
Debugging di Astrazioni

Astrazioni sui dati: specifica

- Ingredienti tipici di una **astrazione sui dati**
- Un insieme di **astrazioni procedurali** che definiscono tutti i modi per utilizzare un insieme di valori
 - Creare
 - Manipolare
 - Osservare
- **Creatori e produttori**: meccanismi primitivi atti alla programmazione della definizione di nuovi valori
- **Mutator**: modificano il valore (ma non hanno effetto su ==, non operano per effetti laterali)
- **Osservatori**: strumento linguistico per selezionare valori

ADT: specifica e verifica

- **Specifica ADT:** guida all'implementazione
- Ovviamente ADT devono poi essere implementati
- Si deve garantire che la realizzazione soddisfi la specifica
- Due strumenti essenziali
 - Invariante di rappresentazione
 - Funzione di astrazione

Implementazione vs. Specifica

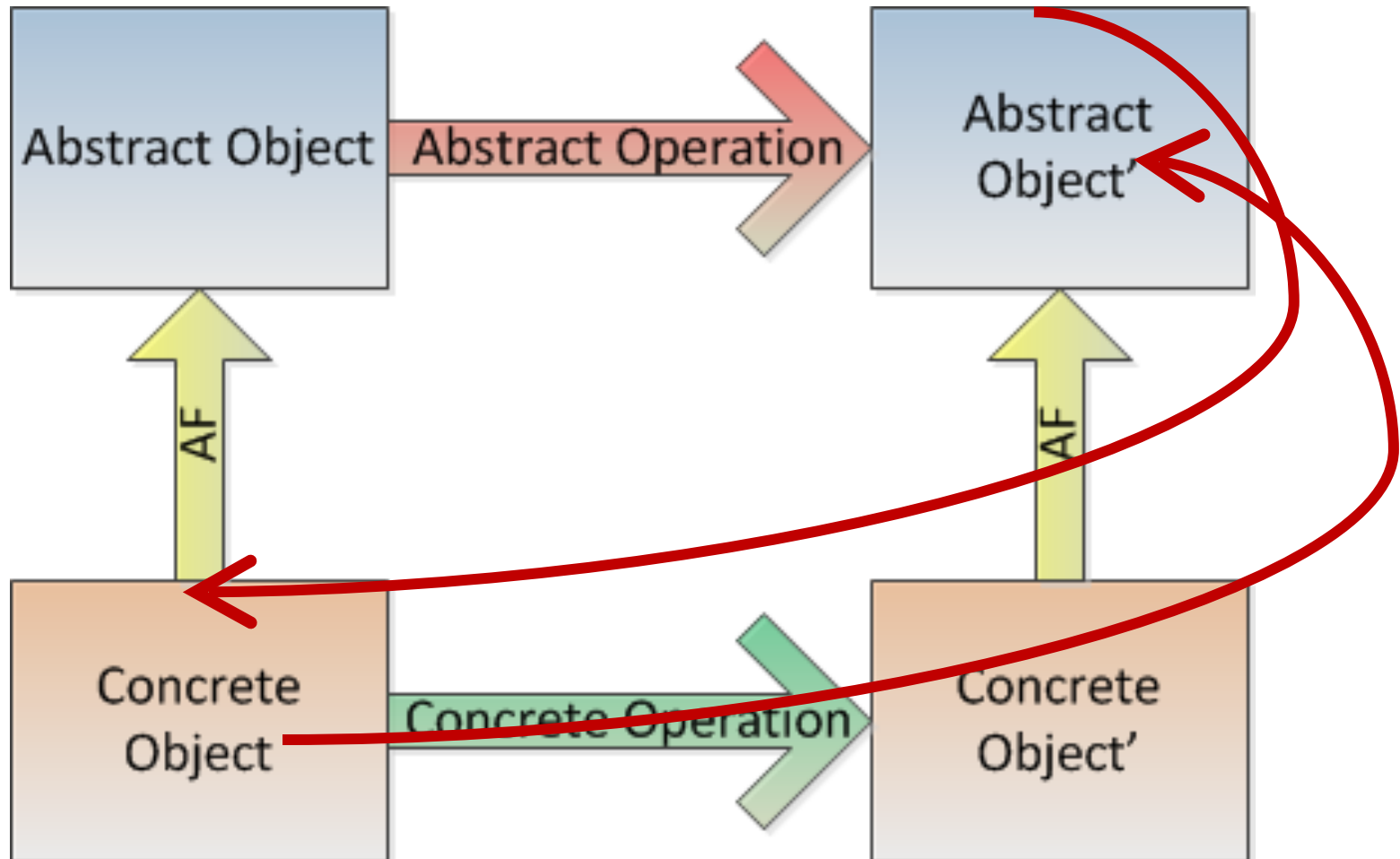
Representation Invariant: Object \rightarrow boolean

