PROGRAMMAZIONE 2
9. ADT: implementazione e verifica

Data abstraction via specifica

- Abbiamo fatto vedere come definire la specifica di un TDA, per astrale dall'implementazione
- Come si implementa un TDA?
- Dobbiamo fornire la rappresentazione
- La rappresentazione deve essere nascosta all'utente esterno, ma visibile all'implementazione delle operazioni (costruttori e metodi)
- InCome si dimostra che l'implementazione soddisfa la specifica?

ADT: Verifica & Validazione

- Come si implementa un ADT?
- Dimostrare proprietà dell'implementazione
  - funzione di astrazione
  - invariante di rappresentazione
  - dimostrazione mediante induzione sui dati

Astrazioni sui dati: implementazione

- Scelta fondamentale è quella della rappresentazione
  - come i valori del tipo astratto sono implementati in termini di altri tipi
    - tipi primitivi o già implementati
    - nuovi tipi astratti che facilitano l'implementazione del nostro
      - tali tipi vengono specificati
      - metodologia: iterazione del processo di decomposizione basato su astrazioni
  - la scelta deve tener conto della possibilità di implementare in modo efficiente i costruttori e gli altri metodi
- In seguito si possono implementare costruttori e metodi

La rappresentazione

- I linguaggi che permettono la definizione di tipi di dato astratti hanno meccanismi molto diversi tra loro per definire come
  - i valori del nuovo tipo sono implementati in termini di valori di altri tipi
- In Java, gli oggetti del nuovo tipo sono semplicemente collezioni di valori di altri tipi
  - definite (nella implementazione della classe) da un insieme di variabili di istanza private
    - ✓ accessibili solo dai costruttori e dai metodi della classe

I tipi record in Java: esempio

- Java non ha un meccanismo primitivo per definire tipi record (le struct di C)
  - ma è facilissimo definirli
  - anche se con una deviazione dai discorsi metodologici che abbiamo fatto
    - ✓ la rappresentazione non è nascosta (non c'è astrazione!)
    - ✓ non ci sono metodi
    - ✓ di fatto non c'è una specifica separata dall'implementazione

Un tipo record

```
class Pair{
// OVERVIEW: un tipo record
   int coeff, exp;
   // costruttore
   Pair(int c, int n) {
     // EFFECTS: inizializza il "record" con i
     // valori di c ed n
     coeff = c; exp = n;
   }
}
```

- La rappresentazione non è nascosta
  - dopo aver creato un'istanza si accedono direttamente i "campi del record"
- La visibilità della classe e del costruttore è ristretta al package in cui figura
- Non ci sono metodi diversi dal costruttore

Implementazione di `IntSet`

```
public class IntSet {
   // OVERVIEW: un IntSet è un insieme modificabile
   // di interi di dimensione qualunque
      //Elemento tipico {x1,…, xn}
   private Vector els; // la rappresentazione
   // costruttore
   public IntSet( ) {
        //EFFECTS: inizializza this all'insieme vuoto
        els = new Vector( );
   }
   ...
```

- Un insieme di interi è rappresentato da un **`Vector`**
  - più adatto dell'array, perché ha dimensione variabile
- Gli elementi di un **`Vector`** sono di tipo **`Object`**
  - non possiamo memorizzarci valori di tipo int (usiamo **`Integer`**!)

Implementazione di `IntSet`

```
public void insert(int x) {
  // EFFECTS: aggiunge x a this
 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
 for (int i = 0; i < els.size( ); i++)
     if (x.equals(els.get(i))) return i;
 return -1; }
```

- Non abbiamo occorrenze multiple di elementi
  - si semplifica l'implementazione di **`remove`**
  - Il metodo privato ausiliario **`getIndex`** ritorna un valore speciale e non solleva eccezioni
  - va bene perché è privato
- Notare l'uso del metodo **`equals`** su **`Integer`**

Implementazione di `IntSet`

```
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);
}
public boolean isIn(int x) {
   // EFFECTS: se x appartiene a this ritorna
    // true, altrimenti false
    return (getIndex(new Integer(x)) >= 0);
}
```

Implementazione di `IntSet`

```
public int size(){
// EFFECTS: ritorna la cardinalità di this
    return els.size();
  }
  public int choose( ) throws EmptyException {
    // EFFECTS: se this è vuoto, solleva
    // EmptyException, altrimenti ritorna un
    // elemento qualunque contenuto in this
    if (els.size() == 0) throw
        new EmptyException("IntSet.choose");
   return ((Integer) els.lastElement()).intValue();
  }
```

- Anche se `lastElement` potesse sollevare un'eccezione, in questo caso non può succedere. Come mai?

