Programmazione Orientata agli Oggetti in Linguaggio Java

Classi e Oggetti:
Metafora
Parte d

versione 2.2

Questo lavoro è concesso in uso secondo i termini di una licenza Creative Commons
(vedi ultima pagina)

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

Sommario

- Proprietà dei Componenti
- Segmenti
  - Stringhe
  - Messaggi Composti
- Aspetti Metodologici
  - Incapsulamento
  - Massimizzare la Coesione
  - Minimizzare l’Accoppiamento

Proprietà dei Componenti

- Finora
  - ⇨ tutti i componenti visti avevano proprietà dei tipi di base
- Proprietà di un componente
  - ⇨ una variabile o una costante
- Variabili e costanti
  - ⇨ di un tipo di base
  - ⇨ o di tipo riferimento

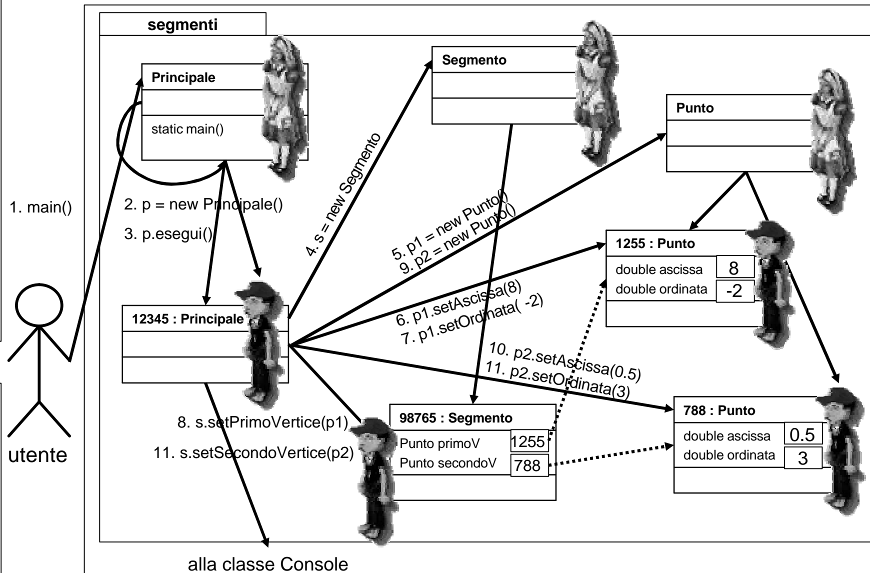
Proprietà dei Componenti

- Di conseguenza
  - ⇨ un componente può avere proprietà di tipo riferimento
- Caso tipico
  - ⇨ un oggetto il cui funzionamento dipende da altri oggetti
  - ⇨ in questo caso l'oggetto può mantenere tra le sue proprietà riferimenti agli oggetti da cui dipende

Segmenti

- Un esempio: analisi di segmenti
  - un programma che analizza segmenti nel piano cartesiano
  - ciascun segmento è rappresentato attraverso le coordinate dei suoi estremi (punti nel piano)
  - il programma calcola il quadrante dei due estremi e la lunghezza del segmento

>> segmentia



Segmenti

- La classe segmento
  - ⇨definisce un’associazione (o aggregazione) di oggetti
- Associazione
  - ⇨corrisponde ad un oggetto che ha tra le sue proprietà riferimenti ad altri oggetti
  - ⇨gruppo di oggetti che è necessario che collaborino strettamente per eseguire i compiti che gli sono affidati

