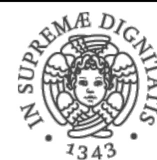
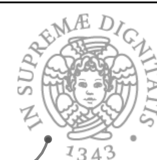


VERIFICA ADT



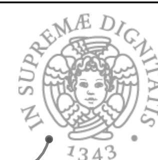
Passi metologici

- Obiettivo: sviluppare codice corretto
  1. Verificare l'invariante di rappresentazione
  2. Verificare che l'implementazione soddisfa le specifiche
  3. Verificare che il codice cliente sia corretto
- Noi consideriamo il passi 1 e 2
  - Alcuni cenni sul passo 3

Verificare Invariante di rappresentazione

- Dimostrare che tutti gli oggetti dell'astrazione soddisfano l'invariante
- Strategia di prova basata su ***Induzione Strutturale***
- **Costruttori ("il caso di base")**:
  - dimostrare che l'invariante vale dopo l'invocazione del costruttore
- **Per tutti gli altri metodi ("passo induttivo")**
  - Assumendo che la proprieta' valga prima della invocazione del metodo (ipotesi induttiva) allora si deve dimostrare che la proprieta' vale anche quanto il metodo termina.

La classe Counter




```
public class Counter {
// Overview: struttura dati modificabile contenitore di un valore intero maggiore o uguale a zero
//Typical element: counter[n]
//
privaite int  count
//abstraction function alpha(c.count) = counter[count]
// representation: I(c.count) ≥ 0

Counter();
//effect: initializza l'oggetto this.count=0

Counter clone();
//effect: restituisce una copia di this

void increment();
// effect: incrementa di 1 il valore di this.count
// post(thi.count) = pre(this.count)+1

Int getValue();
//effect: restituisce il valore this.count
}
```

Dimostrazione RepInv

- Invariante e' soddisfatto dal costruttore
  - this.counter = 0 >= 0
- Passi induttivi: assumiamo che RepInv valga prima della chiamata allore si deve dimostrare che vale quando il metodo termina
- **Metodo clone()**: non modifica il valore di *this.count*, quindi RepInv vale
- **Metodo increment ()** la postcondizione del metodo assicura che *post(this.count) = pre(this.count)+1*, per cui applicando l'induzione abbiamo che *pre(this.count) >=0* che ci permette di concludere.
- **Metodo getValue()** non modifica il valore di *this.count*, quindi RepInv vale
- Morale: RepInv e' soddisfatto

Usare Counter

- Correttezza delle applicazioni che utilizzazano Counter
- Problema: dipende da come viene implementato clone (riferimento all'oggetto)
  - Vedere esempio Java
- Counter e' corretto ma l'applicazione che lo usa no!!
- Problema: Il metodo Counter.clone() e' sottospecificato!!

CharSet

// Overview: A CharSet is a finite mutable set of chars.

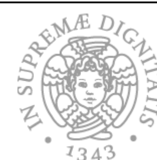
// effects: creates a fresh, empty CharSet
public CharSet ( )

// effects: modifies: this—this(post) = this(pre) U {c}
public void insert (char c);

// effects: modifies: this – this(post) = this(pre) - {c}
public void delete (char c);

//effect: returns: (c ∈ this)
public boolean member (char c);

// effect: returns: cardinality of this
public int size ( );




```
// RepInv: elts != null & elts has no nulls and no duplicates
List<Character> elts;

public CharSet() {
elts = new ArrayList<Character>();
}
public void delete(char c) {
elts.remove(new Character (c));
}
public void insert(char c) {
if (! member(c))
elts.add(new Character(c));
}
public boolean member(char c) {
return elts.contains(new Character(c));
}
…
```

REPINV

Rep invariant: elts ! Nul & elts has no nulls and no duplicates

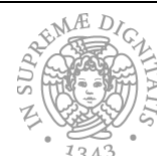
Costruttore

```
public CharSet() {
elts = new ArrayList<Character>();
}
```

Crea l'array elts (!null) vuoto (no null & no duplicate)
RepInv e' soddisfatto

Passi induttivi:
Per tutti i metodi dell'astrazione:

Assumere RepInv
Dimostrare che RepInv vale quando il metodo termina.




```
public void delete(char c) {
elts.remove(new Character(c));
}
```

List.remove ha sue possibili comportamenti

– lascia elts non modificato,
– rimuove l'elemento (definito dal parametro)

