

Astrazioni sui dati

Abstraction by specification sui dati

- Un nuovo tipo di dato per cui siano stati specificati valori e operazioni possibili (tipo di dato astratto o tipo)
- Astraendo dai dettagli di rappresentazione dei valori e d'implementazione delle operazioni, il resto del programma dipende solo dalla specifica del tipo!
- La signature del metodo (o il costruito interface) definiscono solo la sintassi dei metodi

JML: visibilità e metodi puri

- Un metodo è **puro** se non ha effetti collaterali
 - Per questi assignable vale \nothing, ma preferiamo scrivere “pure” perchè ci consente di richiamare il metodo in JML
 - Anche i costruttori possono essere dichiarati “pure”
 - Significa che possono inizializzare gli attributi dichiarati nella classe, ma non possono modificare nient’altro
- Nelle pre e post-condizioni di un metodo pubblico possono comparire solo gli elementi pubblici del metodo e della classe
 - In particolare, i parametri formali e \result, ma anche metodi pubblici puri o attributi pubblici
 - ...ma **non** i metodi pubblici che non sono dichiarati **puri**!

Esempio di metodo puro

- Sono metodi certamente privi di side effect
- Un metodo `size()` che restituisce il numero di elementi di un contenitore è puro

```
//@ ensures (*\result è cardinalità di this *)  
public int /*@ pure @*/ size(){}  

```

Categorie di operazioni

- **Creatori:** producono nuovi oggetti
 - Sono tutti dei costruttori
 - ...ma non tutti i costruttori sono creatori
 - Quando hanno oggetti del proprio tipo come argomenti sono invece **produttori**
- **Produttori:** dati in input oggetti del proprio tipo creano altri oggetti del proprio tipo
- **Modificatori:** modificano oggetti del proprio tipo
- **Osservatori:** dati in input oggetti del proprio tipo restituiscono risultati di altri tipi
 - Tipicamente dichiarati “puri” in JML

IntSet

- Tipo: “insieme di numeri interi” (IntSet)
- Valori possibili:
 - Tutti gli insiemi finiti di numeri interi (di cardinalità qualsiasi) (NB: un insieme non contiene elementi ripetuti)
 - Esempi: \emptyset , {1,10, 35}, {7}, {50, -1, 48, 18, 33, 6, 15, 1000}
- Operazioni possibili su un dato insieme I
 - costruzione: crea I e lo inizializza come insieme vuoto
 - void insert (int x): aggiunge x agli elementi di I
 - void remove (int x): elimina x da I
 - boolean isIn (int x): dice se x appartiene a I o meno
 - int size(): restituisce la cardinalità di I
 - int choose(): restituisce un qualsiasi elemento di I

Mutabilità

- Un tipo è mutabile se ha dei metodi modificatori, altrimenti è immutabile
- **Non** è una proprietà dell'implementazione ma della specifica!
 - Il metodo è modificatore già nella specifica
- Come decidere se mutabile?
 - Meglio per oggetti che modellano entità del mondo reale: automobili, persone, ecc. che cambiano dinamicamente
 - Usato spesso per i contenitori (array, insiemi, ...)

Osservatori

```
public class IntSet {  
  // Insiemi di interi illimitati e modificabili;  
  // Esempio: {1, 2, 10, -55}  
  // Osservatori:  
  
  //@ ensures (*\result == true*) <==>  
  //@  (*x è fra gli elementi di this*);  
  public /*@ pure @*/ boolean isIn (int x){}  
  
  //@ ensures (*\result è cardinalità di this *);  
  public /*@ pure @*/ int size(){}  
  
  //@ ensures this.isIn(\result) && this.size()>0;  
  //@ signals (EmptyException) this.size()==0;  
  public /*@ pure @*/ int choose() throws EmptyException {  
  }  
}
```

Gli observer spesso non sono facilmente definibili con il sottoinsieme di JML che consideriamo

Costruttori e modificatori

// Costruttori:

//@ensures this.size()==0;

public IntSet() {// inizializza *this* come insieme vuoto}

// Modificatori:

