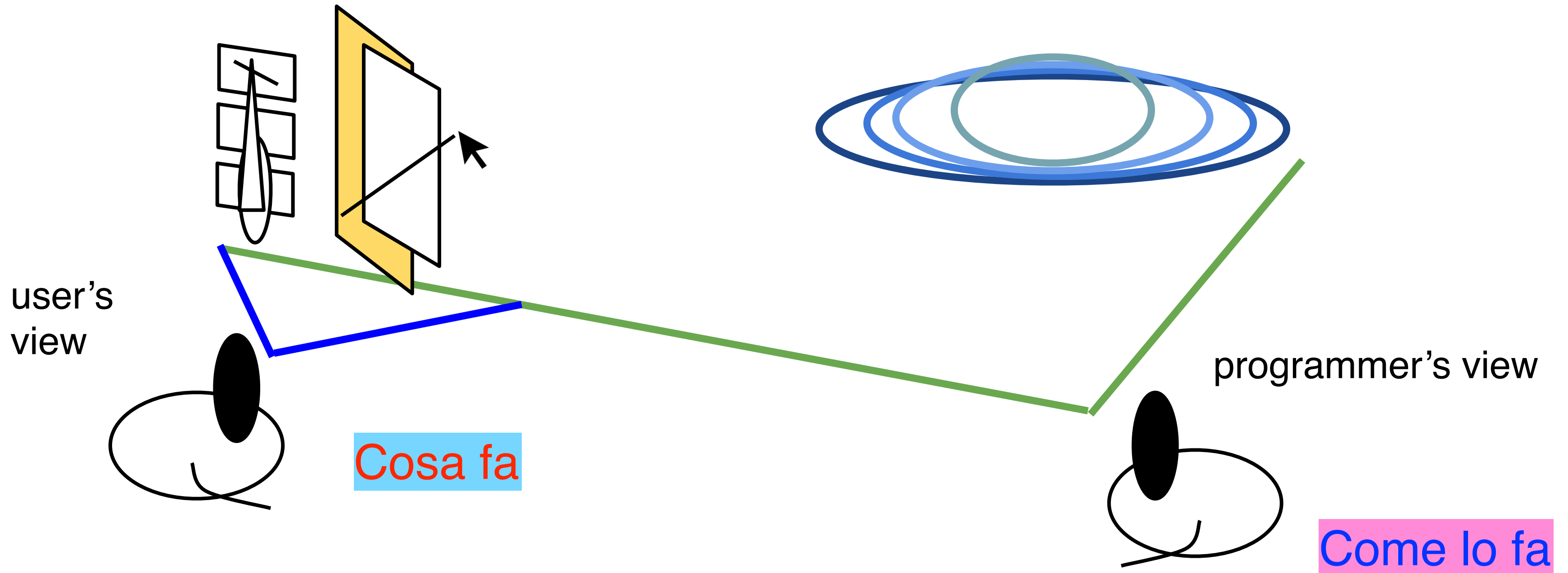
PROGRAMMAZIONE 2
7. Programming by Contract

Modularità



Quando si implementa un programma bisogna considerarsi un utente delle altre parti dalle quali il programma dipende

Contratti di uso

- Un “contratto” è l’insieme dei vincoli di uso concordati tra l’utente di un modulo software e chi implementa effettivamente il modulo
  – è una descrizione delle aspettative delle due parti

- Perché i contratti sono utili?
  – separation of concern: i dettagli implementativi sono mascherati all’utente che vede solo le funzionalità offerte
  – facilitano la manutenzione e il ri-uso del software

Astrazione via specifica

- Intuitivamente la specifica consente di astrarre dalla
  - specifica computazione descrivendo cosa fa e non come lo fa
  - astrazione procedurale (metodi statici)
  - astrazione sui dati (TDA)

Astraiamo tramite la specifica

- Procedural abstraction: separazione delle proprietà logiche di una azione computazionale dai suoi dettagli implementativi
- Data abstraction: separazione delle proprietà logiche dei dati dai dettagli della loro rappresentazione (dati e operazioni)

Metodologia di Programmazione

- Come descriviamo la specifica (tramite annotazioni che descrivo le proprietà)?
- Come dimostriamo che una implementazione soddisfa la specifica (verifica delle proprietà logiche)?
- Cosa succede con l'ereditarietà?

Interfacce come contratti?