Assumiam RepInv,
RepInv puo' diventare falso solamente se
Inseriamo un elemento

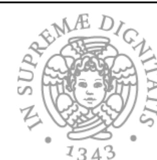
Conclusione: RepInv vale

```
public void insert(char c) {
if (! this.member(c)) elts.add(new Character(c));
}
```

Abbiamo due casi da considerare

1) c $\in$ pre(elts):
elts non viene modificato $\Rightarrow$ RepInv e' preservato

2) c $\notin$ pre(elts)
Nuovo elemento non e' null $\Rightarrow$ RepInv e' preservato



```
public boolean member(char c) {
return elts.contains(new Character(c));
}
```

Dato che il metodo contains non modifica elts
allora anche member non modifica elts

RepInv e' soddisfatto

Domanda: perche' si deve considerare anche il caso di member che e' ovvio (e' un osservatore)?

La prova deve considerare tutti I possibili casi.

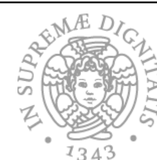
Principio di sostituzione

Tutte le prorieta' garantite dal supertipo devono essere rispettate dal sottotipo

– Nessun indebolimento delle specifiche
– Nessuna elimnazione di metodi

Per I metodi che sono modificati
"weaker precondition" & "stronger postcondition"

Metodi



Metodi modificati

Non deve richiedere maggiori vincoli ("weaker precondition")
La clausola Requires e' al massimo stringente come la clausola Require del supertipo

Deve garantire le stesse proprieta' ("stronger postcondition")
La clausola effect e' stringente quanto quella del supertipo

class Product {
Product recommend(Product ref); }

SaleProduct e' un sottotipo di Product ??

Product recommend(SaleProduct ref); // KO

SaleProduct recommend(Product ref); // OK

Product recommend(Product ref) throws NoSaleException; // KO

```
public class InstrumentedHashSet<E> extends HashSet<E> {

 private int addCount = 0; // count attempted insertions
 public InstrumentedHashSet(Collection<? extends E> c) {
     super(c);
 }
 public boolean add(E o) {
       addCount++;
       return super.add(o);
 }
 public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
       return super.addAll(c);
  }
 public int getAddCount() { return addCount; }
 }
```

Analizziamo il codice:

```
InstrumentedHashSet<String> s = new InstrumentedHashSet<String>();
System.out.println(s.getAddCount()); // 0
s.addAll(Arrays.asList("Programamzione", "2"));
System.out.println(s.getAddCount()); // 4
```

Per capire cosa fa il codice bisogna sapere come e' stata realizzata addAll() in HashSet

HashSet.addAll() chiama add() ⇒**si contano due volte**

Morale: quando si progetta una astrazione bisogna pensare anche che
Potrebbe essere raffinata

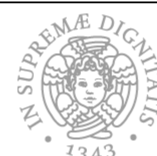
```
public class InstrumentedHashSet<E> {
  private final HashSet<E> s = new HashSet<E>();
  private int addCount = 0;

  public InstrumentedHashSet(Collection<? extends E> c) {
  this.addAll(c);
  }

  public boolean add(E o) {
    addCount++; return s.add(o);
   }

  public boolean addAll(Collection<? extends E> c) {
   addCount += c.size(); return s.addAll(c);
  }

 public int getAddCount() { return addCount; }
// ... and every other method specified by HashSet<E>
  }
```

Problema

InstrumentedHashSet non e' un HashSet

Non si applica il principio di sostituzione

Come lo risolve Java?




```
public class InstrumentedHashSet<E> implements Set<E> {
private final Set<E> s = new HashSet<E>();
private int addCount = 0;
public InstrumentedHashSet(Collection<? extends E> c) {
this.addAll(c);
}
public boolean add(E o) {
addCount++;
return s.add(o);
}
public boolean addAll(Collection<? extends E> c) {
addCount += c.size();
return s.addAll(c);
}
public int getAddCount() { return addCount; }
// ... and every other method specified by Set<E>
```

Valutazioni Pragmatiche

✓Ereditarieta': strumento essenziale per favorire il riuso del codice.

✓Ereditarieta': puo' rompere il principio del mascheramento dell'informazione nella definizione di astrazioni

✓Safe quando tipo e sottotipo sono elementi dello stesso package

✓Morale: progettare avendo in mente che ci possono essere estensioni.