Astrazione procedurale

Procedure stand alone

- ✓ indipendenti da specifici oggetti
- ✓ come si realizzano in Java
  - insieme di metodi statici definiti dentro una classe che non ha
    - variabili e metodi di istanza
    - costruttore
  - può contenere variabili statiche
    - condivise dalle varie attivazioni di metodi
- ✓ una procedura è un mapping da un insieme di argomenti di ingresso ad un insieme di risultati
  - con possibile modifica di alcuni degli argomenti di ingresso
    - solo se sono oggetti
  - possibili effetti laterali su variabili di classe o di istanza visibili

Astrazione via specifica

- ✓ con la specifica, astraiamo dall'implementazione della procedura
- ✓ località
  - l'implementazione di una astrazione può essere letta o scritta senza esaminare le implementazioni delle altre astrazioni
    - utile durante lo sviluppo (anche da parte di più persone) e la manutenzione
- ✓ modificabilità
  - un'astrazione può essere reimplementata senza richiedere modifiche alle astrazioni che la utilizzano
    - utile durante la manutenzione per ridurre gli effetti indotti da una modifica

Un esempio di specifica

```
public class Arrays {
  // OVERVIEW:  La classe fornisce un insieme di
  // procedure utili per manipolare arrays di int
  public static int search (int[] a, int x)
    // EFFECTS: se x occorre in a, ritorna un
    // indice in cui occorre, altrimenti -1
  public static int searchSorted (int[] a, int x)
    // REQUIRES: a è ordinata in modo crescente
    // EFFECTS: se x occorre in a, ritorna un
    // indice in cui occorre, altrimenti -1
  public static void sort (int[] a)
    // EFFECTS: riordina gli elementi di a in
    // modo crescente, per esempio
    // se a=[3,1,6,1]  a_post=[1,1,3,6]
}
```

Un esempio: commenti 1

```
public class Arrays {
   // OVERVIEW:  La classe fornisce un insieme di
   // procedure utili per manipolare arrays di int
   ....
}
```

- ✓ la classe compare nella specifica, perché i metodi dovranno essere reperiti usando il nome della classe

Un esempio: commenti 2

```
...
  public static int search (int[] a, int x)
     ...
  public static int searchSorted (int[] a, int x)
     ...
  public static void sort (int[] a)
     ...
```

- ✓ gli headers dei metodi (codice Java) sono la parte sintattica della specifica del metodo
- ✓ specificano (in aggiunta alla visibilità)
  - nome del metodo
  - nomi e tipi dei parametri formali
  - tipo del risultato
  - *search: int array * int -> int*
- ✓ dovrebbero anche elencare le eventuali eccezioni sollevate dalla procedura
  - ignorate per ora

Un esempio: commenti 3

```
...
  public static int search (int[] a, int x)
     // EFFECTS: se x occorre in a, ritorna un
     // indice in cui occorre, altrimenti -1
  public static int searchSorted (int[] a, int x)
     // REQUIRES: a è ordinata in modo crescente
     // EFFECTS: se x occorre in a, ritorna un
     // indice in cui occorre, altrimenti -1
...
```

- ✓ la clausola REQUIRES descrive le condizioni che devono essere verificate sui parametri di ingresso perché la procedura sia definita
  - possono esserci inputs impliciti (variabili visibili, files, etc.)
- ✓ se la clausola REQUIRES non è presente, la procedura è *totale* (esempio, *search*)
  - è definita per tutti gli inputs corretti rispetto al tipo
- ✓ altrimenti è *parziale* (esempio, *searchSorted*)

Un esempio: commenti 4

```
...
  public static int searchSorted (int[] a, int x)
     // REQUIRES: a è ordinata in modo crescente
     // EFFECTS: se x occorre in a, ritorna un
     // indice in cui occorre, altrimenti -1
  public static void sort (int[] a)
     // EFFECTS: riordina gli elementi di a in
     // modo crescente, per esempio
     // se a=[3,1,6,1]  a_post=[1,1,3,6]
...
```

