Programmazione II

9. Gerarchia dei Tipi (Gerarchia, Dispatching, Classi Astratte, Interfacce) (PDJ 7-7.7)

Gerarchie delle classi

Estensione di una classe

- Una classe A può **estendere** una classe B, aggiungendo attributi di stato o comportamenti
  - Si dice che $A \prec B$ (**A è sottotipo di B**). É una proprietà **transitiva**: se $A \prec B$ e $B \prec C$ allora $A \prec C$
  - Tutte le classi sono sottotipo di **Object** (Classe in cima alla gerarchia)
- A una variabile b di tipo B è sempre possibile assegnare un valore di tipo $A \prec B$
  - Si dice che il **Tipo Apparente** di b è B mentre il **Tipo Concreto** è A

A cosa serve?

- Per specializzare il comportamento
  - Ovvero sostituire il comportamento del tipo base con uno più appropriato
- Per estendere il comportamento
  - Ovvero aggiungere nuove competenze (metodi) a quelli esistenti per il tipo base
- Per permettere diverse implementazioni con lo stesso comportamento
  - Con possibilmente diverse caratteristiche di performance, uso memoria etc
- Per strutturare le classi secondo un'organizzazione ontologia.
  - Ovvero classi con lo stesso comportamento ma significato diverso dato dal nome (es: eccezioni)

Gerarchie delle classi

Esempio Comportamento Specializzato

```
1  public class Bird {
2    public String call() {
3      return "chirp";
4    }
5  }
6
7  public class Crow extends Bird {
8    @Override
9    public String call() {
10     return "gawk";
11   }
12 }
```

Esempio Comportamento Esteso

```
1  public class Point {
2    double x, y;
3
4    public double distanceFrom(Point p) {
5      return Math.sqrt(Math.pow(this.x-p.x)+Math.pow(this.y-p.y));
6    }
7  }
8
9  public class ColoredPoint extends Point {
10   int r,g,b;
11
12   public boolean isBrighter(ColoredPoint p) {
13     return (this.r+this.g+this.b) - (p.r+p.g+p.b) > 0;
14   }
15 }
```

Gerarchie delle classi

Esempio Implementazioni Diverse

```
interface Poly {
  public Poly add(Poly p);
  public Poly sub(Poly p);
  public Poly mul(Poly p);
}

public class PolyDense implements Poly {
  []int constants; // indice = grado, valore = la costante corrispondente

  ... //deve implementare tutti i metodi
}

public class PolySparse implements Poly {
  Map<Integer,Integer> poly; //chiave = grado, valore = la costante corrispondente

  ... //deve implementare tutti i metodi
}
```

Esempio Organizzazione Ontologica

```
public class CalcoloPretendenteVincitoreInCorsoException extends Exception {
  public CalcoloPretendenteVincitoreInCorsoException() {
    super();
  }

  public CalcoloPretendenteVincitoreInCorsoException(String s) {
    super(s);
  }
}
```

Principio di sostituzione di Liskov

Principio di sostituzione

- Una classe $T \prec S$ deve essere utilizzabile ovunque sia usata S
  - Si può assegnare a una **variabile** o a un **parametro formale** $b \in B$ un oggetto $a \in A$ se $A \prec B$
  - Un oggetto di A deve essere sempre in grado di sostituire (con comportamenti) un oggetto di B

Per soddisfare il principio di sostituzione di Liskov, per un dato programma devono essere valide:

- **Regola delle Segnature**
  - Per ogni metodo m di S, ci deve essere un metodo con la stessa intestazione in $T \prec S$
- **Regola dei Metodi**
  - Un metodo m di $T \prec S$ deve avere lo stesso comportamento del metodo m di S che ridefinisce
- **Regola delle Proprietà**
  - Il sottotipo $R \prec S$ deve conservare tutte le proprietà che possono essere dimostrate per S

Principio di sostituzione di Liskov

Regola delle Segnature

Per ogni metodo m di S, **ci deve essere** un metodo m' con la **stessa intestazione** in $T \prec S$

- La segnatura del metodo m' deve essere **compatibile** con la segnatura del metodo m
  - Devono avere lo **stesso nome, stessi tipi* di parametri e di ritorno**
  - Il tipo di ritorno può essere **covariante**, ovvero un **sottotipo del tipo di ritorno della superclasse**
  - Per Liskov i parametri potrebbero essere controvarianti (meno specifici)
    Non supportato in Java* (e nemmeno C++ e altri) perchè poco utile (posso fare overload)
- Il metodo del sottotipo **può avere meno eccezioni o eccezioni di tipo più specifico**
  - ovvero **possono essere sottotipi** di eccezioni già lanciate dalla superclasse
  - questo perchè **la sottoclasse può gestire o rilanciare** eccezioni non gestite nella superclasse
- Questa regola è **verificabile direttamente dal compilatore**
  - Le prossime due regole **non possono essere verificate in maniera automatica**

Principio di sostituzione di Liskov

Regola dei Metodi

Un metodo m' di $T \prec S$ deve avere lo **stesso comportamento** del metodo m di S che ridefinisce