//@ ensures this.isIn(x) &&

//@ (\forall int y; x!=y; \old(isIn(y)) <==> isIn(y));

public void insert(int x) {//inserisce x in this}

//@ ensures !this.isIn(x) &&

//@ (\forall int y; x!=y; \old(isIn(y)) <==> isIn(y));

public void remove(int x) {//rimuove x da this}

Uso degli ADT

- La specifica deve essere **sufficiente** per potere utilizzare l'astrazione senza conoscere l'implementazione
- Esempio:

```
//@ ensures (*restituisce IntSet con tutti e soli gli  
//@ elementi di array a*);  
public static IntSet getArray (int[] a) throws NullPointerException {}
```
- La funzione può essere facilmente implementata usando solo i metodi pubblici di IntSet

```
IntSet s = new IntSet(); // Dalla specifica, s diventa insieme vuoto  
for (int i=0; i<a.length; i++)  
    s.insert(a[i]); // Da specifica, a[i] è inserito in s  
return s;
```

Poly

- Tipo: “Polinomi a coefficienti interi” (Poly)
- Valori possibili:
 - esempio: $1 + 2x + 3x^3 - 2x^4 \dots$
- Costruzione: come costruire un polinomio? Convieni partire dal polinomio più semplice: il monomio
 - Monomio ha coeff intero e grado ≥ 0
- I polinomi si ottengono poi combinando monomi con + - *
 - Costruzione: crea monomio di grado e coefficiente dati
 - Come caso particolare, il polinomio zero (grado 0 coeff 0)
 - `public int degree()` : restituisce il grado del polinomio
 - `public int coeff(int d)` restituisce il coefficiente del termine di grado d
 - `public Poly add(Poly q)` restituisce la somma di this e q
 - `public Poly sub(Poly q)` restituisce la differenza di this e q
 - `public Poly mul(Poly q)` restituisce il prodotto di this e q
 - ...

Specifica

```
/*@ pure @*/ public class Poly {  
  
    // Osservatori:  
    //@ ensures (*\result è grado del polinomio *);  
    public int degree(){}  
  
    //@ requires d>=0;  
    //@ ensures (d>degree() ? \result ==0 : (*\result è coeff. del termine di grado d *));  
    public int coeff(int d){}  
  
    // Costruttori:  
    //@ ensures this.degree()==0 && this.coeff(0)==0;  
    public Poly(){} // Costruisce polinomio zero  
  
    //@ ensures n>=0 && this.coeff(n) == c && this.degree() == n &&  
    //@ (\forall int i; 0<=i && i< this.degree(); this.coeff(i) ==0);  
    //@ signals(NegativeExponentException e) n<0;  
    public Poly(int c, int n) throws NegativeExponentException{}
```

Produttori

- I costruttori di Poly sono in grado di costruire solo monomi!
Come costruire polinomi più complessi? Con i produttori!

```
//@ ensures q != null && (* \result == this + q *);  
//@signals (NullPointerException e) q == null;  
public Poly add(Poly q) throws NullPointerException {...}
```

```
//@ ensures q != null && (* \result == this - q *);  
//@signals (NullPointerException e) q == null;  
public Poly sub(Poly q) throws NullPointerException {...}
```

```
//@ ensures q != null && (* \result == this * q *);  
//@signals (NullPointerException e) q == null;  
public Poly mul(Poly q) throws NullPointerException {...}
```

```
//@ ensures (* \result == -this *);  
public Poly minus() {...}
```

Uso della classe Poly

- La classe Poly può essere usata senza conoscerne l'implementazione ma solo la specifica
- Ed esempio, per scrivere programma che legge un Poly da input

```
public static void main (String argv[]) throws java.io.IOException, NegativeExponentException {  
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));  
    System.out.println("Inserisci grado:");  
    Poly p = new Poly(); // Polinomio p inizializzato a zero  
    int g = Integer.parseInt(in.readLine());  
    for (int i=0; i<=g; i++) {  
        System.out.println("Inserisci coeff. termine grado: " + i + " = ");  
        int c = Integer.parseInt(in.readLine());  
        p = p.add(new Poly(c,i)); // Il Poly p diventa p + cx^i  
    }  
    .... qui si puo' usare il polinomio p.....  
}
```

