Programmazione II

10. Gerarchia dei Tipi (Specifica, Ereditarietà, Composizione) (PDJ 7.8-)

Dragan Ahmetovic

Specifiche delle Gerarchie dei Tipi

Visibilità

Sono possibili modificatori di visibilità delle Classi, Attributi e Metodi

- Visibilità **public**
  - Visibile a **tutte le altre classi** e i loro metodi
  - Per gli attributi è pericoloso perchè si **espone la rappresentazione**
- Visibilità **protected** (metodi e attributi, non ha senso per le classi)
  - Visibile a **tutte le sottoclassi** (e a **tutte le classi del package**)
  - Per gli attributi è comunque pericoloso perchè si **espone la rappresentazione**
- Visibilità **package** (non si usa una keyword)
  - Visibile a **tutte le classi del package**, utile se unico sviluppatore o team
- Visibilità **private** (metodi, attributi, e classi/interfacce nested)
  - Visibile **solo nella classe stessa**
  - Dovrebbe essere usato per **tutti gli attributi** e per i **metodi di utilità interni**

Specifiche delle Gerarchie dei Tipi

Come creare specifiche per le gerarchie di tipi

Una classe che estende un'altra ne **modifica le caratteristiche**

Tali modifiche **devono essere specificate** in maniera appropriata:

- Se viene **modificato il comportamento dei metodi**
  - È necessario **specificare il nuovo comportamento** e in che cosa cambia
- Se vengono **aggiunti nuovi metodi**
  - È necessario **specificare il loro funzionamento**
- L'**overview** della classe deve **cogliere l'obiettivo semantico diverso**
  - In particolare questo è fondamentale per le **strutturazioni ontologiche**

Specifiche delle Gerarchie dei Tipi

Invariante di Rappresentazione

Nel caso di un sottotipo l'**Invariante di Rappresentazione** dovrà considerare se:

- Il sottotipo ha degli **attributi nuovi**
  - in tal caso la **repOk** del sottotipo deve controllare la loro validità in base all'astrazione considerata
- Il sottotipo **modifica gli attributi del supertipo**.
  - In tal caso dovrà richiamare la **repOk** del supertipo (usando `**super.repOk()**`)

Funzione di Astrazione

Dato che vige il **principio di sostituzione di Liskov**

- Il sottotipo **non deve stravolgere l'astrazione** usata dal supertipo
- Conservarla il più possibile, estendendola con ulteriori informazioni del caso

Astrazioni ontologiche errate

Astrazione ontologica e di tipo

Il rapporto tra una superclasse e una sottoclasse è concettualmente un rapporto `**IS-A**`

- Il **membro di una sottoclasse** è **anche un membro della sua superclasse**
- Possono però esserci discrepanze tra l'**astrazione ontologica** e l'**astrazione di tipo**
  - Ontologicamente una classe *A* è una sottoclasse di una classe *B*
  - Es: il quadrato è un retangolo coi lati uguali
- Lo è anche dal punto di vista di **object orientation**? **Rispetta Liskov Substitution Principle**?
  - Il metodo `` `setBase(double b)` `` ha la stessa intestazione
  - Però il **comportamento è diverso**! il quadrato imposta sia base che l'altezza a b
- **Non rispetta Liskov Substitution Principle**!
  - In particolare la regola dei metodi

Astrazioni ontologiche errate

Esempio di ereditarietà errata

```
1  public class Rectangle {
2    private int base, height;
3
4    public Rectangle(int base, int height) {
5      this.base = base;
6      this.height = height;
7    }
8
9    public void base(int base) {
10     this.base = base;
11   }
12
13   public void height(int height) {
14     this.height = height;
15   }
16
17   public int area() {
18     return this.base * this.height;
19   }
20 }
```

Astrazioni ontologiche errate

Esempio di ereditarietà errata

```
1  public class Square extends Rectangle {
2
3    public Square(int base) {
4      super(base, base);
5    }
6
7    @Override
8    public void base(int base) {
9      super.base(base);
10     super.height(base);
11   }
12
13   @Override
14   public void height(int height) {
15     base(height);
16   }
17
18   public static void main(String[] args) {
19     Rectangle r = new Square(5);
20     r.height(6);
21     r.base(3);
22     System.out.println(r.area());
23   }
24 }
```

Encapsulation e Ereditarietà multipla

Encapsulation nell'ereditarietà

L'**incapsulamento** (**encapsulation**) della rappresentazione per non esporla è fondamentale
L'incapsulamento però può essere rotto per permettere l'estensione della classe.