- Stabilisce se una istanza è ben formata
- Stabilisce l'insieme concreto dei valori dell'astrazione (ovvero quelli che sono una implementazione dei valori astratti)
- Guida per chi implementa/modifica/verifica l'implementazione delle astrazioni: nessun oggetto deve violare rep invariant

Abstraction Function: Object \rightarrow abstract value

- Stabilisce come interpretare la struttura dati concreta della implementazione
- È definita solamente sui valori che rispettano l'invariante di rappresentazione
- Guida per chi implementa/modifica l'astrazione: ogni operazione deve fare “la cosa giusta” con la rappresentazione concreta

Implementazione vs. Specifica



Esempio: CharSet

```
// Overview: CharSet un insieme finito modificabile di
// Characters

// @effects: crea un CharSet nuovo e vuoto
public CharSet( ) {...}

// @modifies: this
// @effects: thispost = thispre U {c}
public void insert(Character c) {...}

// @modifies: this
// @effects: thispost = thispre \ {c}
public void delete(Character c) {...}

// @effects: return (c ∈ this)
public boolean member(Character c) {...}

// @effects: return cardinalita' di this
public int size( ) {...}
```

CharSet : rappresentazione?

```
class CharSet {
    private ArrayList<Character> elts =
        new ArrayList<Character>( );

    public void insert(Character c) {
        if (elts.add(c)){return;}
    }
    public void delete(Character c) {
        int i = elts.indexOf(c);
        if (i > -1) elts.remove(i);
    }
    public boolean member(Character c) {
        return elts.contains(c);
    }
    public int size( ) {
        return elts.size( );
    }
}
```

CharSet: implementazione?

```
class CharSet {
    private ArrayList<Character> elts =
        new ArrayList<Character>( );

    public void insert(Character c) {
        if (elts.add(c)){return;}
    }
    public void delete(Character c) {
        int i = elts.indexOf(c);
        if (i > -1) elts.remove(i);
    }
    public boolean member(Character c) {
        return elts.contains(c);
    }
    public int size( ) {
        return elts.size( );
    }
}
```

Dove è nascosto l'errore?

Charset: facciamo un test!!

```

class CharSet {
    private ArrayList<Character> elts =
        new ArrayList<Character>( );

    public void insert(Character c) {
        if (elts.add(c)){return;}
    }
    public void delete(Character c) {
        int i = elts.indexOf(c);
        if (i > -1) elts.remove(i);
    }
    public boolean member(Character a) {
        return elts.contains(a);
    }
    public int size( ) {
        return elts.size();
    }
}

CharSet s = new CharSet( );
Character a = new Character('a');
s.insert(a);
s.insert(a);
s.delete(a);
if (s.member(a))
    System.out.print("wrong");
else
    System.out.print("right");

```

Cerchiamo l'errore

- Primo tentativo: **delete** è sbagliata
 - controlla l'appartenenza ma rimuove tutte le occorrenze?
- Secondo tentativo: **insert** è sbagliata
 - non dovrebbe inserire un carattere quando è già presente
- Come operiamo?
 - se avessimo usato **representation invariant** avremmo evitato l'errore
 - il codice ben documentato e gli strumenti di specifica formale ci aiutano nell'operazione di individuazione e rimozione dell'errore

Invariante di rappresentazione

```
• class CharSet {  
    // Rep invariant:  
    // elts non contiene elementi null e non  
    // contiene duplicati  
    private ArrayList<Character> elts =  
        new ArrayList<Character>( );    ...
```

Quindi formalmente (con gli strumenti di LPP):

```
    ∀ indice i di elts . elts.elementAt(i) ≠ null e  
    ∀ indice i, j di elts .  
        i ≠ j ⇒  
    ¬ (elts.elementAt(i).equals(elts.elementAt(j)))
```

Ora localizziamo l'errore

```
// Rep invariant:
//   elts: no null e no duplicati

public void insert(Character c) {
    if (elts.add(c)) {return;}
}

public void delete(Character c) {
    int i = elts.indexOf(c);
    if (i > -1) elst.remove(c);
}
```

Come si fa il debugging



L'idea che intendiamo perseguire è la seguente:

Progettate del codice in modo tale che tutte le operazioni di “bug-checking” siano implementate utilizzando come guida l'invariante di rappresentazione