Correttezza dell'implementazione

- Dimostrare che la rappresentazione rappresenta in modo corretto i valori del tipo di dato astratto descritto in OVERVIEW

- Dimostrare che le implementazioni dei metodi soddisfano le rispettive specifiche (definite dalle pre-post condizioni)

Correttezza dell'implementazione

- Non possiamo usare la metodologia appena vista!!!!
- Da una parte abbiamo 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
```

- Come mettere in relazione tra loro due insiemi di valori: valori astratti e quelli concreti???

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 implementazione definito in termini della sua rappresentazione ad un corrispondente stato astratto
  - ai valori dell'oggetto astratto è definito nella clausola OVERVIEW

```
public class IntSet {
     // OVERVIEW: un IntSet è un insieme modificabile
     // di interi di dimensione qualunque
     //Elemento tipico {x_1,…, x_n}
     private Vector els; // la rappresentazione
```

- La funzione di astrazione mappa vettori in insiemi!!!
- Quale insieme è rappresentato da un vettore?

La funzione di astrazione

- La funzione di astrazione può essere molti-a-uno

```
public class IntSet {
     // OVERVIEW: un IntSet è un insieme modificabile
     // di interi di dimensione qualunque
     //Elemento tipico {x_1,…x_n}
     private Vector els; // la rappresentazione
```

  - più stati concreti (vettori) vengono portati nello stesso stato astratto (insieme)
  - $\alpha([1,2]) = \{1,2\} = \alpha([2,1])$

Come definirla?

- 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

La funzione di astrazione: `IntSet`

```
public class IntSet {
   // OVERVIEW: un IntSet è un insieme modificabile
   // di interi di dimensione qualunque
      //Elemento tipico {x1,…, xn}
   private Vector els; // la rappresentazione
```

```
// la funzione di astrazione:
// α(c) = { c.els.get(i).intValue( ) |
      //                          0 <= i < c.els.size( ) }
```

Intuitivamente la funzione ci dice quale insieme viene rappresentato da ogni vettore

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
```

- Il vettore **`els`** potrebbe contenere più occorrenze di un elemento
  - questo potrebbe anche essere coerente con la funzione di astrazione
  - ma non rispecchierebbe la nostra scelta di progetto riflessa nell'implementazione dei metodi

Invariante di rappresentazione

- L'invariante di rappresentazione è un predicato **I** : **C** → Bool 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

Invariante di `IntSet`

```
public class IntSet {
   // OVERVIEW: un IntSet è un insieme modificabile
   // di interi di dimensione qualunque
      //Elemento tipico {x1,…, xn}

   private Vector els; // la rappresentazione
```

- Il vettore `els` non deve essere `null`
- Gli elementi del vettore devono essere `Integer`
- Tutti gli elementi del vettore sono distinti

Invariante di `IntSet`

```
public class IntSet {
   // OVERVIEW: un IntSet è un insieme modificabile
   // di interi di dimensione qualunque
      //Elemento tipico {x1,…, xn}
   private Vector els; // la rappresentazione
```

```
// l'invariante di rappresentazione:

// I(c) = c.els != null e
//per ogni i tale che 0 <= i < c.els.size(),
//  c.els.get(i) è un Integer e
//per tutti gli i,j tali che
//0 <= i < j < c. els.size()
  c.els.get(i).intValue()!=
                    c.els.get(j).intValue()
```

Correttezza di un'implementazione

- 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
- Possiamo poi dimostrare, per ogni metodo, che l'implementazione soddisfa la specifica
  - usando funzione di astrazione e invariante

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

Invariante di `IntSet`: caso base

```
public class IntSet {
     private Vector els; // la rappresentazione

// l'invariante di rappresentazione:
// I(c) = c.els != null e
//per ogni i tale che 0 <= i < c.els.size(),
//    c.els.get(i) è un Integer e
//per tutti gli 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 dato che il vettore è vuoto e non null

Invariante di `IntSet`: caso induttivo

```
public class IntSet {
     private Vector els; // la rappresentazione

// l'invariante di rappresentazione:
//  I(c) = c.els != null e
//per ogni i tale che 0 <= i < c.els.size(),
//    c.els.get(i) è un Integer e
//per tutti gli 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
```

Preserva l'invariante dato che aggiunge x al vettore solo se non era già presente

Invariante di `IntSet:` caso induttivo