- la clausola EFFECTS descrive le proprietà degli outputs e le modifiche effettuate su tutti gli inputs
  - compresi gli inputs impliciti
- si suppone che siano verificate le proprietà specificate in REQUIRES
- *a_post* rappresenta il valore di *a* dopo il ritorno del metodo

Un esempio: commenti 5

```
...
  public static void sort (int[] a)
     // EFFECTS: riordina gli elementi di a in
     // modo crescente, per esempio
     // se a=[3,1,6,1]  a_post=[1,1,3,6]
...
```

- ✓ la postcondizione ci fa vedere che alcuni degli inputs vengono modificati
  - potrebbero anche essere inputs impliciti (variabili visibili, files, etc.)
- ✓ se esistono inputs che vengono modificati
  - la procedura produce effetti laterali

Specifica ed implementazione

- ✓ per prima cosa si definisce la specifica
  - "scheletro" formato da headers e pre e post condizioni
  - manca il codice dei corpi dei metodi
    - che può essere sviluppato in un momento successivo ed indipendentemente dallo sviluppo dei "moduli" che usano le procedure specificate
- ✓ comunque, l'implementazione dovrà "soddisfare" la specifica

Esempi di implementazione 1

```
public class Arrays {
  // OVERVIEW:  La classe fornisce un insieme di
  // procedure utili per manipolare arrays di int
  ...
  public static int searchSorted (int[] a, int x)
     // REQUIRES: a è ordinata in modo crescente
     // EFFECTS: se x occorre in a, ritorna un
     // indice in cui occorre, altrimenti -1
     // usa la ricerca lineare
  {if (a == null) return -1;
  for (int i = 0; i < a.length; i++)
       if (a[i] == x) return i;
      else if (a[i] > x) return -1;}
...
}
```

- ✓ la specifica (effects) è sottodeterminata
  - possiamo ottenere risultati diversi con diverse implementazioni
- ✓ se la precondizione non è soddisfatta, l'implementazione non è corretta

Esempi di implementazione 2.1

```
public class Arrays {
  // OVERVIEW: ...
 public static void sort (int[] a)
     // EFFECTS: riordina gli elementi di a in modo
     // crescente, se a=[3,1,6,1]  a_post=[1,1,3,6]
     // usa il QuickSort
  {if (a == null) return;
  quickSort(a, 0, a.length - 1);}
...
}
```

- ✓ dobbiamo inserire nella classe il metodo `quickSort`
  - che può essere inserito come private
    - non visibile al di fuori della classe
- ✓ per i metodi private, potrebbe essere sufficiente l'implementazione
  - non esistono utenti esterni alla classe
- ✓ diamo anche la specifica, che può essere utile nella manutenzione

Esempi di implementazione 2.2

```
private static void quickSort (int[] a, int mi, int ma)
    // REQUIRES: a non è null, 0<=mi, ma<a.length
    // EFFECTS: riordina gli elementi tra a[mi] e
    // a[ma] in modo crescente
  {if (mi >= ma) return;
  int mid = partition(a, mi, ma);
  quickSort(a, mi, mid);
  quickSort(a, mid + 1, ma);}
```

- ✓ dobbiamo inserire nella classe anche il metodo private `partition`
- ✓ quando possibile,
  - se non è troppo costoso (vedi `searchSorted`)

  l'implementazione dovrebbe verificare esplicitamente la precondizione dei metodi invocati (quicksort e partition)
- ✓ la precondizione di quickSort è semplice da verificare, ma non lo facciamo
  - perché è destinata ad essere usata solo nel contesto di questa classe (private)
  - sappiamo che è sempre invocata in modo corretto (lo possiamo dimostrare!)
- ✓ non lo facciamo nemmeno per partition (specificata dopo)

Esempi di implementazione 2.3

```
private static int partition (int[] a, int mi, int ma)
    // REQUIRES: a non è null, 0<=mi<ma<a.length
    // EFFECTS: riordina gli elementi tra a[mi] e
    // a[ma] in due gruppi mi..ris e ris+1..ma, tali
    // che tutti gli elementi del secondo gruppo sono
    // >= di quelli del primo; ritorna ris
  {int x = a[mi];
  while (true) {
    while (a[ma] > x) ma--;
    while (a[mi] < x) mi++;
    if (mi < ma) {
        int temp = a [mi]; a[mi] = a[ma]; a[ma] = temp;
        ma--; mi++;}
    else return ma; }
  }}
```