Verifica del rep invariant

Idea derivata dalle tecniche di prova: controllare ingresso e uscita dai metodi

```
public void delete(Character c) {
    checkRep( );
    int i = elts.indexOf(c);
    if (i > -1) elst.remove(c);
    checkRep( );
}

...
/** elts no duplicati. */
private void checkRep( ) {
    ...
}
```

defensive programming

- Assunzione: programmare è un processo di tipo “**trial and error**”
- Progettare del codice in modo tale che
 - alla chiamata dei metodi
 - ✓ verifica rep invariant
 - ✓ verifica pre-condizioni
 - all’uscita del metodo
 - ✓ verifica rep invariant
 - ✓ verifica post-condizioni
- Verificare rep invariant = verificare la presenza di errori
- Ragionare sul rep invariant = evitare di fare errori

Un esempio sempre **CharSet**

Aggiungiamo un metodo molto utile a **CharSet**

```
// restituisce una lista degli elementi che  
// appartengono a this.
```

Implementazione

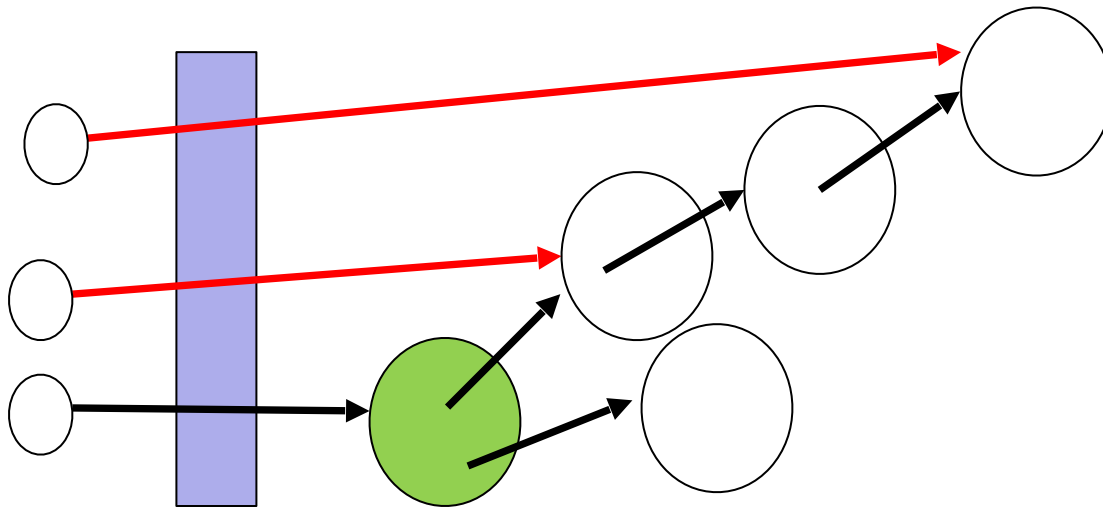
```
// Rep invariant: elts no null e no dupl.  
public List<Character> getElts () { return elts; }
```

L'implementazione di **getElts** preserva rep invariant?

Ma espone la **rappresentazione** !!!

private ... non basta

- L'uso di **private** potrebbe non bastare
 - Aspetto chiave: **aliasing di struttura mutabili all'interno e all'esterno della astrazione**



Esporre la rappresentazione

Consideriamo un cliente (sempre di **CharSet**)

```
CharSet s = new CharSet( );
Character a = new Character('a');
s.insert(a);
s.getElts( ).add(a); // usiamo add in modo liberale
s.delete(a);
if (s.member(a)) ...
```

- Abbiamo una **esposizione della rappresentazione** con un accesso indiretto (tramite il metodo **getElts**)
- Problema: bug da evitare (**vedremo gli iteratori in seguito**)
 - progettare l'astrazione in modo da evitare questo problema
 - progettare dei test con clienti "malevoli": usare valori mutabili per capire cosa avviene nel dettaglio

Come si evita?

- Per evitare l'esposizione della rappresentazione una prima tecnica è quella di fare copie dei dati che oltrepassano la barriera dell'astrazione
 - copia in [parametri che diventano valori della rappresentazione]
 - copia out [risultati che sono parte dell'implementazione]
- Esempio: **Point** ADT modificabile

```
class Line {
    private Point s, e;
    public Line(Point s, Point e) {
        this.s = new Point(s.x,s.y);
        this.e = new Point(e.x,e.y);
    }
    public Point getStart( ) {
        return new Point (this.s.x,this.s.y);
    }
    ...
}
```

deep copying

- Una copia **shallow** (operazioni sui puntatori) non è sufficiente a causa dell'aliasing !!!
- Analizzare questo codice

```
class PointSet {  
    private List<Point> points = ...  
    public List<Point> getElts ( ) {  
        return new ArrayList<Point>(points) ;  
    }  
}
```