Ci sono diverse strategie per evitare di esporre la rappresentazione:

- Utilizzare la **visibilità più stringente** possibile
  - Se posso cerco di cavarmela con attributi definiti private
- Non avere uno stato nella superclasse
  - Vincolo già **soddisfatto** nel caso delle **interfacce**
- Implementare i metodi **per comportamento**
  - Non agendo **direttamente** sulla rappresentazione ma implementare usando solo chiamate ai metodi
  - Metodi della superclasse possono usare i metodi ridefiniti nella sottoclasse (es: interfacce)
  - Metodi della sottoclasse possono usare i metodi della superclasse senza esporre la rappresentazione

Dispatching

Dispatching nell'ereditarietà

È fondamentale comprendere il **meccanismo di dispatching** nel caso dell'**ereditarietà**

- **Tipo Apparente** può essere una **superclasse del Tipo Concreto**
- L'**esistenza di un metodo** viene valutata **in base al Tipo Apparente** durante la **compilazione**
  - Possono essere eseguiti **solo i metodi presenti nel Tipo Apparente** (o i loro Override)
  - Il metodo eseguito prioritariamente è un eventuale **Override nel Tipo Concreto**
  - Altrimenti viene chiamato il metodo del Tipo Apparente
- I **metodi sovraccaricati** sono accessibili **solo se presenti anche nel Tipo Apparente**
  - Attenzione alle conversioni implicite
  - es: se B b = new A() e sono definiti i metodi B.m(double) e A.m(int)
    b.m(10) chiamerà il metodo definito in B
- Attenzione: un metodo di superclasse **può chiamare un metodo ridefinito** nella sottoclasse!
  - In tal caso il metodo della sottoclasse **potrebbe generare un comportamento imprevisto**

Dispatching

Esempio di ereditarietà e dispatching

```
1  public class Superclass {
2    public void f(double x) {
3      System.out.println("Superclass::f(double)");
4    }
5  }
6
7  public class Subclass extends Superclass {
8    public void f(int x) {
9      System.out.println("Subclass::f(int)");
10   }
11
12   public static void main() {
13     Superclass s = new Subclass();
14
15     s.f(1.0); //Superclass::f(double)
16     s.f(1); //Superclass::f(double) //il metodo con parametro int non esiste in Superclass
17     //L'unica cosa chiamabile e' il metodo con parametro double (con conversione implicita)
18   }
19 }
```

Dispatching

Esempio di ereditarietà e dispatching

```java
public class Adder {
  private int result = 0;

  public void add(int x) {
    result += x;
  }

  public void addAll(List<Integer> l) {
    for (int x : l)
        add(x); //se usato da LogAdderExt il metodo chiama add di LogAdderExt
  }
}

public class LogAdderExt extends Adder {
  private List<Integer> seen = new ArrayList<>();

  @Override
  public void add(int x) {
    seen.add(x);
    super.add(x);
  }

  @Override
  public void addAll(List<Integer> l) {
    seen.addAll(l);
    super.addAll(l); //questo metodo richiama Adder.addAll(l)
  }
}
```

Uguaglianza nell'ereditarietà

Ereditarietà e uguaglianza

Un problema che sorge con l'uso dell'ereditarietà è la verifica dell'**uguaglianza di due oggetti**

- `s.equals(t)` giudica vera l'uguaglianza di un oggetto $s \in S$ con un oggetto $t \in T \prec S$
  - Infatti il controllo '`t instanceOf S`' e i controlli sugli attributi restituiscono 'true'
- Tuttavia viene **infranta la proprietà simmetrica** dell'uguaglianza, infatti `t.equals(s)` da 'false'
  - Infatti `s instanceOf T` restituisce false e inoltre s **può non avere tutti gli attributi** di t
- Controllo di tutti gli attributi se `s instanceOf T`, altrimenti solo quelli di S **rompe la transitività**
  - Infatti per un $r \in T$ può essere che '`t.equals(s)`' e '`r.equals(s)`' ma che non sia vero '`t.equals(r)`'
- `a.getClass()` **restituisce la classe concreta**. Al posto di `instanceOf` **restinge l'uguaglianza**
  - Questo però **va contro la regola dei Metodi** perchè '`s.equals(s)`' e `s.equals(t)`' non sono uguali
  - Infatti non permette di considerare uguali sottotipi, anche se non modificano la rappresentazione
  - Altra possibilità è dichiarare final equals nel supertipo, e non modificare la rappresentazione

