Programmazione II

11. Astrazione Iterazione (Uso, Specifica, Implementazione) (PDJ 6-6.4)

Dragan Ahmetovic

Riassunto sulle Tipologie di Astrazione

Tipi di Astrazione

Tutti i tipi di astrazione hanno come l'obiettivo la separazione dello scopo del programma dall'effettivo modo in cui è sviluppato.

- Astrazione Procedurale
  - Specificare il comportamento di una procedura
  - Senza definirne l'implementazione
- Astrazione su Tipi di Dati
  - Specificare un tipo di dati in base all'informazione che vuole modellare e al suo comportamento
  - Senza definire la sua rappresentazione e la sua implementazione
- Astrazione di Iterazione
  - A che cosa potrebbe servire?

Astrazione di Iterazione

Motivazione

Ci possono essere dei tipi di dati che potrebbero dover avere nella loro rappresentazione degli **insiemi di dati** di un qualche tipo

- Nella **rappresentazione** avranno delle **strutture dati** apposite (array, `List`, `Set`, `HashMap`...)
  - `IntSet`, `Poly`, ma anche `Studente` coi suoi corsi
- L'utente del tipo di dato potrebbe voler fare dei **ragionamenti di insieme** sui dati contenuti
  - Elementi positivi di `IntSet`, corsi superati con >25 dallo `Studente`, monomi col grado pari di `Poly`
- Come posso gestire queste esigenze?
  - Con metodi esterni alla classe, esponendo la rappresentazione - rischio modifica del dato
  - Con metodi esterni alla classe, copiando la rappresentazione - costoso e vincolo l'implementazione
  - Con qualche indicizzazione - non tutte le strutture dati sono indicizzabili
  - Creando metodi appositi - irrealistico aggiungere nuovi metodi per ogni esigenza

Astrazione di Iterazione

Concetto di Iterator

Meccanismo universale per accedere ai dati di interesse, **senza esporre la rappresentazione**

- Un `Iterator` è un oggetto che permette di:
  - **Verificare se esistono ancora elementi** su cui iterare
  - **Restituire un nuovo elemento** tra quelli presenti
  - **Eliminarlo se lo si desidera**
  - **Attenzione**: Liskov chiama un tale tipo "Generatore" (noi lo chiameremo `Iterator`)

Interfaccia `Iterator`

- Questo comportamento è definito nell'interfaccia `` `Iterator<E>` ``
- Tipo iterato `E` stabilito mediante **generics**
  - Ha un metodo `` `boolean hasNext()` `` per verificare se ci sono altri elementi di tipo `E`
  - Ha un metodo `` `E next()` `` per restituire il prossimo oggetto di tipo E
  - Può avere un metodo `` `remove()` `` per rimuovere l'ultimo elemento restituito da `` `next()` ``
- L'oggetto `Iterator` dovrà avere uno stato per tenere traccia di dov'è arrivata l'iterazione

Astrazione di Iterazione

Concetto di `Iterable`

Un oggetto che ha una serie di dati che si vuole iterare in genere non itera direttamente sui dati

- Invece, avrà un metodo per **produrre** un apposito oggetto di tipo `Iterator`
  - L'`Iterator` creato dovrà poter iterare (e quindi accedere) i dati contenuti nella classe che lo genera
  - Il metodo può creare molteplici `Iterator`, che iterano sugli stessi dati in maniera indipendente
  - È possibile anche che l'oggetto abbia diversi metodi per restituire diversi `Iterator` se ha diverse serie di dati (o se vuole iterare una serie in maniera diversa)

Interfaccia `Iterable`

- Questo comportamento è definito nell'interfaccia '`Iterable<E>`'
  - Ha un metodo `Iterator<E> iterator()` per restituire un `Iterator` ai dati in questione
  - Liskov chiama questo metodo "Iteratore" (noi lo chiameremo **metodo** `iterator()`)
- Quindi si dovrà creare un qualche nuovo tipo di dato che implementa `Iterator`
  - il metodo `iterator()` dovrà creare un nuovo oggetto di quel tipo e lo dovrà restituire
  - Ogni `Iterator` dovrà tenere traccia in maniera indipendente del proprio stato di iterazione

Astrazione di Iterazione

`Iterable` o `Iterator`?