Segmenti

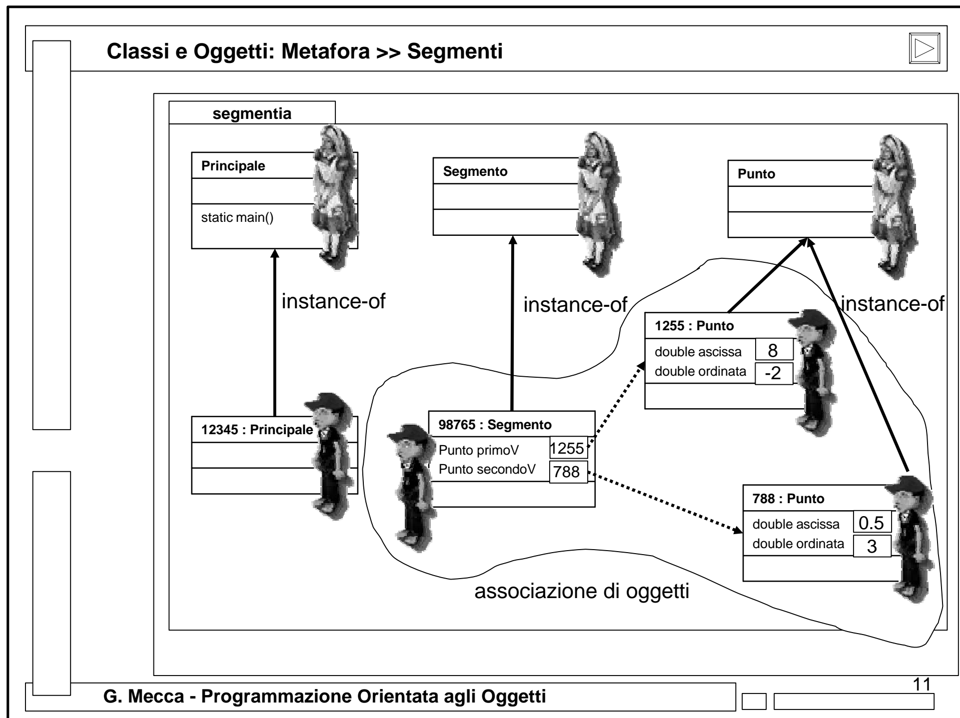
- Metafora
  - ⇨un’associazione di oggetti è una squadra di lavoro
  - ⇨un gruppo di oggetti che deve collaborare più strettamente degli altri per svolgere i propri compiti
  - ⇨le richieste devono essere gestite dall’intera squadra e non possono essere gestite da uno degli oggetti separatamente

Segmenti

- Nell'esempio
  - ⇨l'oggetto segmento, dopo la creazione, non è in grado di svolgere le sue funzioni (calcolare la sua lunghezza)
  - ⇨ha bisogno che venga completata la squadra di lavoro, e cioè che vengano creati i due punti che rappresentano i suoi vertici
  - ⇨a quel punto, utilizzando i due punti, può svolgere i suoi compiti

Segmenti

```
public double getLunghezza() {
    double x1 = this.primoVertice.getAscissa();
    double y1 = this.primoVertice.getOrdinata();
    double x2 = this.secondoVertice.getAscissa();
    double y2 = this.secondoVertice.getOrdinata();
    return Math.sqrt(Math.pow(x1 – x2, 2) +
                     Math.pow(y1 – y2, 2));
}
```



Segmenti

- Alcuni aspetti del codice
  - ⇨utilizzo della classe String
  - ⇨messaggi composti
  - ⇨li vediamo rapidamente nel seguito
- Successivamente
  - ⇨due aspetti metodologici interessanti
  - ⇨incapsulamento
  - ⇨attribuzione delle responsabilità

Stringhe

- La classe java.lang.String
  - le Stringhe di Java sono oggetti
  - la sintassi di Java è volutamente ingannevole: sembra di trattare dati primitivi
  - ma in realtà si manipolano attraverso i riferimenti
- Un'operazione frequente sulle stringhe
  - la concatenazione: +

Stringhe

- Esempio: nella classe Punto
  - il metodo toString(): costruisce una rappresentazione del punto sotto forma di stringa (riferimento ad una stringa)
  - es: "(0.2, 4)"

```
public String toString() {
    return "(" + this.ascissa + ", " + this.ordinata + ")";
}
```

