



PROGRAMMAZIONE 2

9. Astrazioni sui dati: ragionare sui tipi di dato astratti



ADR: 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

Modificabilità 1

- I tipi **non modificabili sono più “sicuri”**
 - la condivisione di sottostrutture non crea problemi
- 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 deve 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
 -

Modificabilità 2

- Un tipo non modificabile può essere implementato utilizzando strutture modificabili
 - array, vector, tipi record, tipi astratti modificabili

```
public class MyString{  
    // OVERVIEW: una stringa è una sequenza di  
    // caratteri non modificabile  
    private final char[] data; // la rappresentazione
```

- Attenzione comunque agli effetti laterali “nascosti”
 - un metodo può restituire la rappresentazione modificabile (esporre la rappresentazione)
 - un tipo non modificabile può contenerne uno modificabile che può essere restituito da un metodo (e poi modificato)

Categorie di operazioni 1

- ***creatori***

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

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

- ***produttori***

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

```
public IntSet sub(IntSet s) throws NullPointerException  
    // EFFETS: s=null solleva NullPointerException  
    // altrimenti ritorna this \ s
```

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
```

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 l'efficienza dell'implementazione dei metodi e la complessità della classe

Dimostrare proprietà dell'astrazione

- Aspetto significativo dello sviluppo di sw “safe”:
garantire proprietà delle astrazioni
 - storicamente sviluppato per le astrazioni procedurali, ma più interessante per le astrazioni sui dati
- Per dimostrare la proprietà dobbiamo usare le **specifiche**
- Tecnica di dimostrazione: **induzione strutturale**
 - 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

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
- Per provarla, dobbiamo considerare il costruttore e i due modificatori



Una proprietà di **IntSet**

- Proprietà: per ogni **IntSet** la sua *size* è ≥ 0
- Caso base: il costruttore

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

- L'insieme vuoto ha cardinalità 0
 - la proprietà vale per il costruttore



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 e della rimozione, vale anche dopo perché l'inserimento può solo incrementare la cardinalità



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



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 nelle **pre-post condizioni**)

Correttezza dell'implementazione

- Non possiamo usare 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”: i 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**a uno 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  
    private Vector els; // la rappresentazione
```

- La funzione di astrazione porta vettori in insiemi

La funzione di astrazione

- La **funzione di astrazione** può essere multi-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) vengono portati nello stesso stato astratto (insieme)
 - $\alpha([1,2]) = \{1,2\} = \alpha([2,1])$
- La funzione di astrazione deve sempre essere definita
 - perché è una parte importante delle decisioni relative all'implementazione
 - sintatticamente, è 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

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



La funzione di astrazione: `IntSet`

```
public class IntSet {  
  // OVERVIEW: un IntSet è un insieme modificabile  
  // di interi di dimensione qualunque  
  // un tipico IntSet è  $\{x_1, \dots, x_n\}$   
  
  private Vector els; // la rappresentazione  
  
  // la funzione di astrazione:  
  //  $\alpha(c) = \{ c.els.get(i).intValue( ) \mid$   
    //  $0 \leq i < c.els.size( ) \}$ 
```

Funzione di astrazione: 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

la **funzione di astrazione di IntSet**:

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

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);  
}
```

Verso IR

- 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 { $x_1, \dots, x_n$ }  
  
    private Vector els; // la rappresentazione  
    // la funzione di astrazione:  
    //  $\alpha(c) = \{ c.els.get(i).intValue( ) \mid$   
        //  $0 \leq i < c.els.size( ) \}$ 
```

Verso IR

- 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  $\{x_1, \dots, x_n\}$ 
```

```
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

L'invariante di rappresentazione

- L'invariante di rappresentazione è un predicato $I : C \rightarrow \text{Bool}$ 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
- La **funzione di astrazione** è una funzione parziale definita solo per stati concreti che soddisfano l'invariante

L'invariante di `IntSet`

```
public class IntSet {  
  // OVERVIEW: un IntSet è un insieme modificabile  
  // di interi di dimensione qualunque  
  // un tipico IntSet è  $\{x_1, \dots, x_n\}$   
  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 non deve essere `null`
- Gli elementi del vettore devono essere `Integer`
- Tutti gli elementi sono distinti

L'invariante di `IntSet`

```
public class IntSet {  
    private Vector els; // la rappresentazione  
    // la funzione di astrazione:  
    //  $\alpha(c) = \{ c.els.get(i).intValue() \mid$   
        //  $0 \leq i < c.els.size() \}$   
  
    // l'invariante di rappresentazione:  
    //  $I(c) = c.els \neq null$  e  
    // per ogni  $i$  tale che  $0 \leq i < c.els.size()$ ,  
    //  $c.els.get(i)$  è un Integer e  
    // per tutti gli  $i, j$  tali che  
    //  $0 \leq i < j < c.els.size()$ ,  
    //  $c.els.get(i).intValue() \neq$   
    //  $c.els.get(j).intValue()$ 
```

Verificare IR

- Può esser verificata in modo dinamico usando un metodo speciale **repOk**
- In alternativa, l'implementazione è accompagnata da una *dimostrazione formale*, che ci garantisce *che tutti i metodi preservano l'invariante* (induzione strutturale)
- La verifica “dinamica” via **repOk** diventa inutile!

Implementare l'invariante

- 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
```

repOK

- Può comunque esser usato da programmi di test per verificare se una implementazione preserva l'invariante
- Può venir usato nell'implementazione 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`



repOK per IntSet

```
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()
```



repOK per IntSet

```
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;
}
```

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

L'invariante di `IntSet`

```
public class IntSet {  
    private Vector els; // la rappresentazione  
  
    // l'invariante di rappresentazione:  
    // l(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'IR perché restituisce un `Vector` vuoto**

Correttezza di **IntSet**

```
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
}
```

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

Correttezza di **IntSet**

```
public class IntSet {  
    private Vector els; // la rappresentazione  
    // l'invariante di rappresentazione:  
    // l(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);  
    }  
}
```

- Il metodo **remove** soddisfa l'invariante perché rimuove **x** da **this**

I 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

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

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

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 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

Esempio: invariante

```
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
```

Una invariante più debole
È soddisfatta ma?

Non basta!!

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

    /
    public int size( ) {
        // EFFECTS: ritorna la cardinalità di this
    return els.size( );
    }
}
```

Non basta per garantire che il numero di elementi del vettore sia la cardinalità dell'insieme

il rep invariant deve garantire che non ci sono elementi duplicati in els