- In Java la nozione di interfaccia permette di definire esplicitamente il “confine” tra cliente e implementatore
- Le interfacce in Java definiscono la sintassi e il tipo dei metodi di un metodo
- Sintassi e tipi forniscono una informazione limitata ai cliente!!

```
public interface IntSet {
        public void insert(int elem);
         public void remove(int elem);
        public boolean isIn(int elem);
        public boolean isEmpty( );
                        ...}
```

Javadoc

- non esiste una notazione formale, ma solo un insieme di convenzioni
- Javadoc: le principali convenzioni
  - Segnatura (tipo) dei metodi
  - Descrizione testuale del comportamento atteso (astrazione sul comportamento)
  - Dalla documentazione di Java
  - @param: description of what gets passed in
  - @return: description of what gets returned
  - @throws: exceptions that may occur

Un esempio: `String.contains`

```
public boolean contains(CharSequence s)
```

Returns true if and only if this string contains the specified sequence of char values.

Parameters:

s- the sequence to search for

Returns:

true if this string contains s, false otherwise

Throws:

`NullPointerException`

Since:

1.5

Simile ma…

- La pre-condizione del metodo: dichiara i vincoli che devono valere prima della invocazione del metodo (se i vincoli non sono sodddisfatti allora il contratto non vale)
  - `@requires`: l'obbligo del cliente
- La post-condizione del metodo: dichiara quali sono le proprietà che devono valere al termine dell'esecuzione del metodo (nell'ipotesi che la pre-condizione sia valida)
  - `@modifies`: descrive la portata delle modifiche effettuate durante l'esecuzione. Solo le entità descritte nella clausola "modifies" sono effettivamente modificate
  - `@throws`: le eccezioni che possono essere sollevate (come Javadoc)
  - `@effects`: proprietà che valgono sullo stato modificato
  - `@return`: il valore che viene restituito (come Javadoc)

Alternativamente

- La pre-condizione del metodo: come prima!!
- La post-condizione del metodo: dichiara quali sono le proprietà che devono valere al termine dell'esecuzione del metodo (nell'ipotesi che la pre-condizione sia valida)
- Quindi `@effects`:
  - descrive la portata delle modifiche effettuate durante l'esecuzione.
  - le eccezioni che possono essere sollevate (come Javadoc)
  - proprietà che valgono sullo stato modificato
  - il valore che viene restituito (come Javadoc)

Esempio: Vector

```
public static int
        change(Vector vec, Object oldelt, Object newelt)

@requires      v, oldelt e newelt sono valori non-null
               oldelt appartiene a vec

@modifies      vec

@effects       la prima occorrenza del valore oldelt in
               vec viene modificata con il valore newelt
               & gli altri elementi di vec non sono
               modificati

@returns       l'indice della posizione in vec che
               conteneva il valore oldelt e che ora
               contiene il valore newelt
```

Commenti: procedura stand-alone

- Procedura parziale
- Che succede se il cliente dell’astrazione invoca il metodo `change` con dei parametri che non soddisfano le pre-condizioni del metodo?
- Il codice del metodo è libero di fare qualunque cosa dato che non è vincolato dalla pre-condizione
- È opportuno generare un fallimento piuttosto che generare dei comportamenti che possono introdurre errori (usando opportune eccezioni)

Esempio: specifica di procedure stand-alone

```
public class Arrays{
    public static int search(int[] a,int x)
           throws NullPointerException,NotFoundException{
    \\EFFECTS: se a e’ null solleva NullPointerException,
     se x non appartiene ad a allora solleva NotFoundException,
     altrimenti restituisce un indice i tale che x=a[i]}
    public static int searchsorted(int[] a,int x)
            throws NullPointerException,NotFoundException{
   \\REQUIRES: a ordinato in modo crescente
   \\EFFECTS: se a e’ null solleva NullPointerException,
     se x non appartiene ad a allora solleva NotFoundException,
     altrimenti restituisce un indice i tale che x=a[i]}
   public static void sort(int[] a,int x)
            throws NullPointerException{
         \\MODIFIES: a
   \\EFFECTS: se a e’ null solleva NullPointerException
     altrimenti ordina a in modo crescente}
}
```

Esempio: commenti

- `search` e `searchsorted` non hanno effetti collaterali
- Le eccezioni vengono usate per segnalare casi particolari (devono sempre essere riportate nella specifica)
- `sort` modifica l'array
- `searchsorted` e' parziale