Oggetto astratto vs. oggetto concreto

- La specifica di un'astrazione descrive un “oggetto astratto”
 - Es. per un IntSet
 - L'oggetto astratto è “un insieme di interi”
 - L'oggetto concreto potrebbe essere un ArrayList, un albero binario, o altra struttura dati usata per l'implementazione della classe IntSet
- L'implementazione di un'astrazione descrive un “oggetto concreto”
 - Es. per Poly:
 - L'oggetto astratto è “un polinomio...”
 - L'oggetto concreto potrebbe essere un array o una lista delle adiacenze o un'altra struttura dati comoda
- I due concetti sono **differenti** e non vanno confusi
- Le eventuali modifiche al codice potrebbero modificare, per esigenze implementative, la struttura dati dell'oggetto concreto, lasciando immutato l'oggetto astratto

Gli observer non bastano

- In generale lo stato concreto degli oggetti della classe non è visibile (se attributi sono privati)
 - Esso non può (e non deve) essere usato nella specifica
- Lo stato astratto invece è osservabile attraverso metodi observer (puri) e quindi usabile nella specifica
- Però spesso i metodi observer non sono sufficienti a descrivere pre/post/...., perché non catturano completamente lo stato astratto
- Questo accade spesso (non sempre!) con le collezioni

Specifica di push e pop per Pila

```
public class Pila<T> {  
  
    //@ensures (*\result è numero di elementi in this *);  
    public int size(){..  
  
    //@ requires size()>0;  
    //@ ensures (* \result è cima della pila *)  
    public /*@ pure @*/ T top();  
  
    //@ ensures size()==0;  
    public Pila() {..  
  
    //@ensures size()==\old(size()+1)&&top()==x && (*elementi sotto top invariati*)  
    public void push(T x) {...}  
  
    //@ requires 0 < size();  
    //@ ensures size() == \old(size()-1) && (* elimina elemento da cima della pila *)  
    public void pop(){..  
  
}
```

Come esprimerlo
senza conoscere i
dettagli interni
della struttura
dati?

Oggetto astratto tipico

- La specifica della Pila è comprensibile, ma non è completamente formalizzata
 - Gli osservatori non bastano per definire il resto della specifica
- Per formalizzarla, si può definire nella sola specifica un **oggetto astratto tipico** (OAT), che definisce in modo preciso l'oggetto astratto definito dalla specifica della classe
 - L'OAT è di solito una lista, un set, o una stringa
- Le operazioni possono essere descritte come effetto sull'OAT
- L'OAT può essere utilizzato per specificare, in modo più o meno formale, qualunque classe

Esempio

- Pila (LIFO) di interi
 - Oggetto tipico: una sequenza di interi $x_1x_2...x_n$
 - L'operazione `top()` a partire da $x_1x_2...x_n$ restituisce x_n
 - L'operazione `size()` a partire da $x_1x_2...x_n$ restituisce n
 - L'operazione `push(k)` a partire da $x_1x_2...x_n$ trasforma lo stack in $x_1x_2...x_n$: ossia $x_1 x_2 ... x_n+k$
 - L'operazione `pop` a partire da $x_1x_2...x_n$ genera $x_1x_2...x_{n-1}$
- Si può introdurre esplicitamente nei commenti un attributo OAT, utilizzabile solo per la specifica

Specifica con oggetto tipico

Dichiariamo che la specifica si baserà su un OAT

```
public class Pila<T> {  
    //@spec_public List<T> oat;  
    // Oggetto tipico OAT è <x_1, x_2, ..., x_n>, n>=0, x_i in T, x_n è il top  
  
    //@ensures \result == oat.size();  
    public /*@ pure @*/ int size(){}  
  
    //@requires size()>0;  
    //@ ensures \result== oat.get(oat.size()-1);  
    public /*@ pure @*/ T top() throw EmptyException;  
  
    //@ ensures size()==0;  
    public Pila() { .. }  
  
    //@ensures size()==\old(size()+1) && top()==x && \old(oat).equals(oat.sublist(0,oat.size()-2));  
    public void push(T x) {...}  
  
    //@requires size()>0;  
    //@ensures size()==\old(size()-1) && oat.equals(\old(oat.sublist(0,oat.size()-2))  
    public void pop(){...}  
}
```

Questa specifica si basa sulla definizione dell OAT!

