Programmazione Orientata agli Oggetti in Linguaggio Java

Ereditarietà e Polimorfismo: Polimorfismo - a

Programmare con il Polimorfismo

versione 1.2

G. Mecca – Università della Basilicata – mecca@unibas.it

Sommario

- Il Problema
  - ⇨Analisi del Tipo
- La Soluzione Ideale
  - ⇨Programmare con il Polimorfismo
  - ⇨Questioni Aperte

Il Problema

- In questa lezione
  - ci concentriamo sul secondo tipo di ereditarietà: l'ereditarietà di tipo
  - e sul fenomeno collegato del polimorfismo
- Ma, per cominciare
  - da dove nasce la necessità di utilizzare il polimorfismo nei linguaggi OO ?
  - nasce da un problema collegato all'uso degli oggetti

Il Problema

- Il Problema
  - molto frequentemente oggetti di classi diverse devono essere usati assieme
  - esempio tipico: le collezioni (array, liste, matrici)
  - questo succede spesso nel caso di gerarchie, ma può succedere anche con oggetti che non appartengono a gerarchie

Il Problema

- Esempio n. 1: gestione appuntamenti
  - ⇨per ogni giorno, ho bisogno di tenere traccia della lista degli impegni
  - ⇨gli impegni possono essere lezioni o riunioni
- Esempio n. 2: volpi e conigli
  - ⇨la scacchiera deve contenere riferimenti sia a volpi sia a conigli
  - ⇨ovvero a oggetti di tipi diversi

Il Problema

- Una possibile soluzione
  - ⇨tenere separati gli oggetti di tipi diversi
- Esempio: appuntamenti
  - ⇨una lista separata per ciascun tipo di impegno (lista di riunioni, lista di lezioni ecc.)
  - ⇨es: lista di lezioni e lista di riunioni

```
// una possibile soluzione

public class Giorno {

    private java.util.ArrayList listaRiunioni;
    private java.util.ArrayList listaLezioni;

    public void addRiunione(Riunione riunione) {...}
    public void addLezione(Lezione lezione) {...}

    public void removeRiunione(int i) {...}
    public void removeLezione(int i) {...}

    public int getNumeroRiunioni() {...}
    public int getNumeroLezioni(){...}

    public Riunione cercaRiunione(Orario orario)  {...}
    public Lezione cercaLezione(Orario orario) {...}

    ....
}
```


Il Problema

- Il grande svantaggio
  - ⇨il codice è sostanzialmente duplicato
  - ⇨se aggiungo un nuovo tipo di impegno, devo addirittura triplicarlo
- Nel caso di Volpi e Conigli
  - ⇨la programmazione si complica decisamente
  - ⇨dovrei tenere due matrici diverse o due liste con posizioni sulla scacchiera

Il Problema

- Regola generale
  - ⇨duplicare il codice è molto male
  - ⇨adottare soluzioni complicate è malissimo
  - ⇨in entrambi i casi si complica la manutenzione
- Di conseguenza
  - ⇨dobbiamo scartare questa prima soluzione
  - ⇨gli svantaggi sono eccessivi

Il Problema

- Una soluzione alternativa
  - ⇨sfrutto la proprietà di “genericità” delle collezioni di Java (anche se non ho ancora capito bene come funzionano realmente)
- In questo caso
  - ⇨definisco un’unica collezione di oggetti di tipo diverso
  - ⇨tutti trattati in realtà come istanze di Object

```
// una possibile soluzione

public class Giorno {

    private java.util.ArrayList listaImpegni

    public void addImpegno(Object o) {...}

    public void removeImpegno(int i) {...}

    public int getNumeroImpegni() {...}

    public Object cercaImpegno(Orario orario)  {...}

    ....
}
```


Il Problema

- Ma nascono altri problemi
  - ⇨è necessario effettuare cast quando prelevo gli oggetti dalla collezione; come fare il cast corretto ?
- Esempio
  - ⇨la stampa della lista di impegni
  - ⇨il metodo cercaImpegno()
- Uno strumento per analizzare il tipo
  - ⇨l'operatore instanceof

Analisi del Tipo

- instanceof
  - ⇨operatore binario che si applica ad un riferimento e ad un nome di classe
  - ⇨restituisce true se il riferimento è ad un oggetto della classe specificata
- Esempi
  - ⇨Lezione l = new Lezione();
  - ⇨System.out.println(l instanceof Lezione); // true
  - ⇨System.out.println(l instanceof Riunione); // false

Analisi del Tipo

- In questo modo
  - ⇨posso analizzare i riferimenti estratti dalle collezioni “miste” ed effettuare i cast corretti
- Un esempio
  - ⇨un metodo che visualizza sullo schermo la lista degli impegni di un certo giorno

Analisi del Tipo

```
public void stampa (ArrayList lista) {
      for (int i = 0; i < lista.size(); i++) {
         Object o = lista.get(i);
         if (o instanceof Riunione) {
            Riunione r = (Riunione)o;
            System.out.println("Riunione: " + r.getDescrizione());
         } else if (o instanceof Lezione) {
            Lezione l = (Lezione)o;
            System.out.println("Lezione: " + l.getCorso());
         }
      }
   }
```


Analisi del Tipo

- Cosa non va in questa soluzione ?
  - ⇨la presenza dell'if
  - ⇨se vengono introdotti altre tipologie di impegno, l'if diventa più ramificato
  - ⇨inoltre, un if simile dovrebbe essere ripetuto dovunque, nel codice, ci sia bisogno di lavorare con gli impegni
  - ⇨es: il metodo cercaImpegno per orario