Implementazione: procedure

- Le procedure possono essere implementate in modi diversi
- Le implementazione di `search` sono corrette per `searchsorted` ma non sfruttano l'ordinamento
- Per esempio `sort` potrebbe essere implementato usando un `quicksort` e `mergesort` (altri algoritmi)
- In ogni caso il cliente delle procedure invoca i metodi in base al comportamento descritto nella specifica e non vede i dettagli dell'implementazione

Specifiche ed implementazioni

Supponiamo che Impl sia una possibile implementazione della specifica dell'astrazione S

Impl sodddisfa S se

- ogni comportamento di Impl è un comportamento permesso dalla specifica S
- "i comportamenti di Impl sono un sottoinsieme dei comportamenti specificati da S"

Se Impl non soddisfa S, allora Impl oppure S non sono "corretti"

- pragmaticamente è meglio cambiare l'implementazione che la specifica

Un esempio

```
public static int search(int[] a, int value) {
      for (int i = 0; i < a.length; i++)
        if (a[i] == value) return i;
      return -1;}
}
```

- Specifica A
  - **requires**: value è un valore memorizzato nell’array a
  - **return**: un indice i tale che a[i] = value
- Specifica B
  - **requires**: value è un valore memorizzato nell’array a
  - **return**: il più piccolo indice i tale che a[i] = value

Sono corrette? Quali delle due è più forte???

Un altra specifica…..

```
public static int search(int[] a, int value) {
      for (int i = 0; i < a.length; i++)
        if (a[i] == value) return i;
      return -1;}
}
```

- Specifica A
  - **requires**: value è un valore memorizzato nell'array a
  - **return**: un indice i tale che a[i] = value
- Specifica B
  - **requires**: value è un valore memorizzato nell'array a
  - **return**: il più piccolo indice i tale che a[i] = value
- Specifica C
  - **return**: l'indice i tale che a[i]=value, oppure -1 nel caso in cui value
  - non sia memorizzato nell'array a

Diverse specifiche sono possibili..

- Una specifica “forte”
  - difficile da soddisfare (maggiore numero di vincoli sull’implementazione)
  - facile da usare (il cliente dell’astrazione può fare maggiori assunzioni sul comportamento)
- Una specifica “debole”
  - facile da verificare (facile da implementare, molto implementazioni la possono soddisfare)
  - difficile da usare per il minor numero di assunzioni

Una visione formale

- Una specifica è una formula logica
  - S1 è più forte di S2 se S1 implica S2
- Lo avete visto formalmente a LPP
  - trasformate la specifica in una formula logica e poi verificate l'implicazione
  - $(S1 \Rightarrow S2)$

Conclusioni

- Nel caso delle procedure stand-alone la specifica descrive gli effetti sui parametri, sui valori restituiti e gli eventuali effetti collaterali
- Meglio usare le eccezioni per controllare le proprietà dei parametri e rendere il comportamento totale vs parziale

Forme di astrazione

- Procedural abstraction: separazione delle proprietà logiche di una azione computazionale dai suoi dettagli implementativi
- Data abstraction: separazione delle proprietà logiche dei dati dai dettagli della loro rappresentazione

Come descriviamo la specifica di un TDA?? Cosa cambia??

Abstract Data Type

- Un abstract data type (ADT) è una collezione di elementi il cui comportamento logico è definito da un dominio di valori e da un insieme di operazioni su quel dominio
- ADT
  - Nome
  - Valori
  - Operazioni
  - Semantica delle operazioni

Gli ingredienti di una specifica

- Java (parte sintattica della specifica)
  - classe o interfaccia
    - ✓ per ora solo classi
  - nome per il tipo (la classe)
  - operazioni
    - ✓ metodi di istanza, incluso il/i costruttore/i
- la specifica del tipo
  - fornita dalla clausola OVERVIEW che descrive i valori astratti degli oggetti e alcune loro proprietà
    - ✓ per esempio la modificabilità
- per il resto la specifica è una specifica dei metodi
  - strutturata tramite pre-condizioni (clausola requires) e post-condizioni (clausola effects) che fa riferimento a **`this`**

ADT: visione costruttiva

- Le operazioni sono caratterizzate da una pre-condizione e da una post-condizione
- Pre-condizione: formula logica che caratterizza le proprietà e il valore degli argomenti
- Post-condizione: formula logica che caratterizza il risultato calcolato dall'operazione rispetto al valore degli argomenti