- Se si usa il metodo m' si deve poter usare gli stessi input e ottenere lo stesso effetto di m'.
- Le **pre-condizioni** (**REQUIRES**) del sottotipo possono essere **indebolite**
  - $pre_m \implies pre_{m'}$
  - Ovvero, m' **non può escludere input** che m avrebbe permesso
  - Ma **potrebbe permettere altri input** oltre a quelli del metodo originale
  - es: m permette interi positivi
    m' permette interi
- Le **post-condizioni** (**EFFECTS**) del sottotipo **possono essere rafforzate**
  - $(pre_m \wedge post_{m'}) \implies post_m$
  - Ovvero, per un input valido per m, l'output di m' **deve rispettare i vincoli dell'output** di m
  - Inoltre **possono valere altri vincoli** e **per altri input** validi per m' il comportamento **non è vincolato**
  - es: m restituisce un array di interi
    m' restituisce un array di interi ordinato

Principio di sostituzione di Liskov

Regola delle Proprietà

Il sottotipo $T \prec S$ **deve conservare tutte le proprietà** che possono essere dimostrate per S

- In T **devono rimanere valide le invarianti** di S
  - Ovvero **condizioni sullo stato** dell'oggetto valide per S devono valere anche per T
  - es: una classe che estende IntSet non può comunque permettere elementi ripetuti
  - L'**invariante di rappresentazione** è la **congiunzione delle invarianti**
- In T **devono rimanere valide le proprietà evolutive** di S
  - Ovvero, **i cambi di stato permissibili** in T sono gli stessi di S (per gli attributi condivisi)
  - es: si assume che uno Studente non possa cancellare un esame già sostenuto
    allora una classe che estende Studente non può similmente eliminare esami sostenuti

Tipi di Ereditarietà

Meccanismi per estendere i tipi di dati

Ci sono due meccanismi per estendere i tipi di dati:

- Estensione di **una** classe (superclasse)
  - Possibilmente modificando alcuni metodi (ma non il loro "**comportamento**")
  - Aggiungendo nuovi metodi e nuovi attributi
  - Sintassi: `` `class C extends E` ``
- Implementazione di **una o più** interfacce
  - Un'**interfaccia** è un **elenco di intestazioni di metodi** senza una **l'implementazione**
  - Una classe che **implementa** un'interfaccia **deve implementare i suoi metodi**
  - L'implementazione di un metodo di interfaccia è considerato **@Override**
  - Sintassi: `` `class C implements E` ``

Tipi di Ereditarietà

Classi concrete

Solitamente le classi sono **classi concrete** e definiscono un **tipo istanziabile**. Possono avere:

- Attributi e metodi `nuovi`, **definiti e implementati** solamente all'interno della classe
  - Questi **non esistono nella superclasse** e devono essere completamente definiti nella sottoclasse
- Attributi e metodi **ereditati** dalla propria superclasse
  - Questi non sono definiti nella classe stessa ma **usano la definizione data nella superclasse**
  - Per il principio di sostituzione di Liskov funzionano comunque per la sottoclasse
- Metodi ridefiniti (**override**)
  - **Sostituiscono il comportamento** del metodo della superclasse
  - Possono comunque riferirsi all'implementazione della superclasse con la keyword `**super**`
- Metodi dichiarati **final**, che non possono essere estesi.
  - Anche le classi possono essere dichiarate final e quindi non estendibili

Tipi di Ereditarietà

Esempio Definizione Classi Concrete

```java
public class Studente {
//OVERVIEW: classe che modella uno Studente
    int matricola;
    String nome;
    ArrayList<String> corsi;

    Studente(int matricola, String nome) {
    //EFFECTS: inizializza this con matricola e nome
        this.matricola = matricola;
        this.nome = nome;
        this.corsi = new ArrayList<String>();
    }

    public String getNome() {
    //EFFECTS: restituisce il nome dello studente
        return this.nome;
    }

    public int getMatricola() {
    //EFFECTS: restituisce la matricola dello studente
        return this.matricola;
    }

    public boolean frequenta(String corso) {
    //EFFECTS: restituisce il verso del topo
        return this.corsi.contains(corso);
    }
}
```

Tipi di Ereditarietà

Esempio extend Classi Concrete

```
1  public class StudenteMagistrale extends Studente {
2  //OVERVIEW: classe che modella uno Studente Magistrale
3      int votoLaurea;
4
5      public StudenteMagistrale(int matricola, String nome, int votoLaurea) {
6      //EFFECTS: inizializza nuovo StudenteMagistrale con matricola, nome e votoLaurea
7          super(matricola, nome);
8          this.votoLaurea = votoLaurea;
9      }
10
11     public int getVotoLaureaTriennale() {
12     //EFFECTS: restituisce il voto di laurea triennale dello studente
13         return this.votoLaurea;
14     }
15 }
```

Tipi di Ereditarietà

Classi astratte

Oltre a **classi concrete** che definiscono e implementano un tipo, esistono anche **classi astratte**