Analisi del Tipo

```
public void stampaDatiImpegno (Orario orario, Giorno giorno) {
        Object o = giorno.cercaImpegno(Orario);
        if (o != null) {
           if (o instanceof Riunione) {
              Riunione r = (Riunione)o;
              System.out.println("Riunione: " + r.getDescrizione());
           } else if (o instanceof Lezione) {
              Lezione l = (Lezione)o;
              System.out.println("Lezione: " + l.getCorso());
           }
        }
}
```


Analisi del Tipo

- Altra regola generale
  - ⇨questo tipo di if è male
  - ⇨è dimostrato che l'utilizzo di if molto ramificati è una fonte di problemi nella manutenzione del codice
  - ⇨lo rende meno leggibile
  - ⇨e aumenta la probabilità di commettere errori
  - ⇨quindi, è opportuno evitarli

La Soluzione Ideale

- Idealmente
  - ⇨vorrei poter adottare un'altra soluzione
  - ⇨basata sulle seguenti osservazioni
- Osservazione a:
  - ⇨per quanto diversi, tutti i tipi di impegno hanno comunque la caratteristica comune di essere impegni

La Soluzione Ideale

- Osservazione b:
  - ⇨nello spirito della programmazione a oggetti, è opportuno che ciascun impegno sappia trasformare sè stesso in stringa
- In altri termini
  - ⇨vorrei poter scrivere il metodo stampa come segue

Nota: nel seguito il codice è semplificato rispetto a quello delle applicazioni reali

La Soluzione Ideale

```
public static void stampa (ArrayList lista) {
    for (int i = 0; i < lista.size(); i++) {
        Impegno impegno = (Impegno)lista.get(i);
        System.out.println(impegno.toString());
    }
}
```

L'output atteso:
- riunione con Marco
- lezione di Prog. Oggetti
- riunione della CIP

gli oggetti prelevati dalla lista sono trattati come oggetti di tipo "Impegno" indifferentemente dalle differenze di tipo

a ciascun oggetto viene chiesto di eseguire il metodo toString() nel modo più appropriato per il tipo di impegno

La Soluzione Ideale

- Cosa succede nel metodo stampa ?
  - ⇨sto in effetti manipolando oggetti di tipo diverso (riunioni, lezioni ecc.) senza preoccuparmi del loro tipo
  - ⇨faccio riferimento ad un tipo più generale (Impegno)
  - ⇨e confido che ciascun oggetto sappia stampare i propri dati nel modo più appropriato

La Soluzione Ideale

- Nel caso di volpi e conigli
  - ⇨vorrei poter scrivere il metodo che effettua la simulazione come segue

```
public void simula(Scacchiera scacchiera) {
    for (int i = 0; i < scacchiera.getRighe(); i++) {
        for (int j = 0; j < scacchiera.getColonne(); j++) {
            Animale animale = (Animale)scacchiera.getElemento(i, j);
            if (animale != null) {
                animale.agisci(scacchiera);
            }
        }
    }
}
```


La Soluzione Ideale

- Anche in questo caso
  - ⇨ho identificato un “tipo” comune per gli oggetti della scacchiera: Animale
  - ⇨tratto tutti gli oggetti (che in questo caso non fanno realmente parte di una gerarchia) come oggetti del tipo comune
  - ⇨mi affido a ciascun oggetto perché esegua correttamente il metodo agisci()

Programmare con il Polimorfismo

ATTENZIONE
allo stile di programmazione con il polimorfismo

- L'intuizione
  - ⇨"utilizzare gli oggetti senza conoscerne realmente il tipo"
  - ⇨definire tipi generali comuni a più classi di oggetti
  - ⇨definirne implementazioni diverse per i metodi
  - ⇨lasciare che ciascun oggetto scelga l'implementazione corretta dei metodi

Programmare con il Polimorfismo

```
public static void stampa (ArrayList lista) {
    for (int i = 0; i < lista.size(); i++) {
        Impegno impegno = (Impegno)lista.get(i);
        System.out.println(impegno.toString());
    }
}

public void simula(Scacchiera scacchiera) {
     for (int i = 0; i < scacchiera.getRighe(); i++) {
          for (int j = 0; j < scacchiera.getColonne(); j++) {
               Animale animale = (Animale)scacchiera.getElemento(i, j);
               if (animale != null) {
                    animale.agisci(scacchiera);
               }
          }
     }
}
```

Programmare con il Polimorfismo

- Vantaggi di questo approccio
  - ho eliminato gli if da questo e da molti altri punti del programma
  - le applicazioni diventano estensibili
  - posso facilmente introdurre nuove tipologie di impegno (es: pranzi di lavoro, seminari ecc.)
  - o nuove categorie di animali

Questioni Aperte

- Prima di raggiungere l'obiettivo, però
  - dobbiamo risolvere un paio di problemi
- Primo problema
  - cosa vuol dire definire un tipo più generale per gli oggetti coinvolti ?
  - es: Impegno
  - es: Animale (attenzione: in questo caso non c'è ereditarietà su cui contare)

Questioni Aperte

- Secondo problema
  - ⇨com'è possibile che chiamate dello stesso metodo su riferimenti dello stesso tipo producano risultati diversi
  - ⇨es: toString() di Lezione vs. toString() di Riunione
  - ⇨es: agisci() di Volve vs agisci() di Coniglio

Riassumendo

- Il Problema
  - ⇨Analisi del Tipo
- La Soluzione Ideale
  - ⇨Programmare con il Polimorfismo
  - ⇨Questioni Aperte

Termini della Licenza