DEBUGGING DI DATA ABSTRACTION

1

Astrazioni sui dati: specifica

- Ingredienti tipici di una astrazione sui dati
- Un insiene di *astrazioni* procedurali che definiscono tutti modi per utilizzare un insieme di *valori*
- Creare
- Manipolare
- Osservare

- Ceatori e produttori: meccanismi primitivi atti alla programmazione della definizione di nuovi valori
- Mutators: modificato il valore (ma non hanno effetto su **==,** non operano per effetti laterali)
- Osservatori: strumento linguistico per selezionare valori

2

Implementazione vs Specifica

Representation Invariant: Object → boolean

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

Abstraction Function: Object → abstract value

- Stabilisce come interpretare la struttura dati concreta della implementazione
- E' 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

3

Esempio: CharSet


```
// Overview: CharSet insieme finito modificabile di
// Characters
// @effects: crea un CharSet nuovo e vuoto
public CharSet() {…}
// @modifies: this
// @effects: this_post = this_pre + {c}
public void insert(Character c) {…}
// @modifies: this
// @effects: this_post = this_pre - {c}
public void delete(Character c) {…}
// @effects: return (c ∈ this)
public boolean member(Character c) {…}
// @effects: retuucardinalita' di this (this.size())
public int size() {…}n
```

4

CharSet: implementazione?

```
class CharSet {
  private List<Character
      new ArrayList<Cha
  public void insert(Cha
     elts.add(c);
  }
  public void delete(Cha
     int I = elts.index
     if I > -1 elst.remo
  }
  public boolean member
     return elts.contain
  }
  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");
```

Dove e' nascosto l'errore?

Cerchiamo l'errore

- *Primo tentativo:* **`delete`** e' sbagliata
  - Controlla l'appartenenza ma rimuove tutte le occorrenze?
- *Secondo tentativo* **`insert`** e' sbagliata
  - Non dovrebbe inserire un carattere quando e' gia' presente?
- Come operiamo?
  - Utilizziamo representation invariant per muoverci e eliminare l'errore
  - E' il codice ben documentato e gli strumenti di specifica formale che ci aiutano nel'operazione di individuazione e rimozione dell'errore

Invariante di rappresentazione

- Rep Invariatiant:

```
class CharSet {
 // Rep invariant:
 // elts non contiene elementi null e non
 //Contiene duplicati
    private List<Character> elts = …
    …
```

Possiamo scriverlo anche formalmente (con gli strumenti di LPP):

∀ indices i di elts . elts.elementAt(i) ≠ null

∀ indices i, j di elts .

i ≠ j ⇒ ¬ elts.elementAt(i).equals(elts.elementAt(j))

Notare che ArrayList ammette null !!!

7

Ora localizziamo l'errore

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

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

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

8

Come si fa il debugging

L'idea e lo strumento intendiamo definire e' 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 all'ingresso e uscita dei metodi

```
public void delete(Character c) {
  checkRep(); //altrnativa invocare repOK()
  int I = elts.indexOf(c);
      if I > -1 elst.remove(;

  // Come garantire che venga sempre invovata?
  // (usiamo un blocco finally)
  checkRep();
}
…
/** elts no duplicati. */
private void checkRep() {
  :;
  }
}
```

defensive programming

- Assunzione: programmare e' un processo di tipo "trial and error"
- Progettare del codice in modo tale che
  - Alla chiamata dei metodi:
    - ✓ Verifica rep invariant
    - ✓ Verifica precondizioni
  - All'uscita del metodo
    - ✓ Verifica rep invariant
    - ✓ Verifica post condizioni
- Verificare rep invariant = verificare la presenza di errori
- Raggionare sul rep invariant = evitare di fare errori.

11

Sempre CharSet

Aggiungiamo un metodo a **`CharSet`**

```
// restituisce una lista degli elementi che
appartengono a this.
public List<Character> getElts();
```

Implementazione:

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

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

Boh!!….

12

Esporre la rappresentazione

Consieriamo un cliente (sempre di **CharSet**):

```
CharSet s = new CharSet();
Character a = new Character('a');
s.insert(a);
s.getElts().add(a);
s.delete(a);
if (s.member(a)) …
```

- Abiamo un esposizione della rappresentazione con un accesso indiretto (tramite il metodo getElts)
- Problema: bug da evitare.
  - 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 e' quella di fare copie dei dati che oltepassono 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 e' sufficiente a causa dell'aliasing !!!
- Analizzare questo codice?

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

15

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;
   }
   …
```

16

Strutture non modificabili

- Vantaggi
  - Aliasing non e' un problema
  - Non e' necessario fare copie
  - Rep invariant non puo' 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 immatabili delle librerie di Java: **`String`**, **`Character`**, **`Integer`**, …

17

Ancora il caso **`getElts`**


```
class CharSet {
  // Rep invariant: elts: no null e no dupl.
  private List<Character> elts = …;

  // returns: elts nell'insieme corrente
  public List<Character> getElts() {
    return new ArrayList<Character>(elts); //copy out
  }
  …
}
```

18

Alternative


```
// 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`**.

19

Good news direbbero negli states!!


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

- Clienti non possono spezzare il rep invariant
- Se la lista e' di dimensioni elevate e' piu' efficiente della tecnica del copy out
- Si usano librerie standard (sempre una buona cosa)

20

bad news
direbbero sempre negli states

```
public List<Character> getElts() { // versione 1
 return new ArrayList<Character>(elts);//copy out!
}
public List<Character> getElts() { // versione 2
 return Collections.unmodifiableList<Character>(elts);
}
```

Le due implementazioni sono differenti!!!