Come stabilire se un tipo deve implementare `Iterable` o direttamente `Iterator`

- **Standalone** `Iterator`
  - Se il tipo non ha altri comportamenti a parte quello di iterare su una serie di dati
  - Se iterare è la funzionalità principale e i comportamenti del tipo sono legati al suo stato di iterazione
  - Se non serve o non è possibile iterare in maniera indipendente sulla stessa serie di dati
  - Se serve per generare dei dati nuovi, senza una struttura dati (`Collection`) a supporto
  - es: `Scanner`, generatori di serie numeriche
- `Iterable`
  - Se iterare sui dati del tipo è una funzionalità secondaria
  - Se serve poter iterare in maniera indipendente sui dati del tipo
  - Se il tipo contiene diverse serie di dati su cui iterare
  - Se l'astrazione del tipo non prevede di mantenere uno stato di iterazione
  - es: `ArrayList`, `Studente`

Utilizzo degli Iteratori

Sintassi

- Assegnamento di un iteratore:
  - `List<String> items = ...`
  - `Iterator<String> itemsIterator = items.iterator()`
- Utilizzo dell'iteratore:

```
while(itemsIterator.hasNext()) {
  String item = itemsIterator.next();
  System.out.println(item);
}
```

- Foreach - per le classi che implementano l'interfaccia "`Iterable`" è possibile anche fare:

```
//foreach fa 3 cose:
//  1) crea un Iterator di items chiamando items.iterator()
//  2) a ogni iterazione controlla se ci sono altri elementi con hasNext() dell'iterator
//  3) mette il prossimo elemento in item con next() dell'iterator
for(String item: items)
  System.out.println(item);
```

Utilizzo degli Iteratori

Esempio Uso Singolo Iteratore

```
1  import java.util.ArrayList;
2  import java.util.Iterator;
3
4  public class Main {
5    public static void main(String[] args) {
6
7      ArrayList<Integer> numbers = new ArrayList<Integer>();
8      numbers.add(12);
9      numbers.add(8);
10     numbers.add(2);
11     numbers.add(23);
12
13     Iterator<Integer> it = numbers.iterator();
14
15     while(it.hasNext())
16       if(it.next() < 10)
17         it.remove();
18
19     System.out.println(numbers);
20   }
21 }
```

Utilizzo degli Iteratori

Esempio Uso Più Istanze dello Stesso Iteratore

```
1  import java.util.ArrayList;
2  import java.util.Iterator;
3
4  public class Main {
5    public static void main(String[] args) {
6
7      ArrayList<Integer> numbers = new ArrayList<Integer>();
8      numbers.add(1);
9      numbers.add(2);
10     numbers.add(3);
11     numbers.add(4);
12     numbers.add(5);
13     numbers.add(6);
14     numbers.add(7);
15     numbers.add(8);
16     numbers.add(9);
17     numbers.add(10);
18
19     while(int i : numbers)
20       while(int j : numbers)
21         System.out.println(i + " x " + j + " = " + i*j);
22   }
23 }
```

Utilizzo degli Iteratori

Esempio Uso Più Iteratori Diversi

```
1  import java.util.ArrayList;
2  import java.util.Iterator;
3
4  public class Studente {
5
6    private ArrayList<String> corsiSeguiti;
7    private ArrayList<String> esamiDati;
8
9    ...
10
11   public Iterator<String> iteratoreCorsi() { ... }
12   public Iterator<String> iteratoreEsami() { ... }
13
14   public static void main(String[] args) {
15     Iterator<String> corsi = s.iteratoreCorsi()
16     while(corsi.hasNext())
17       System.out.println(corsi.next())
18
19     Iterator<String> esami = s.iteratoreEsami()
20     while(esami.hasNext())
21       System.out.println(esami.next())
22   }
23 }
```

Specifica e Implementazione degli Iteratori

Specifiche sintattiche

Una buona parte delle specifiche degli Iterator e Iterable è data dalle suddette interfacce