Adeguatezza di un tipo (1)

- Un tipo è **adeguato** se fornisce operazioni sufficienti perché il tipo possa essere utilizzato semplicemente ed efficientemente
 - Esempio: se `IntSet` avesse solo `insert` e `remove`, non si potrebbe verificare `membership`
- Semplice verifica di adeguatezza
 - Se tipo è **mutable** almeno: creatori, osservatori e modificatori
 - Se tipo è **immutable** almeno: creatori, osservatori e produttori
 - Un tipo deve essere **totalmente popolato** (fully populated): usando creatori, osservatori e mutatori/produttori deve essere possibile ottenere (convenientemente) ogni possibile istanza di stato astratto
 - Ad esempio con solo i costruttori di `Poly` non si potrebbero definire tutti i polinomi a coefficienti interi, ma usando i produttori sì

Adeguatezza di un tipo (2)

- Anche se totalmente popolato, verificare se si può ragionevolmente migliorare efficienza fornendo ulteriori operazioni
 - Esempio: su IntSet, con le sole insert, remove e size si può verificare membership (verificando se size cambia dopo un remove) ma sembra molto inefficiente
 - ...ma non bisogna includere troppe operazioni (che non si adattano allo scopo del tipo), altrimenti l'astrazione diventa più difficile da comprendere, implementare, e mantenere
 - Esempio: calcolo della trasformata di Laplace o della derivata di un Poly potrebbero essere lasciate a metodi statici esterni
- Adeguatezza è concetto informale e dipendente dal contesto d'uso
 - Se contesto è limitato, bastano poche operazioni

Proprietà astratte

- Sono proprietà osservabili con i metodi observer
- Due categorie
 - Proprietà **evolutiva**: relazione fra uno stato astratto osservabile e lo stato astratto osservabile successivo
 - Esempio: mutabilità; grado di un Poly non cambia
 - Non immediatamente rappresentabili in JML
 - Proprietà **invariante**: proprietà degli stati astratti osservabili
 - Esempio: size() di un IntSet è sempre ≥ 0 , il grado di un Poly non è negativo
 - Sono rappresentabili in JML con **public invariant**

Invarianti astratti

- Ignoriamo l'implementazione e usiamo solo la specifica per dimostrare invarianti astratti, che sono la formalizzazione di una proprietà astratta
- Un abstract invariant è una condizione che deve essere sempre verificata per l'oggetto astratto
 - La dimensione di un IntSet è sempre ≥ 0
 - Un coefficiente di grado n di un Poly è sempre 0 per $n > \text{degree}()$
- JML public invariant
 - Possono usare solo attributi e metodi public della classe

```
//@ public invariant this.size() >= 0;  
//@ public invariant this.degree() >= 0;
```


Esempio

```
public /*@ pure @*/ class Triangolo {  
    //@ensures (* costruisce un triangolo non degenerare *)  
    //@signals (* TriangoloInvalidoException e) vertici.length !=3 &&  
    // (* i vertici[] costituiscono un triangolo non degenerare*);  
    public class Triangolo(Punto vertici[]) throws TriangoloInvalidoException  
  
        //@ ensures (*restituisce la lunghezza del lato minore *)  
        public float latoMinore()  
        //@ ensures (*restituisce la lunghezza del lato intermedio *)  
        public float latoMedio()  
        //@ ensures (*restituisce la lunghezza del lato maggiore *)  
        public float latoMaggiore()  
}
```

Invariante astratto

- Il triangolo non è degenere:
 - vale disequaglianza triangolare e i lati hanno lunghezza >0

//@ public invariant

//@ latoMaggiore() < latoMinore() + latoMedio() &&

//@ latoMinore() * latoMedio() * latoMaggiore() > 0;