Messaggi Composti

- Messaggi composti
  - i messaggi ai componenti possono essere composti
  - ogni volta che il risultato di un metodo è un riferimento ad un oggetto, sul riferimento è possibile chiamare un altro metodo
  - meccanismo simile a quello della composizione delle funzioni nelle espressioni
  - es: double x = sin(log(y));

Messaggi Composti

il metodo restituisce un riferimento ad un Punto; uso il riferimento per eseguire toString()

```
String stringaPrimoVertice =
        segmento.getPrimoVertice().toString();
```

equivalente a:

```
Punto primoVertice = segmento.getPrimoVertice();
String stringaPrimoVertice = primoVertice.toString();

System.out.println("Quadrante del primo vertice   : " +
   segmento.getPrimoVertice().getQuadrante());
```

equivalente a:

```
Punto primoVertice = segmento.getPrimoVertice();
System.out.println("Quadrante del primo vertice :" +
   primoVertice.getQuadrante();
```

Messaggi Composti

```
private void schermoStampaDatiSegmento(Segmento segmento) {
   System.out.println("\nDati del segmento:");
   System.out.println("----------------------");
   String stringaPrimoVertice = segmento.getPrimoVertice().toString();
   String stringaSecondoVertice = segmento.getSecondoVertice().toString();
   System.out.println("Primo vertice   : " + stringaPrimoVertice);
   System.out.println("Secondo vertice : " + stringaSecondoVertice);
   System.out.println("Quadrante del primo vertice   : " +
                        segmento.getPrimoVertice().getQuadrante());
   System.out.println("Quadrante del secondo vertice : " +
                        segmento.getSecondoVertice().getQuadrante());
   System.out.println("Lunghezza del segmento: " +
                        segmento.getLunghezza());
}
```

Messaggi Composti

```
public double getLunghezza() {
   double x1 = this.primoVertice.getAscissa();
   double y1 = this.primoVertice.getOrdinata();
   double x2 = this.secondoVertice.getAscissa();
   double y2 = this.secondoVertice.getOrdinata();
   return Math.sqrt(Math.pow(x1 - x2, 2)
                  + Math.pow(y1 - y2, 2));
}
```

Aspetti Metodologici

- Per concludere questa unità
  - introduciamo alcuni aspetti metodologici collegati alla programmazione ad oggetti
- In particolare
  - il concetto di incapsulamento
  - i principi per organizzare i componenti ed attribuire loro le responsabilità

Incapsulamento

- Caratteristica dei componenti
  - separazione tra “cosa sa fare” e “come lo fa”
  - basata sui modificatori di visibilità
- Cosa il componente sa fare
  - la sua interfaccia: i servizi che è in grado di fornire all’esterno
- Come lo fa
  - la sua implementazione: il codice di cui è composto

Incapsulamento

- Esempio: gli oggetti di tipo Punto
  - ⇨i metodi pubblici (l'interfaccia)
  - ⇨getAscissa, setAscissa
  - ⇨getOrdinata, setOrdinata
  - ⇨getQuadrante, toString
- L'implementazione di questi metodi
  - ⇨alcuni sono basati su proprietà (ascissa e ordinata), altre no (quadrante)
  - ⇨NOTA: valeva lo stesso per Circonferenza

Incapsulamento

- Ma
  - ⇨all'esterno non è possibile distinguere il primo dal secondo caso
  - ⇨infatti le proprietà sono esclusivamente private e non è possibile ispezionarle
  - ⇨in altri termini: l'implementazione è realmente nascosta all'interno del componente

Incapsulamento

- Di conseguenza
  - è possibile modificare l'implementazione di un componente senza dover modificare i componenti che ne utilizzano l'interfaccia
  - si abbassa il livello di accoppiamento tra i diversi componenti
  - promettente strumento per la riorganizzazione e la manutenzione del codice

Incapsulamento

- Alcuni esempi
  - potrei cambiare l'implementazione della classe Punto aggiungendo la proprietà (privata) quadrante
  - potrei cambiare l'implementazione del sistema di coordinate, passando alle coordinate polari
  - purchè non cambi l'interfaccia, gli altri componenti non sono influenzati (inoltre, il collegamento è dinamico)