Una seconda soluzione

- Usare strutture dati non modificabili
- Esempio: **Point** (non modificabile)

```
class Line {
    private Point s, e;
    public Line(Point s, Point e) {
        this.s = s;
        this.e = e;
    }
    public Point getStart( ) {
        return this.s;
    }
    ...
}
```

Strutture non modificabili

- Vantaggi
 - l'aliasing non è un problema
 - non è necessario fare copie
 - rep invariant non può essere “rotto”
- Richiede tuttavia scelte di programmazione differenti

```
void raiseLine(double deltaY) {
    this.s = new Point(s.x, s.y+deltaY);
    this.e = new Point(e.x, e.y+deltaY);
}
```
- Classi immutabili nella libreria: **String**, **Character**, **Integer**, . . .

Alternative: TDA modificabili

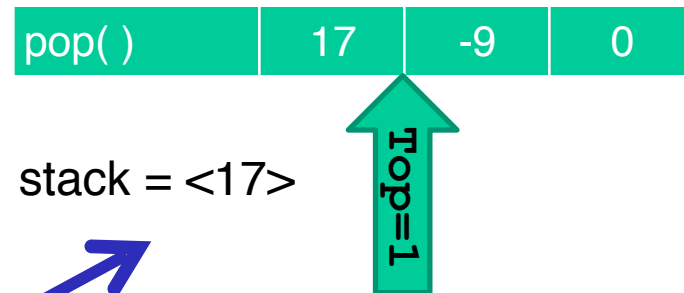
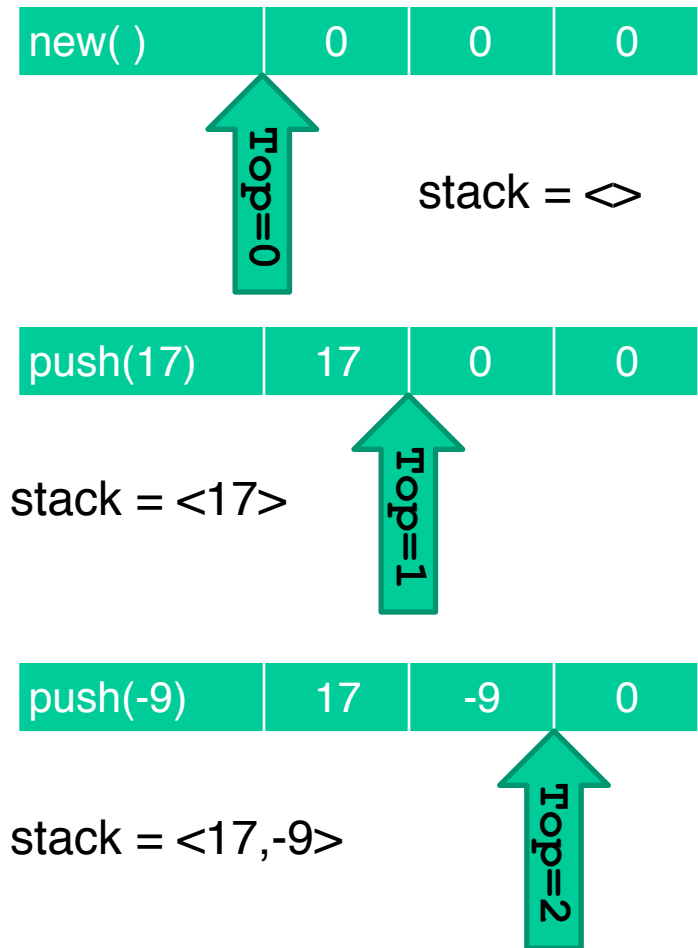
```
// returns: ...
public List<Character> getElts () { // versione 1
    return new ArrayList<Character>(elts); } //copy out!
}

public List<Character> getElts () { // versione 2
    return Collections.unmodifiableList<Character>(elts);
}
```

JavaDoc: `Collections.unmodifiableList`:

Returns an unmodifiable view of the specified list. This method allows modules to provide users with "read-only" access to internal lists. Query operations on the returned list "read through" to the specified list, and attempts to modify the returned list... result in an `UnsupportedOperationException`.

Esempio: AF per Stack



Stati astratti
stack = <17> = <17>

Stati concreti
 <[17,0,0], top=1>
 ≠
 <[17,-9,0], top=1>

AF è una funzione
 L'inverso di AF non lo è

Riassunto finale

Rep invariant

- Quali sono i valori concreti che rappresentano valori astratti

Abstraction function

- Per ogni valore concreto restituisce il corrispondente valore astratto

Obiettivo comune: sono entrambe indispensabili per controllare la correttezza dell'astrazione

Di solito, la documentazione fa vedere solamente il rep invariant