Le proprietà astratte sono utili

- Gli utilizzatori della classe possono usare le proprietà come assunzioni sul comportamento della classe
- Esempi
 - Una stack ha proprietà evolutiva LIFO
 - Un triangolo non è degenere
 - Un Poly ha sempre grado ≥ 0
 - Un IntSet non ha size negativa
- Per essere davvero utili, però, bisogna essere sicuri che gli invarianti siano davvero verificati
- E' possibile verificare questo solo in base alla specifica

Ragionare al livello dell'astrazione

- L'invariante di Triangolo è verificato dalla specifica?
 - Occorre avere specifica precisa del costruttore
- Se il costruttore non consente di costruire triangolo degenere, allora siamo sicuri che l'invariante vale, in quanto la classe è pura: una volta costruito, il triangolo non può essere modificato
- La “prova” non utilizza l'implementazione ma solo la specifica del costruttore
 - Potrebbe essere dimostrata prima di implementare
 - Se l'implementazione sarà corretta rispetto alla specifica, l'invariante resterà verificato anche sull'implementazione

Ragionare con classi non pure

- Con classi non pure? non basta dimostrare proprietà per il costruttore, ma occorre considerare i mutatori
- Metodo induttivo:
 - Invariante vale al termine del costruttore
 - Sotto l'ipotesi che l'invariante sia verificato al momento della chiamata, mostrare che vale al termine di ciascun modificatore
- Esempio: dimostriamo l'invariante pubblico di IntSet
 - Vale per il costruttore (restituisce IntSet vuoto): dim. =0
 - Vale per insert: non può decrementare la dim. di IntSet
 - Vale per remove: remove eseguita solo se elem era già nell'insieme
- Non occorre dimostrare che l'invariante vale per gli osservatori (non possono modificare stato)

Implementazione di un ADT

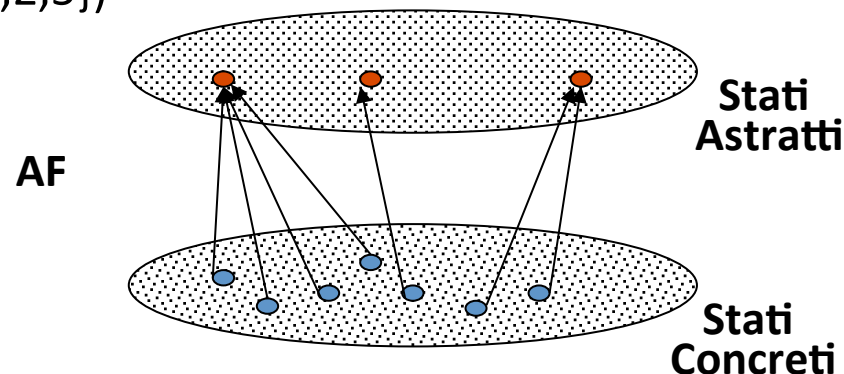
- L'implementazione di un tipo deve fornire
 - Una rappresentazione (**rep**) per gli oggetti del tipo, cioè una struttura dati per rappresentarne i valori
 - ...cioè un insieme di “instance variables” dichiarate private
 - L'implementazione di tutte le operazioni
- Esempio:
 - Per IntSet: gli elementi nell'insieme cambiano: astrazione è mutabile; allora si può pensare di usare un ArrayList per memorizzare gli elementi
 - Invece String è immutabile: una sua implementazione può usare gli array tradizionali

Come scegliere un rep?

- Scelta del rep dipende da **efficienza, semplicità, riuso** di strutture dati esistenti
- Un rep per IntSet con liste a puntatori avrebbe la stessa flessibilità dei ArrayList con efficienza leggermente maggiore se ci sono molti inserimenti, ma più scomoda da programmare
- Un rep per IntSet con array ordinato o albero binario potrebbe consentire, se utile, di ricercare se un elemento è già inserito con una procedura veloce come la ricerca binaria