Attribuzione delle Responsabilità

- Attribuzione delle responsabilità
  - ⇨ogni componente ha precise responsabilità
  - ⇨la corretta attribuzione delle responsabilità ai componenti è il problema centrale della programmazione a oggetti
- Principi per attribuire le responsabilità
  - ⇨“rivolgersi all’esperto”
  - ⇨massimizzare la coesione
  - ⇨minimizzare l’accoppiamento

Il Principio dell’Esperto

- Un principio fondamentale
  - ⇨ciascun compito deve essere svolto dal componente che ha le informazioni per farlo (“esperto”)
- Esempio: Punto e Segmento
  - ⇨chi deve calcolare il quadrante dei vertici del segmento ?
  - ⇨chi deve calcolare la lungh. del segmento ?

Il Principio dell'Esperto

- Punto
  - conosce le proprie coordinate
  - e fornisce i metodi relativi di accesso ("get") e modifica ("set")
  - sa calcolare il proprio quadrante
  - sa trasformarsi in una stringa opportunamente formattata
  - attribuzione naturale: il punto conosce le informazioni necessarie (proprie coordinate)

Il Principio dell'Esperto

- Segmento
  - conosce i propri vertici (attraverso due riferimenti che rappresentano l'associazione)
  - sa calcolare la propria lunghezza: conosce le informazioni necessarie
- Nota
  - il segmento potrebbe anche calcolare il quadrante dei propri punti, ma non sarebbe un esperto (conoscenza indiretta)

Il Principio dell'Esperto

si tratta di un'attribuzione innaturale di responsabilità a Segmento

```
// Nella classe Segmento potrei scrivere:

public int getQuadrante(Punto punto) {
        int quadrante = 4;
        if (punto.getAscissa() >=0 && punto.getOrdinata() >= 0) {
            quadrante = 1;
        } else if (punto.getAscissa() < 0 && punto.getOrdinata() >= 0){
            quadrante = 2;
        } else if (punto.getAscissa() < 0 && punto.getOrdinata() < 0){
            quadrante = 3;
        }
        return quadrante;
  }
}
```

Massimizzare la Coesione

- Massimizzare la coesione
  - ⇨ciascun componente deve avere compiti ben definiti e strettamente collegati gli uni agli altri
- Due "funzioni applicative" distinte
  - ⇨la classe Principale: gestisce l'interfaccia (schermi) e il flusso di controllo dell'applicazione
  - ⇨le classi Segmento e Punto: gestiscono le operazioni di calcolo ("logica applicativa")

Massimizzare la Coesione

- Compiti di interfaccia e controllo
  - ⇨visualizzare gli schemi
  - ⇨interagire con gli utenti
  - ⇨controllare il flusso di esecuzione dell'applicazione ("qual è la prossima cosa da fare")
- Tipicamente
  - ⇨un componente Principale
  - ⇨eventuali altri componenti secondari

Massimizzare la Coesione

- Compiti di modello (logica applicativa)
  - ⇨compiti di elaborazione e di calcolo
  - ⇨corrispondenti alla reale logica applicativa
  - ⇨indipendenti dall'interazione con l'utente
- Esempi
  - ⇨Calcolatrice
  - ⇨Circonferenza
  - ⇨Segmento e Punto

Massimizzare la Coesione

- Di conseguenza
  - ⇨Segmento e Punto non contengono schermi e non prendono decisioni sul flusso di controllo ("cosa fare dopo")
  - ⇨Principale non effettua calcoli
- Le classi della logica applicativa
  - ⇨sono una descrizione ("modello") del dominio applicativo

Massimizzare la Coesione

- Metafora
  - ⇨la società dei robot è ad alta coesione
  - ⇨società di omini e donnine moderna e organizzata – di carattere industriale
  - ⇨ciascuno ha un compito preciso ed è specializzato nel suo compito
- Viceversa
  - ⇨applicazione a bassa coesione: società primitiva di tipo non specializzato