Formato della specifica

```
public class NuovoTipo {
  // OVERVIEW: Gli oggetti di tipo NuovoTipo
  // sono collezioni modificabili di …

  // costruttori
  public NuovoTipo( )
    // REQUIRES:…
    // EFFECTS: ...

  // metodi
  // specifiche degli altri metodi
}
```

Specifica: **IntSet**

```
public class IntSet {
  // OVERVIEW: un IntSet è un insieme modificabile
  // di interi di dimensione qualunque
 //Elemento tipico: {x1,…,xn}
  //costruttore
    public IntSet( )
      // EFFECTS: inizializza this all’insieme vuoto
  // metodi
    public void insert(int x)
       // EFFECTS: aggiunge x a this
    public void remove (int x)
       // EFFECTS: toglie x da this
    public boolean isIn(int x)
       // EFFECTS: se x appartiene a this ritorna true,
       // false altrimenti
...
}
```

Specifica: IntSet

```java
public class IntSet {
    ...
    // metodi
    ...
    public int size( )
        // EFFECTS: ritorna la cardinalità di this
    public int choose( ) throws EmptyException
        // EFFECTS: se this è vuoto, solleva
        // EmptyException, altrimenti ritorna un
        // elemento qualunque contenuto in this
}
```

IntSet: analisi

```
public class IntSet {
  // OVERVIEW: un IntSet è un insieme modificabile
  // di interi di dimensione qualunque
  //Elemento tipico: {x_1,….x_n}
```

- I valori astratti degli oggetti della classe sono descritti nella clausola OVERVIEW in termini di concetti noti
  - gli insiemi matematici
  - l’uso di una notazione matematica (in seguito)
- Gli stessi concetti sono usati nella specifica dei metodi
  - aggiungere, togliere elementi
  - appartenenza, cardinalità

IntSet: analisi

```
public class IntSet {
 //OVERVIEW: un IntSet è un insieme modificabile
 //di interi di dimensione qualunque

//costruttore
  public IntSet( )
   //EFFECTS: inizializza this all'insieme vuoto
```

- Un solo costruttore (senza parametri)
    - inizializza **this** (l'oggetto nuovo)
    - non è possibile vedere lo stato dell'oggetto tra la creazione e l'inizializzazione

IntSet: analisi

```
public int size( )
        //EFFECTS: ritorna la cardinalità di this
public int choose( ) throws EmptyException
        // EFFECTS: se this è vuoto, solleva
        // EmptyException, altrimenti ritorna un
        // elemento qualunque contenuto in this
public boolean isIn(int x)
       // EFFECTS: se x appartiene a this ritorna true,
       // false altrimenti
```

- Osservatori
  - non modificano lo stato del proprio oggetto: **`choose`** può sollevare un'eccezione (se l'insieme è vuoto)
    - **`EmptyException`** può essere unchecked, perché l'utente può utilizzare **`size`** per evitare di farla sollevare
    - **`choose`** è sotto-determinata (implementazioni corrette diverse possono dare risultati diversi)

IntSet: analisi

```
public void insert(int x)
   //MODIFIES:this
   //EFFECTS:aggiunge x a this
public void remove(int x)
  //MODIFIES:this
  // EFFECTS:toglie x da this
```

- Modificatori
  - modificano lo stato del proprio oggetto
  - notare che né **insert** né **remove** sollevano eccezioni
    - ✓ se si inserisce un elemento che c'è già
    - ✓ se si rimuove un elemento che non c'è

Aspetti metodologici

- Per prima cosa si definisce la specifica
  - “scheletro” formato da header, overview, pre- e post- condizioni di tutti i metodi
  - mancano la rappresentazione degli oggetti e il codice dei corpi dei metodi…
    - ✓ che possono essere sviluppati in un momento successivo e indipendentemente dallo sviluppo dei “moduli” che usano il nuovo tipo di dato
    - ✓ ed è molto importante riuscire a differire le scelte relative alla rappresentazione

Un cliente di **IntSet**

```
public static IntSet getElements(int[ ] a)
                                 throws NullPointerException {
   // EFFECTS: se a=null solleva NullPointerException,
  // altrimenti restituisce un insieme che contiene
  // tutti e soli gli interi presenti in a

   IntSet s = new IntSet( );
   for (int i = 0; i < a.length; i++)
        s.insert(a[i]);
   return s;
}
```

- Scritta solo conoscendo la specifica di **IntSet** non accedendo all'implementazione che magari non esiste ancora, ma pure se ci fosse non potrebbe "vederla"
  - costruisce, accede e modifica l'oggetto solo attraverso i metodi

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

Data abstraction via specifica

- Con la specifica, astraiamo dall'implementazione del tipo di dato (come nel caso delle procedure)
- Come si implementa un TDA?
- Dobbiamo fornire la rappresentazione (nascosta all'utente esterno, mentre è visibile all'implementazione delle operazioni) e implementare le operazioni (costruttori e metodi)
- Come si dimostra che l'implementazione soddisfa la specifica?