Funzione di Astrazione


- Stati astratti (della **specifica**) vs. stati concreti (del **rep**)
 - Esempio: per IntSet: $\{1,2,5\}$ è uno stato astratto, mentre il valore $[1,2,5]$ (dell'ArrayList del rep) è uno stato concreto
- Funzione di astrazione AF: $\text{ConSt} \rightarrow \text{AbsSt}$
 - Associa a ogni stato concreto c di ConSt uno e un solo stato astratto $\text{AbsSt}(c)$
 - AF può essere totale o parziale, ma (di solito) non iniettiva: molti stati concreti possono essere associati allo stesso stato astratto
 - Esempio: per IntSet, $[1,2,5]$ e $[2,1,5]$ corrispondono allo stesso stato astratto $\{1,2,5\}$



Definizione di AF: IntSet

- La funzione di astrazione AF definisce il significato della rappresentazione
 - Cioè stabilisce come ogni oggetto della classe IntSet implementa un oggetto dell'astrazione “insieme di interi”
- Come definirla? Con un invariante!
 - Un invariante dichiarato “private” è autorizzato a usare anche le parti private (metodi puri e attributi) della classe
 - Si può usare un invariante per descrivere una relazione (funzione) fra le parti private e i metodi osservatori della classe
 - Per IntSet AF può essere scritta come:

```
/*@private invariant  
@ (\forall int i;; this.isIn(i)<==>arrayRep.contains(i))  
@*/
```



L'oggetto ArrayList
che abbiamo scelto
come rep di IntSet

Definizione di AF: Poly

- Poly di grado n implementato con vettore `trms` di dimensione $n+1$ che contiene coeff. del termine di grado i -esimo in i -esima posizione, con $0 \leq i \leq n$
- Quindi coefficiente i -esimo di un Poly c è `c.trms[i]`, con $0 \leq i \leq c.deg$
 - I termini di grado superiore a `deg` sono tutti nulli

```
/*@ private invariant
```

```
@ (\forall int i; 0 <= i && i <= deg; coeff(i) == trms[i]) &&
```

```
@ (\forall int j; j < 0 || j > deg; coeff(j) == 0);
```

```
@*/
```

Implementare AF

- AF descrive l'interpretazione del rep
 - Associa a ogni oggetto concreto l'oggetto astratto corrispondente
- Soluzione banale: usare toString(), che diventa utile per cercare errori nel codice
- public String toString() è un metodo predefinito della classe Object, che va sempre ridefinito
 - Implementazione di default: nome dell'oggetto + hash-code
- Scopo di toString() è restituire una rappresentazione testuale dell'oggetto astratto
- Quindi possiamo usare toString() per mappare l'oggetto concreto this nell'oggetto astratto

Come definire toString: IntSet

- Per esempio: IntSet: {1, 7, 3}

```
public String toString() {  
    if (els.size()==0) return "IntSet: {}";  
    String s = "IntSet: {" + arrayRep.elementAt(0).toString();  
    for (int i = 1; i<els.size(); i++)  
        s = s + ", " + arrayRep.elementAt(i).toString();  
    return s + "}";  
}
```

Rappresentanti legali

- Non tutti gli oggetti concreti di una classe sono rappresentanti legali degli oggetti astratti
 - Es: valore `[1,2,2]` per vettore `arrayRep` è scorretto, perché un insieme di interi non contiene elementi duplicati
- Le rappresentazioni legali sono quelle per cui valgono tutte le ipotesi sottese all'implementazione
 - Vogliamo descrivere in modo preciso queste ipotesi e inserirle nella documentazione della classe
 - AF è definita solo per i rappresentanti legali

Invariante di rappresentazione

- **Invariante di rappresentazione** (“rep invariant” o RI):
predicato $J: \text{ConcSt} \rightarrow \text{Boolean}$
 - È vero solo per oggetti “legali”: $J([1,2,2]) = \text{false}$, $J([1,3,2]) = \text{true}$
- Esempio per IntSet: il rep invariant di un oggetto concreto c è
 - $c.\text{arrayRep}$ è sempre $\neq \text{null}$ &&
 - $c.\text{arrayRep}$ contiene solo elementi $\neq \text{null}$ &&
 - in $c.\text{arrayRep}$ non ci sono mai due interi con lo stesso valore
- RI non specifica nulla sulla relazione fra oggetto astratto e concreto: questo è compito di AF

```
//@ private invariant arrayRep!= null &&  
                        !arrayRep.contains(null) &&  
//@ (\forall int i; 0 <= i && i<arrayRep.size();  
//@   (\forall int j; i < j && j<arrayRep.size();  
                        !(arrayRep.get(i).equals(arrayRep.get(j))));
```

- L’invariante è “private” perché riguarda solo l’implementazione!