Uguaglianza nell'ereditarietà

Problema dell'uguaglianza tra sottotipi - riflessività

```java
public class S {
  private final int a;

  public S(int a) {
    this.a = a;
  }

  public boolean equals(Object o) { //s.equals(t) == true;
    if(o instanceOf S)
      return ((S)o).a == this.a;
    return false;
  }
}

public class T extends S {
  private final int b;

  public T(int a, int b) {
    super(a);
    this.b = b;
  }

  public boolean equals(Object o) { //t.equals(s) == false; //errore simmetria!
    if (o instanceOf T)
      return super.equals(o) && ((T)o).b == this.b;
    return false;
  }
}
```

Uguaglianza nell'ereditarietà

Problema dell'uguaglianza tra sottotipi - transitività

```java
public class T extends S {
  private final int b;

  public T(int a, int b) {
    super(a);
    this.b = b;
  }

  public boolean equals(Object o) { //t.equals(s)==true;r.equals(s)==true;r.equals(t)==false;//errore trans.
    if (o instanceOf T)
      return super.equals(o) && ((T)o).b == this.b;
    else if (o instanceOf S)
      return super.equals(o);
    return false;
  }
}
```

Problema dell'uguaglianza tra sottotipi - regola dei metodi

```java
public boolean equals(Object o) { //s.equals(s) true; t.equals(s) false; //errore regola dei metodi!
  if (getClass() == o.getClass())
    return ((T)o).a = this.a && ((T)o).b = this.b; //se equals di S usa getClass non posso chiamare super
  return false;
}
```

Composizione

Cos'è la Composizione

Extend con modifica di rappresentazione presenta problemi. Possibile usare la **composizione**

- Invece di estendere una classe *S*, dentro la classe *T* possibile **creare un attributo** *s* di tipo *S*
- I comportamenti implementati nel tipo *S* a *s* sono **usati per delega** da *T*
  - In pratica i metodi di *S* ricreati in *T* non faranno altro che chiamare i metodi di *S*
- Non estendendo *S* **non si viola l'incapsulamento** della rappresentazione di *S*
- Non ci sono problemi di principio di sostituzione di Liskov (**vige solo localmente alla classe**)
- Però i due tipi *S* e *T* **non sono in alcuna relazione** tra di loro
- È possibile creare e usare **un'interfaccia che metta in relazione i due tipi**
  - Entrambi implementeranno l'interfaccia che definisce il comportamento comune
  - In *S* ci sarà l'implementazione completa dei metodi d'interfaccia
  - In *T* l'implementazione dei metodi d'interfaccia farà la delega ai metodi di *s*

Composizione

Esempio composizione

```
1  public class Triangolo implements Poligono {
2      double[3] lati;
3
4      public Triangolo(l1,l2,l3) {
5        this.lati[0] = l1;
6        this.lati[1] = l2;
7        this.lati[2] = l3;
8      }
9
10     @Override
11     public double getPerimetro() {
12         return this.l1+this.l2+this.l3;
13     }
14
15     @Override
16     public int getFacce() {
17         return 3;
18     }
19
20     @Override
21     public boolean regolare() {
22         return false;
23     }
24 }
```

Composizione

Esempio composizione

```
1  public class TriangoloRegolare implements Poligono {
2      Triangolo t; double lato;
3
4      public TriangoloRegolare(l) {
5        this.t = new Triangolo(l,l,l);
6        this.lato = l;
7      }
8
9      @Override
10     public double getPerimetro() {
11         return this.t.getPerimetro();
12     }
13
14     @Override
15     public int getFacce() {
16         return this.t.getFacce();
17     }
18
19     @Override
20     public boolean regolare() {
21         return true;
22     }
23
24     @Override
25     public double getLato() {
26         return this.lato;
27     }
28 }
```

Programmazione II

10. Gerarchia dei Tipi (Specifica, Ereditarietà, Composizione) (PDJ 7.8-)

Dragan Ahmetovic