- Iterator:
  - `` `boolean hasNext()` `` - Returns true if the iteration has more elements
  - `` `E next()` `` - Returns the next iteration element. Throws NoSuchElementException if no more elements
  - `` `void remove()` `` - Removes the last element returned by this iterator (optional operation)
- Iterable:
  - `` `Iterator<T> iterator()` `` - Returns an iterator over a set of elements of type T

Specifiche **semantiche**

- Serve dire su cosa si itera (es: esami dello studente) e magari come (es: ordine crescente)
- Per Standalone `Iterator` le specifiche vanno in `OVERVIEW` e sui metodi della classe (se serve)
- Per `Iterable` scrivere le specifiche sul metodo che produce `Iterator` (anche dei suoi metodi)
  - **Effects** dovrà includere una parte sul **comportamento dell'Iterator**
  - Le clausole **Requires** e **Modifies** riferiti all'Iterator vanno alla fine

Specifica e Implementazione degli Iteratori

Implementazione Standalone `Iterator`

- Modella i dati su cui si itera e un indicatore dello stato di iterazione.
  - `hasNext()` è un observer e deve solo dire se ci sono altri elementi rimasti su cui iterare
  - `next()` restituisce l'elemento successivo e aggiorna lo stato di iterazione al prossimo (o eccezione)

```
1  public class Range implements Iterator<Integer> {
2  //OVERVIEW: standalone iterator che produce int partendo da min e arrivando a max
3    private int min, max, step, current;
4
5    public Range(int min, int max) throws IllegalArgumentException {
6    //MODIFIES: this
7    //EFFECTS: inizializza iteratore con min max e step
8      if(min > max)
9        throw new IllegalArgumentException("min > max");
10
11     this.min = this.current = min;
12     this.max = max;
13   }
14
15   @Override
16   public boolean hasNext() {
17     return current < max;
18   }
19
20   @Override
21   public Integer next() {
22     if(!this.hasNext())
23       throw new NoSuchElementException("No other elements in the range");
24
25     return current++;
26   }
```

Specifica e Implementazione degli Iteratori

Implementazione `iterator()` per delega

- Spesso serve un `Iterator` agli elementi di un attributo che è una `Collection`
  - `Collection` implementano Iterable, quindi possono produrre un Iterator chiamando il loro `iterator()`
  - possibile usare questo `Iterator`, ottenendo una semplice implementazione per delega

```
1  public class Studente implements Iterable<String> {
2  //OVERVIEW: modella uno studente con nome/cognome, matricola e esami dati
3    final String nome, cognome;
4    final int matr;
5    HashMap<String, Integer> esamiDati = new HashMap<>();
6
7  ...
8
9    @Override
10   public Iterator<String> iterator() {
11   //EFFECTS: restituisce un iteratore sui nomi degli esami dati dallo Student
12   //REQUIRES: this non deve essere modificato mentre l'iteratore in uso
13   //MODIFIES: this (serve perche' Iterator restituito ha il metodo remove())
14
15     return esamiDati.keySet().iterator();
16   }
17 }
```

Specifica e Implementazione degli Iteratori

Perchè creare un nuovo iteratore?

Spesso possibile restituire l'iteratore default dell'attributo per delega. Perché non va bene?

- Gli iterator predefiniti delle List (e altri) hanno il metodo **remove**
  - Utilizzare l'iteratore dell'attributo stesso **espone la rappresentazione**
- Se l'iteratore default non mi va bene
  - IntSet con rappresentazione su ArrayList con doppioni
- Nel caso di Standalone `Iterator` **non c'è un iteratore default**.
  - Es: contatore ingressi
  - Es: iteratore che stampa numeri primi
- Per creare un nuovo iteratore serve definire una classe nuova:
  - Che **implementi l'interfaccia Iterator**
  - Che abbia **accesso alla rappresentazione** del dato della Classe
  - Che tenga traccia dello stato dell'iterazione (con degli appositi attributi)

Programmazione II

11. Astrazione Iterazione (Uso, Specifica, Implementazione) (PDJ 6-6.4)

Dragan Ahmetovic