Precisione del rep invariant

- Quando scrivere il rep invariant?
 - Prima di implementare qualunque operazione
- Come scrivere il rep invariant?
 - private invariant
- Che cosa scrivere nel rep invariant?
 - Le proprietà che caratterizzano quegli stati concreti che rappresentano qualche stato astratto
 - rep invariant, insieme ad AF, deve includere tutto ciò che serve per potere implementare i metodi, data la specifica
 - Es. Alberto implementa metodi insert e isIn, Bruno implementa remove e size, ma i due non si parlano: usano solo il rep invariant!
 - Es. Per IntSet non basta scrivere `els != null`: bisogna anche dire che non ci sono duplicati

Validità del rep invariant

- Se un metodo è observer, allora non può modificare RI
- RI deve essere valido negli stati osservabili da un utilizzatore della classe
 - Costruttore: rep invariant deve sempre valere all'uscita da un costruttore (salvo che nel caso in cui il costruttore lanci un'eccezione)
 - Per ogni metodo m, se rep invariant è verificato al momento della chiamata di m, allora rep invariant deve valere anche al momento dell'uscita da m
- Se rep non vale all'entrata di un metodo m, allora non si può garantire nulla
- Durante l'esecuzione di un metodo RI può diventare falso: deve essere verificato solo quando si ritorna al chiamante
 - Esempio: un'implementazione di insert(int x), che inserisce x in ultima posizione di arrayRep e poi verifica se non è duplicato

Implementare RI

- Il rep invariant definisce tutte le ipotesi sottostanti all'implementazione di un tipo
 - Definisce quali rappresentazioni sono legali, cioè per quali valori AF è definita
 - Si può implementare ad esempio con un repOk()

```
public boolean repOk() {  
    // Ensures: restituisce true se rep invariant  
    // vale per this, altrimenti restituisce false  
}
```

repOk per IntSet

```
public boolean repOk() {  
    // els <> null &&  
    if (els == null) return false;  
  
    //!els.contains(null) &&  
    if (els.contains(null)) return false;  
  
    //for all integers i, j. (0 <= i < j < c.els.size =>  
    //      c.els[i].intValue != c.els[j].intValue:  
    for (int j = i+1; j < els.size(); j++)  
        if (x.equals(els.get(j))) return false;  
    return true;  
}
```

Come verificare RI con repOk

- Nel codice: inserire il controllo di repOk nei mutators e nei costruttori, tramite il meccanismo delle asserzioni
 - Si chiama `assert repOk()`; al termine del metodo o del costruttore
 - Esempio per `IntSet`:

```
public void insert(int x) {  
    //codice che inserisce in this.els  
    assert repOk();  
    return;}

```
 - Se risultato è `true` si continua, altrimenti `AssertionError`
 - Molto utile, ma codice più lento!

Ragionare sulle astrazioni

- Scrivendo un programma, spiegandolo o leggendolo, si cerca di “convincersi” che sia corretto, ragionandovi
- Ragionare su procedure: data preconditione, ci si convince che il codice della procedura ha gli effetti desiderati
- Ragionare su data abstraction è più complicato
 - Occorre considerare l'intera classe e non solo una singola procedura
 - Il codice manipola il rep, ma ci si deve convincere che soddisfa la specifica degli oggetti astratti
- Metodo in due passi
 - Mostrare che implementazione conserva il rep invariant
 - Mostrare che le operazioni sono corrette rispetto alla specifica dell'astrazione, eventualmente utilizzando il rep invariant e proprietà dell'astrazione

