Invariante di Rappresentazione

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

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

Implementazione vs. Specifica

Representation Invariant: Object → 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 → 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

Esempio: CharSet

```
// Overview: CharSet insieme finito modificabile di
// Characters {c1, …. cn}
// @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: return cardinalita' di this (this.size( ))
public int size( ) {...}
```


CharSet: implementazione?

```
class CharSet {
  private List<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) elst.remove(i);
  }
  public boolean member(Character c) {
     return elts.contains(c);
  }
  public int size( ) {
     return elts.size( );
  }
}
```

CharSet: implementazione?

```
class CharSet {
  private List<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) elst.remove(i);
  }
  public boolean member(Character c) {
     return elts.contains(c);
  }
  public int size( ) {
     return elts.size( );
  }
}
```

Dove è nascosto l'errore?

CharSet: implementazione?

```
class CharSet {
  private List<Character> elts =
       new ArrayList<Character>( );
  public void insert(Character c) {
     if (elts.add(c)
  }
  public void delete
     int i = elts.in
      if (i > -1) el
  }
  public boolean mem
     return elts.con
  }
  public int size( )
     return elts.siz
  }
}
```

```
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");
```

CharSet: implementazione?

```
class CharSet {
  private List<Character> elts =
       new ArrayList<Character>( );
  public void insert(Character c) {
     if (elts.add(c)
  }
  public void delete
     int i = elts.in
      if (i > -1) el
  }
  public boolean mem
     return elts.con
  }
  public int size( )
     return elts.siz
  }
}
```

```
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 è nascosto l'errore?

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

- Terzo tentatvo: **`insert`** non contralla se il parametro è null

- Come operiamo?
  - utilizziamo representation invariant per muoverci e eliminare 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 List<Character> elts = ...
       ...
  ```

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

  $\forall$ indice i di elts . elts.elementAt(i) $\neq$ null

  $\forall$ indice i, j di elts .

  $i \neq j \Rightarrow \neg$ elts.elementAt(i).equals(elts.elementAt(j))

  Notare che ArrayList ammette null !!!

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);

   // Come garantire che venga sempre invocata?
   // (usiamo un blocco finally)
   checkRep( );
}
...
/** elts no duplicati. */
private void checkRep( ) {
  // codifica dell'invariante ... Throws exception in
   // caso di violazione
}
```

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**

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?

Mah?!….

Esporre la rappresentazione

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

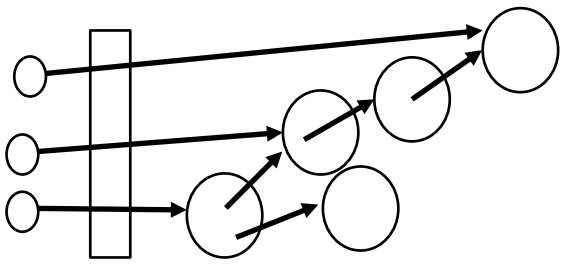
```
CharSet s = new CharSet( );
Character a = new Character('a');
s.insert(a);
s.getElts( ).add(a); // usiamo add in modo liberale
s.delete(a);
if (s.member(a)) ...
```

• Abbiamo una esposizione della rappresentazione con un accesso indiretto (tramite il metodo getElts)

• Problema: bug da evitare
- progettare l'astrazione in modo da evitare questo problema
- progettare dei test con clienti "malevoli": usare valori mutabili per capire cosa avviene nel dettaglio

`private` … non basta

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



Come si evita?

- Per evitare l'esposizione della rappresentazione una prima tecnica è 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 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`**, . . .

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
   }
   ...
}
```


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`**.

Some good news

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

- i clienti non possono spezzare il rep invariant
- se la lista è di dimensioni elevate è più efficiente della tecnica del copy out
- si usano librerie standard (sempre una buona cosa)

Some bad news

```
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!!

Implementazione vs. Specifica

Representation Invariant: Object → 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**

Abstraction Function: Object → 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

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

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 è eseguibile: è un "valore" concettuale della astrazione
- tuttavia, la funzione di astrazione ci permette di ragionare sulle modalità con le quali i metodi operano in termini della visione astratta (che hanno i clienti)

Il caso **`insert`**

La specifica di **`insert`**

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

La AF ci dice effettivamente cosa significa il rep invariant

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

Invochiamo **`insert`**

All'ingresso del metodo vale $AF(this_{pre}) \approx elts_{pre}$

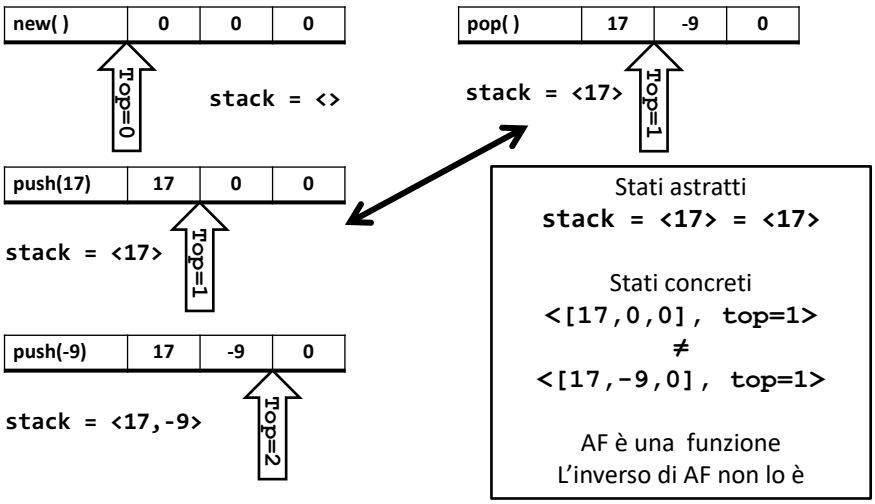
All'uscita $AF(this_{post}) = AF(this_{pre}) \cup \{encrypt('a')\}$

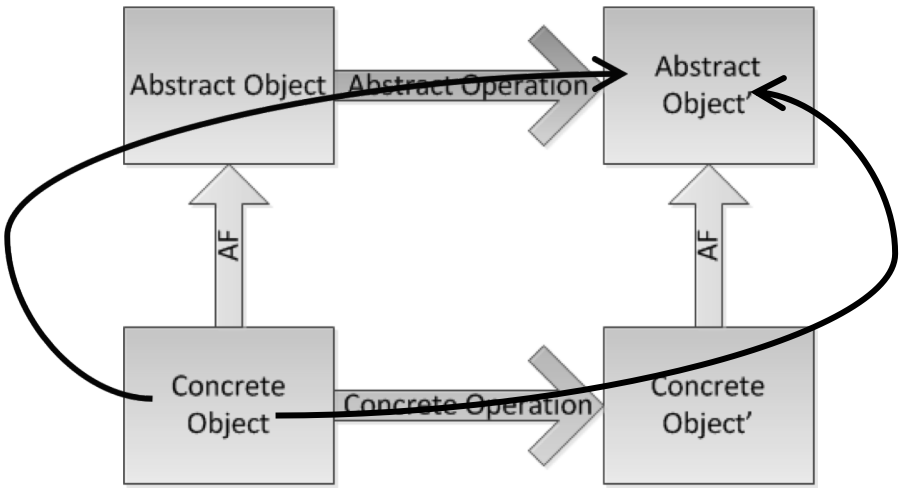
Meglio usare questa AF alternativa

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

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

Esempio: AF per Stack





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