- Entrambe permettono di evitare di rompere il rep invariant
- Entrambe restituiscono una lista di elementi

Ma …. :
```
xs = s.getElts();
s.insert('a');
xs.contains('a');
```

La versione 2 permette di *osservare* la rappresentazione

21

Implementazione vs Specifica

Representation Invariant: Object → boolean

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

Abstraction Function: Object → abstract value

- Stabilisce come interpretare la struttura dati convreta della implementazione
- E' 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

22

Rep inv. vincola la struttura

Implementazione di **insert** che preserva rep invariant:

```
public void insert(Character c) {
  Character cc = new Character(encrypt(c));
  if (!elts.contains(cc))
    elts.addElement(cc);
}
public boolean member(Character c) {
  return elts.contains(c);
}
```

Il programma presenta dei comportamenti non adeguati

```
CharSet s = new CharSet();
s.insert('a');
if (s.member('a'))
    ...
```


La funzione di astrazione (AF)

La abstraction function associa la rappresentazione concreta ai valori astratti.

AF: Object → abstract value

AF(CharSet this) = { c | c appartiene a this.elts }

"insieme dei caratteri in this.elts"

- Non e' eseguibile, e' un "valore" concettuale dellla astrazione.
- Tuttavia la funzione di astrazione ci permette di ragionare sulle modalita' in cui i metodi operano in termini della visione astratta (che hanno in clienti)

Il caso **`insert`**

La specifica di insert:

```
// modifies: this
// effects: this_post = this_pre U {c}
public void insert (Character c) {…}
```

La AF ci dice effettivamente cosa significa il rep inv.

AF(CharSet this) = { c | c appartenenti a this.elts }

Invochiamo **`insert`**:

All'ingresso del metedo vale AF($this_{pre}$) ≈ $elts_{pre}$

All'uscita AF($this_{post}$) = AF($this_{pre}$) ∪ {encrypt('a')}

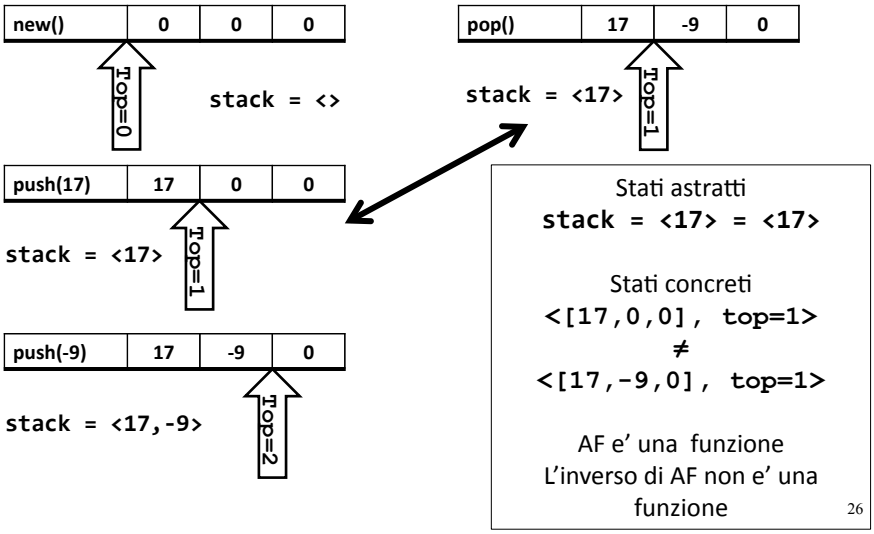
Meglio usare questa AF alternativa

AF(this) = { c | encrypt(c) appartenenti a this.elts }

= { decrypt(c) | appartenenti a this.elts }

25

Esempio AF per Stack



Stati astratti

`stack = <17> = <17>`

Stati concreti

`<[17,0,0], top=1>`

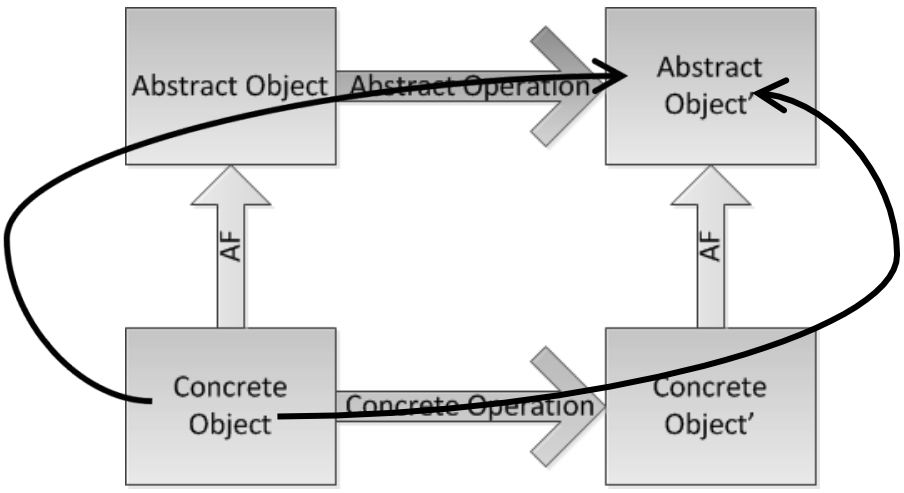
≠

`<[17,-9,0], top=1>`

AF e' una funzione

L'inverso di AF non e' una funzione

26



27

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 insispensabili per controllare la correttezza dell'astrazione

Di solito la documentazione fa vedere solamente il rep invariant

28