Procedure ed eccezioni

- durante l'esecuzione di una procedura si possono verificare varie situazioni che possiamo considerare eccezionali
  - generazione di errori a run time la cui presenza non può essere verificata a tempo di compilazione
    - accesso ad un elemento di un array con indice "scorretto"
    - accesso a puntatori ad oggetti vuoti (null)
    - impossibilità di effettuare conversioni forzate di tipo (casting)
  - non è verificata la precondizione (procedure parziali)
    - potrebbe succedere di tutto, dal ritorno di risultati privi di significato, alla non terminazione, al danneggiamento di dati permanenti
  - anche se la precondizione è verificata, possono esserci valori degli inputs, per i quali la procedura ha un comportamento particolare
    - per esempio, ritorna valori speciali con cui si informa il chiamante della situazione

Precondizione non soddisfatta

- non è verificata la precondizione
  - potrebbe succedere di tutto, dal ritorno di risultati privi di significato, alla non terminazione, al danneggiamento di dati permanenti

```
public static int mcd (int n, int d)
    // REQUIRES: n, d > 0
    // EFFECTS: ritorna il massimo comun divisore
    // di n e d
```

- chiunque utilizzi la procedura deve preoccuparsi di verificare che i dati passati verifichino la precondizione
  - chi lo garantisce?
- chi implementa la procedura può ignorare i casi non previsti

Comportamenti particolari

- anche se la precondizione è verificata, possono esserci valori degli inputs, per i quali la procedura ha un comportamento particolare
  - per esempio, ritorna valori speciali con cui si informa il chiamante della situazione

```
public static int search (int[] a, int x)
    // EFFECTS: se x occorre in a, ritorna un
    // indice in cui occorre, altrimenti -1

public static int fact (int n)
    // EFFECTS: se n>0, ritorna n!, altrimenti 0
```

- il chiamante deve comunque trattare in modo speciale il valore che codifica la situazione particolare
  - chi lo garantisce?

Robustezza

- le procedure parziali e la codifica di situazioni particolari portano a programmi poco robusti
- un programma robusto si comporta in modo ragionevole anche in presenza di "errori" (graceful degradation)
  - per esempio, continua dopo il verificarsi dell'errore con un comportamento ben-definito che approssima quello normale
  - come minimo, termina con un messaggio di errore "informativo" senza danneggiare dati permanenti
- cosa serve?
  - un meccanismo (o approccio) che trasferisca l'informazione al chiamante in tutte queste situazioni
  - distinguendo le varie situazioni
  - con una gestione delle situazioni "strane" separata dal flusso di controllo normale della procedura

Il meccanismo delle eccezioni

- una procedura può terminare
  - normalmente, ritornando un risultato
  - in modo eccezionale
    - ci possono essere diverse terminazioni eccezionali
    - in Java, corrispondono a diversi tipi di eccezioni
    - il nome del tipo di eccezione viene scelto da chi specifica la procedura per fornire informazione sulla natura del problema
- le eccezioni giocano un ruolo molto importante nell'astrazione via specifica
  - la specifica del comportamento deve riguardare anche le terminazioni eccezionali

Eccezioni nella specifica

```
public static int fact (int n) throws NonpositiveExc
    // EFFECTS: se n>0, ritorna n!
    // altrimenti solleva NonpositiveExc

public static int searchSorted (int[] a, int x) throws
  NullPointerExc, NotFoundExc
    // REQUIRES: a è ordinato in modo crescente
    // EFFECTS: se a è null solleva NullPointerExc
    // se x non occorre in a solleva NotFoundExc
    // altrimenti ritorna un indice in cui occorre
```

- ✓ le procedure possono continuare ad essere parziali
  - verificare la precondizione e sollevare un'eccezione ridurrebbe in modo inaccettabile l'efficienza di `searchSorted`
  - la specifica del comportamento eccezionale presume comunque che l'eventuale precondizione sia soddisfatta