // la rappresentazione
    private int deg; // la rappresentazione
    // l(c) = c.termini != null e
    // c.termini.length >= 1 e
    // c.deg = c.termini.length-1 e
    // c.deg > 0 ==> c.termini[deg] != 0
    public Poly sub (Poly q) throws
        NullPointerException
    {return add(q.minus()); }
}
```

- ✎ il metodo sub soddisfa l'invariante perché
 - o lo soddisfano q e this
 - o lo soddisfano add e minus

40

Le implementazioni dei metodi soddisfano la specifica



- ✎ si ragiona un metodo alla volta
- ✎ ciò è possibile solo perché abbiamo già dimostrato che il rep invariant è soddisfatto da tutte le operazioni
 - il rep invariant cattura le assunzioni comuni fra le varie operazioni
 - permette di trattarle separatamente

41

Correttezza dei metodi di un tipo astratto



**l'induzione sulle chiamate di procedura
nel caso di astrazioni procedurali**

```

Procedura P
  // REQUIRES: prep
  // EFFECTS: postp
{...;
 Q;
...}
Procedura Q
  // REQUIRES: preq
  // EFFECTS: postq
1.  assumo prep
2.  ...
3.  dimostro preq
4.  assumo postq
5.  ...
6.  dimostro postp
    
```

**l'induzione sulle chiamate di metodo
nel caso di astrazioni sui dati**

```

Metodo P
  // REQUIRES: prep
  // EFFECTS: postp
{...;
 Q;
...}
Metodo Q
  // REQUIRES: preq
  // EFFECTS: postq
1.  assumo prep
2.  ...
3.  dimostro preq
4.  assumo postq
5.  ...
6.  dimostro postp
    
```

- devo dimostrare che $post_p$ vale in uno stato concreto σ
- devo prima trasformare σ in uno stato astratto $\alpha(\sigma)$, usando la funzione di astrazione α perché $post_p$ è definito sugli stati astratti

42

Correttezza di IntSet



```
public class IntSet {
  private Vector els; // la rappresentazione
  // la funzione di astrazione
  //  $\alpha(c) = \{ c.els.get(i).intValue() \mid$ 
  //  $0 \leq i < c.els.size() \}$ 
  public IntSet ()
    // EFFECTS: inizializza this a vuoto
    {els = new Vector();}
```

- ✎ l'astrazione di un vettore vuoto è proprio l'insieme vuoto

43

Correttezza di IntSet



```
public class IntSet {
  private Vector els; // la rappresentazione
  // la funzione di astrazione
  //  $\alpha(c) = \{ c.els.get(i).intValue() \mid$ 
  //  $0 \leq i < c.els.size() \}$ 
  public int size ()
    // EFFECTS: ritorna la cardinalità di this
    {return els.size(); }
```

- ✎ il numero di elementi del vettore è la cardinalità dell'insieme perché
 - la funzione di astrazione mappa gli elementi del vettore in quelli dell'insieme
 - il rep invariant garantisce che non ci sono elementi duplicati in els senza dover andare a guardare come è fatta insert

44

Correttezza di IntSet 3



```
public class IntSet {
    private Vector els; // la rappresentazione
    // la funzione di astrazione
    //  $\alpha(c) = \{ c.els.get(i).intValue() \mid 0 \leq i < c.els.size() \}$ 
    public void remove (int x)
        // EFFECTS: toglie x da this
        {int i = getIndex(new Integer(x));
        if (i < 0) return;
        els.set(i, els.lastElement());
        els.remove(els.size() - 1);}
}
```

- ✦ se x non occorre nel vettore non fa niente
 - corretto perché in base alla funzione di astrazione x non appartiene all'insieme
- ✦ se x occorre nel vettore lo rimuove
 - e quindi in base alla funzione di astrazione x non appartiene all'insieme modificato