Conservazione del rep invariant

- Sia C una data astrazione sui dati/tipo di dato astratto
- Mostrare che i costruttori ritornano oggetti che conservano rep invariant
 - Esempio: `public IntSet() { arrayRep = new ArrayList();}`
 - All'uscita da `IntSet()`, `arrayRep <> null` e `arrayRep.size = 0`, quindi rep invariant è verificato
- Mostrare che i metodi mantengono il rep invariant
 - Per ogni metodo m, si ipotizza che il rep invariant valga per this e per quei parametri c1, .., cn che sono proprio di tipo C
 - Si ipotizza inoltre che valga la preconditione del metodo
 - Si mostra allora che, quando m ritorna al chiamante, rep invariant vale sia per this che per i parametri c1, .., cn

Correttezza delle operazioni

- Non basta conservare RI!
- Ad esempio: `insert (int x) {}` conserva l'invariante, ma non inserisce `x`!
- Occorre mostrare che, dato RI, tutte le operazioni del rep implementano correttamente la specifica
 - I costruttori
 - I metodi
- La relazione fra rep e specifica è data da AF
- Avendo provato RI, si può ragionare per ogni operazione in modo indipendente
 - Noto RI, si prova la correttezza di ciascuno metodo

Effetti collaterali benevoli

- Un'astrazione mutabile deve avere un rep mutabile
- Però è possibile che il rep muti anche se l'oggetto astratto **non** cambia!
- Un'implementazione ha un effetto collaterale benevolo se modifica il rep senza influenzare lo stato astratto di un oggetto
 - La modifica non è quindi visibile al di fuori dell'implementazione
 - Di fatto, si passa da una rappresentazione a un'altra dello stesso oggetto astratto, ad esempio per motivi di efficienza
 - In questi casi, un'astrazione immutabile può avere un rep mutabile
- Con JML, purtroppo non è consentito avere effetti collaterali benevoli chiamando metodi puri...

Esempio

- Numeri razionali implementati come coppia di interi: int num, denom;
- Un razionale tipico è un qualunque rappresentante del razionale n/d

// $AF(c) = c.num/c.denom$

- Implementiamo i costruttori senza ridurre num e denom ai minimi termini, per non perdere tempo nell'inizializzazione; $c.denom \neq 0$ è un RI
- Possibile implementazione del confronto (equals)
 - Riduce entrambi i numeri ai minimi termini (chiamando metodo opportuno) e poi confronta i due numeratori e i due denominatori
 - Al termine, num e denom possono essere cambiati, ma l'oggetto astratto no
 - Vantaggio: non si perde tempo a semplificare numeri razionali poco usati, ma si semplifica solo quando un numero è usato con operazioni come equals
 - L'oggetto astratto non cambia, cambia solo quello concreto

Esporre parti mutabili del rep

- Un'implementazione espone il rep quando fornisce a utenti degli oggetti un modo per accedere a parti mutabili del rep
 - È un grave, ma comune, errore di implementazione!!
- Questo avviene tipicamente in due modi
 - Restituendo a un metodo chiamante un riferimento a una componente mutabile del rep
 - Inglobando nel rep un componente mutabile per cui esiste un riferimento all'esterno dell'oggetto
- Non c'è nessun problema a fornire un riferimento a una componente immutabile del rep (tanto essa non può mutare)

Esporre il rep

- Esempio: se classe IntSet avesse il metodo:

```
//@ ensures (* restituisce un ArrayList con tutti gli elementi  
//@ in this senza ripetizioni e in ordine arbitrario *);  
public ArrayList<Integer> allEls() {  
    return els;  
}
```

ARGHHHH!!!

- Gli utenti possono chiamare allEls e accedere all'implementazione: possono ad esempio inserire elementi duplicati e distruggere il rep invariant!
 - ArrayList<Integer> v = o.allEls();
 - v.add(new Integer(4));
 - v.add(new Integer(4)); //inserisce due copie di 4
- Soluzione corretta: restituire una copia di els: return els.clone();

Come non esporre il rep

- Dichiarare tutti gli attributi private (o al massimo protected o friend)
- Se gli attributi riferiscono oggetti mutabili
 - Non restituire reference a oggetti mutabili
 - Se necessario, creare copie degli oggetti interni mutabili per evitare di restituire gli originali attraverso i metodi
 - Non salvare riferimenti a oggetti esterni mutabili passati al costruttore
 - Se necessario fare copie e salvare riferimenti alle copie