```
public class IntSet {
      private Vector els; // la rappresentazione
// l'invariante di rappresentazione:
// I(c) = c.els != null e
//per ogni i tale che 0 <= i < c.els.size(),
//    c.els.get(i) è un Integer e
//per tutti gli 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);
   }
```

Preserva l'invariante banalmente dato che rimuove

Commenti

- Abbiamo dimostrato l'invariante di **`IntSet`**!!!
- Cosa succederebbe se la rappresentazione fosse pubblica?
- Altrimenti non sarebbe possibile ragionare sulla correttezza di un TDA
- Abbiamo dimostrato che l'implementazione soddisfa la specifica?

I metodi soddisfano la specifica?

- Si ragiona un metodo alla volta usando le proprietà fornite da rep invariant e dalla funzione di astrazione
- 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

Correttezza di `IntSet`

```
public class IntSet {
  private Vector els; // la rappresentazione

// la funzione di astrazione:
// α(c) = { c.els.get(i).intValue( ) |
//          0 <= 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!!

Correttezza di `IntSet`

```
public class IntSet {
     private Vector els; // la rappresentazione

// la funzione di astrazione:
// α(c) = { c.els.get(i).intValue( ) |
          //      0 <= 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

Correttezza di `IntSet`

```
public class IntSet {
      private Vector els; // la rappresentazione

// la funzione di astrazione:
// α(c) = { c.els.get(i).intValue( ) |
          //      0 <= 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

Implementare l'invariante

- Il metodo **`checkRep`** 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 checkRep( )
// EFFECTS:ritorna true se il rep invariant
// vale per this, altrimenti ritorna false
```

checkRep per IntSet

```
public boolean checkRep( ) {
   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;
  }
```

Funzione di Astrazione e toString

La funzione di astrazione deve sempre essere definita

- perché è una parte importante delle decisioni relative all'implementazione
- sintatticamente, è inserita come commento dopo le dichiarazioni di variabili di istanza che definiscono la rappresentazione
- senza funzione di astrazione, non si può dimostrare la correttezza dell'implementazione
- Se pensiamo a valori astratti rappresentati come stringhe
- possiamo anche implementare la funzione di astrazione, che è esattamente il metodo `toString` utile per stampare valori astratti

`toString` per `IntSet`

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

```
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);
}
```

Esempio: invariante troppo debole

```
public class IntSet {
private Vector els; // la rappresentazione

// l'invariante di rappresentazione:
// I(c) = c.els != null e
//per ogni i tale che 0 <= i < c.els.size(),
//   c.els.get(i) è un Integer
```

Invariante troppo debole, chiaramente soddisfatta ma non basta per dimostrare la correttezza di `IntSet` !!!!

Esempio: `IntStack` modificabile

```
public interface IntStack {
public int top( ) throws Exception;
   //se la pila non e' vuota restituisce
       //           l'elemento al top
 public void  pop() throws Exception;
     //se la pila non e' vuota rimuove
       //           l'elemento al top
 public void push(int x);
     //aggiunge  x al top della pila
 public int size( );
     //restituisce il numero di elementi della pila

 public boolean  isEmpty( );
    //se la pila e' vuota restituisce true
    // altrimenti false
}
```

Esempio: `IntStack` non modificabile

```
public interface IntStack {
 public int top( ) throws Exception;
   //se la pila non e' vuota restituisce
       //  l'elemento al top
 public IntStack  pop() throws Exception;
  //se la pila non e' vuota restituisce la pila da
  // cui e' stato rimosso l'elemento al top
 public IntStack push(int x);
    //se la pila non e' vuota restituisce la pila in
  // cui e' stato inserito l'elemento al top
 public int size( );
     //restituisce il numero di elementi della pila
 public boolean  isEmpty( );
    //se la pila e' vuota restituisce true
    // altrimenti false}
```

I metodi pop e push non sono modificatori ma produttori

Esercizio: pila modificabile

Definire la specifica dell'interfaccia **`IntStack`**
(overview pre e post)
Suggerimento: usare **`EmptyStackException`**
(checked) quando la pila e' vuota

Implementare una classe concreta **`MyIntStack`**
usando che implementa **`IntStack`**
Suggerimento: per memorizzare i valori in ordine LIFO usare un **`Vector`**

Definire funzione di astrazione e
invariante di rappresentazione