Specifica di un tipo “primitivo”

- Le specifiche sono ovviamente utili anche per capire usare correttamente i tipi di dato “primitivi” di Java
- Vedremo, come esempio, il caso dei vettori
  - `Vector`
  - array dinamici che possono crescere e ridursi
  - sono definiti nel `package java.util` Le specifiche sono ovviamente utili anche per capire usare correttamente i tipi di dato “primitivi” di Java
  -

Specifica di Vector

```
public class Vector {
 // OVERVIEW: un Vector è un array modificabile
 // di dimensione variabile i cui elementi sono
 // di tipo Object: indici tra 0 e size - 1
 // costruttore
 public Vector( )
     // EFFECTS: inizializza this a vuoto
 // metodi
 public void add(Object x)
      //MODIFIES:this
     // EFFECTS: aggiunge una nuova posizione a this
     // inserendovi x
 public int size( )
     // EFFECTS: ritorna il numero di elementi di this
  ...
}
```

Specifica di **Vector**

```
public Object get(int n) throws IndexOutOfBoundsException
  // EFFECTS: se n<0 o n>= this.size solleva
  // IndexOutOfBoundsException, altrimenti ritorna
  // l'oggetto in posizione n in this
public void set(int n, Object x) throws
   IndexOutOfBoundsException
  //MODIFIES:this
   // EFFECTS: se n<0 o n>= this.size solleva
   // IndexOutOfBoundsException, altrimenti modifica this
   // sostituendo l'oggetto x in posizione n
public void remove(int n) throws IndexOutOfBoundsException
     //MODIFIES:this
 // EFFECTS: se n<0 o n>= this.size solleva
   // IndexOutOfBoundsException, altrimenti modifica this
   // eliminando l'oggetto in posizione n
```

`Vector`: analisi

```
public class Vector {
  // OVERVIEW: un Vector è un array modificabile
  // di dimensione variabile i cui elementi sono
  // di tipo Object: indici tra 0 e size - 1
  // costruttore
```

- Gli oggetti della classe sono descritti nella specifica in termini di concetti noti: gli `array`
- Gli stessi concetti sono anche usati nella specifica dei metodi
  - indice, elemento identificato dall'indice
- Il tipo è modificabile come l'array
- Notare che gli elementi sono di tipo `Object`
  - non possono essere int, bool o char

Vector: analisi

```
public class Vector {
 // OVERVIEW: un Vector è un array modificabile
 // di dimensione variabile i cui elementi sono
 // di tipo Object: indici tra 0 e size - 1
 // costruttore
 public Vector( )
     // EFFECTS: inizializza this a vuoto
```

- Un solo costruttore (senza parametri)
  - inizializza this (l'oggetto nuovo) a un "array" vuoto

Vector: analisi

```
public void add(Object x)
     //MODIFIES:this
     //EFFECTS: aggiunge una nuova posizione a this inserendovi x
public void set(int n, Object x)
                          throws IndexOutOfBoundsException
  //MODIFIES:this
  //EFFECTS: se n<0 o n>= this.size solleva
  // IndexOutOfBoundsException, altrimenti modifica this
 // sostituendovi l'oggetto x in posizione n
public void remove(int n) throws IndexOutOfBoundsException
   //MODIFIES:this
   //EFFECTS: se n<0 o n>= this.size solleva
   // IndexOutOfBoundsException, altrimenti modifica this
   // eliminando l'oggetto in posizione n
```

- Sono modificatori
  - modificano lo stato del proprio oggetto
  - **set** e **remove** possono sollevare un'eccezione unchecked

Vector: analisi

```
public int size( )
    // EFFECTS: ritorna il numero di elementi di this
public Object get(int n)
        throws IndexOutOfBoundsException
  // EFFECTS: se n<0 o n>= this.size solleva
  // IndexOutOfBoundsException, altrimenti ritorna
  // l'oggetto in posizione n in this
```

- Sono osservatori
  - non modificano lo stato del proprio oggetto
  - `get` può sollevare un'eccezione primitiva unchecked