- Sintassi: `` `abstract class C` ``
- Una classe astratta **non fornisce un'implementazione completa** di tutti i suoi metodi
  - Alcuni metodi possono essere forniti **solo come intestazione**
  - Sono preceduti dalla keyword **abstract**
- Serve **per forza una sottoclasse** che la estenda, completando i metodi mancanti
- I **metodi già implementati** sono comportamenti generici validi in diverse sottoclassi
  - Servono per **ridurre ridondanza di codice** tra diverse sottoclassi
  - È possibile **conservare i metodi già implementati** o ridefinirli
- Implementare metodi mancanti potrebbe richiedere **cambi su attributi** della classe astratta
  - L'inerente **esposizione della rappresentazione** rende questo meccanismo **problematico**
  - Neccessarie repOk apposite nella classe astratta e nella estesa

Tipi di Ereditarietà

Esempio Definizione Classi Astratte

```
1  abstract class Mammifero {
2  //OVERVIEW: classe astratta che modella un mammifero
3      int eta; //giorni
4      double peso; //grammi
5
6  //metodi concreti
7      public double getPeso() {
8      //EFFECTS: restituisce il peso dell'animale in grammi
9          return peso;
10     }
11
12     public int getEta() {
13     //EFFECTS: restituisce l'eta' dell'animale in giorni
14         return eta;
15     }
16
17 //metodi astratti
18     public abstract String verso();
19     //EFFECTS: restituisce il verso dell'animale
20 }
```

Tipi di Ereditarietà

Esempio extend Classi Astratte

```
1  public class Topo extends Mammifero {
2  //OVERVIEW: classe che modella un topo, sottotipo di Mammifero
3      String colore;
4
5      public String getColore() {
6      //EFFECTS: restituisce il colore del topo
7              return this.colore;
8      }
9
10     @Override
11     public String verso() {
12     //EFFECTS: restituisce il verso del topo
13         return "squik";
14     }
15 }
```

Tipi di Ereditarietà

Interfacce

Le interfacce sono simili alle classi abstract:

- Contengono **intestazioni dei metodi senza la loro implementazione**
  - Questi metodi devono essere implementati nella classe che implementa l'interfaccia
- Ci possono essere dei **metodi opzionali** che non è necessario implementare
  - Per modellare comportamenti validi per tutte le classi che implementano l'interfaccia
  - Possibile solo un'**implementazione per comportamento**
  - Possono servire per modellare **metodi opzionali**
  - In tal caso implementazione default è lancio di un eccezione (UnsupportedOperationException)
- **Non hanno una rappresentazione** propria, più sicure delle classi astratte
- A differenza delle classi, è possibile **implementare più interfacce**

Tipi di Ereditarietà

Esempio Definizione Interfacce

```
1  interface Poligono {
2  //OVERVIEW: Interfaccia che definisce i comportamenti di un poligono
3
4  //metodi astratti
5      double getPerimetro();
6      //EFFECTS: restituisce il perimetro del poligono
7
8      int getFacce();
9      //EFFECTS: restituisce il numero delle facce del poligono
10
11     boolean regolare();
12     //EFFECTS: restituisce true se il poligono e' regolare, false altrimenti
13
14 //metodi default
15     double getLato() throws UnsupportedOperationException {
16     //EFFECTS: restituisce la lunghezza di un lato nel caso di un poligono regolare
17         throw new UnsupportedOperationException("metodo valido solo per poligoni regolari")
18     }
19 }
```

Tipi di Ereditarietà

Esempio implement Interfacce

```java
public class TriangoloEquilatero implements Poligono {
//OVERVIEW: modella un triangolo equilatero
    double lato;

    @Override
    public double getPerimetro() {
        return lato*3;
    }

    @Override
    int getFacce() {
        return 3;
    }

    @Override
    boolean regolare() {
        return true;
    }

    @Override
    double getLato() {
        return this.lato;
    }
}
```

Tipi di Ereditarietà

Ereditarietà Multipla

Per **ereditarietà multipla** si intende la possibilità di una classe di **ereditare da più superclassi**

- Ereditarietà di classi (**extends**) **non può essere multipla**
  - Si può **estendere una sola superclasse** (concreta o astratta che sia)
- Ereditarietà delle interfacce (**implements**) **può essere multipla**
  - È **possibile implementare più interfacce** contemporaneamente
- È **possibile** inoltre **estendere una classe** e **anche implementare delle interfacce**
- Cosa fare nel caso dell'ereditarietà multipla con **stesso metodo già implementato**?
  - **Override ha la precedenza** su tutto
  - **Metodo concreto** ereditato da classe ha la **precedenza su metodi default** di interfacce
  - Compile error se due interfacce hanno lo **stesso metodo default** (e non c'è Override)

Programmazione II

9. Gerarchia dei Tipi (Gerarchia, Dispatching, Classi Astratte, Interfacce) (PDJ 7-7.7)

Dragan Ahmetovic