Minimizzare l'Accoppiamento

- Corrispondentemente
  - ⇨ è importante ridurre al minimo l'accoppiamento tra le classi
- Classi accoppiate
  - ⇨ la classe A chiama i metodi della classe B
  - ⇨ A "dipende" da B
- Livello di accoppiamento
  - ⇨ tanto più alto quanti più sono i metodi di B chiamati da A

Minimizzare l'Accoppiamento

- Un esempio
  - ⇨ Segmento e Punto
  - ⇨ gli oggetti di tipo segmento devono poter collaborare con i punti che ne rappresentano i vertici
- Modi per "collegare" i due componenti
  - ⇨ attraverso l'utilizzo dei metodi set
  - ⇨ attraverso l'uso dei costruttori

Minimizzare l'Accoppiamento

- Il primo modo
  - viene utilizzato nella versione vista dell'applicazione segmenti (segmentia)
- Esempio : in Principale

```
Segmento s = new Segmento();
Punto p1 = new Punto();
Punto p2 = new Punto();
...
s.setPrimoVertice(p1);
s.setSecondoVertice(p2);
```

Minimizzare l'Accoppiamento

- Vantaggio di questo stile
  - evita di scrivere il costruttore di Segmento
- Svantaggi di questo stile
  - accoppia Principale e Punto
  - alla creazione, il segmento non è in uno stato consistente finchè non vengono inizializzati i riferimenti ai due punti
  - omettere le chiamate dei metodi set genera evidentemente un errore logico

Minimizzare l'Accoppiamento

- Una soluzione alternativa
  - la creazione dei punti è responsabilità di Segmento
- Esempio

```
public void Segmento(double x1, double y1,
                     double x2, double y2)  {
   this.primoVertice = new Punto(x1, y1);
   this.secondoVertice = new Punto(x2, y2);
}
```

Minimizzare l'Accoppiamento

- Vantaggio di questo stile
  - disaccoppia Principale da Punto: Principale non utilizza il costruttore di Punto, solo Segmento utilizza il costruttore di Punto
  - inoltre, dopo l'esecuzione del costruttore, il segmento è già in uno stato "consistente"
- Svantaggio di questo stile
  - obbliga a scrivere il costruttore con argomenti (marginale)

Minimizzare l'Accoppiamento

>> segmentib

- Di conseguenza
  - ⇨la seconda soluzione è da considerarsi preferibile perchè riduce l'accoppiamento tra le classi
  - ⇨consente di eliminare i metodi set di Segmento e di Punto e di rendere entrambi i componenti immutabili
  - ⇨richiede la scrittura di meno codice

Minimizzare l'Accoppiamento

- La creazione degli oggetti
  - ⇨una responsabilità frequente e delicata
- Regole per attribuire la responsabilità
  - ⇨è opportuno che il componente A crei oggetti di tipo B se (in ordine di priorità)
  - ⇨A è in associazione con B
  - ⇨A dipende strettamente da B
  - ⇨A è un esperto per la creazione di B

Minimizzare l'Accoppiamento

- Nel caso precedente
  - ⇨ le regole confermano la bontà della seconda scelta
  - ⇨ Principale è un esperto per la creazione di Punto (acquisisce i dati attraverso lo schermo)
  - ⇨ ma Segmento è in associazione con Punto, per cui la prima regola prevale

Riassumendo

- Proprietà dei Componenti
- Segmenti
  - ⇨ Stringhe
  - ⇨ Messaggi Composti
- Aspetti Metodologici
  - ⇨ Incapsulamento
  - ⇨ Massimizzare la Coesione
  - ⇨ Minimizzare l'Accoppiamento

Termini della Licenza

- This work is licensed under the Creative Commons Attribution-ShareAlike License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/1.0/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

- Questo lavoro viene concesso in uso secondo i termini della licenza “Attribution-ShareAlike” di Creative Commons. Per ottenere una copia della licenza, è possibile visitare http://creativecommons.org/licenses/by-sa/1.0/ oppure inviare una lettera all’indirizzo Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.