Corso di Informatica

Modulo T1

C1–Incapsulamento e tecniche OOP

Prerequisiti

- Tecnica elementare della programmazione
- Principi di programmazione OOP
- Metodologie di progettazione software

Introduzione

Gli elementi della **OOP** (*Object Oriented Programming*) introdotti finora, ci hanno portato a considerare questo paradigma di programmazione più astratto rispetto al paradigma procedurale. Approfondiamo la questione chiedendoci:

Perché e come la OOP si concentra sul *cosa* più che sul *come*?
Come la OOP *espande* le possibilità del paradigma procedurale?
Cosa offre la OOP in termini di *protezione dei dati*?
Come e perché il software ad oggetti è *espandibile* e *riusabile*?

Ci proponiamo di dare risposta a questi quesiti.

Effetti dell'incapsulamento

Una classe è vista come una **ADT** (*Abstract Data Type*) accessibile dall'esterno soltanto mediante le **interfacce** previste.

La proprietà dell'incapsulamento porta i seguenti vantaggi:

- **protezione dei dati**
- **astrazione sui dati**
- **astrazione sulle procedure**
- **mascheramento dell'informazione**
- **indipendenza tra gli oggetti**
- **riutilizzo del software**

Effetti dell'incapsulamento

Protezione dei dati

Grazie all'incapsulamento l'utente non può accedere direttamente ai membri privati, ma lo può fare solo attraverso i metodi implementati come **pubblici**.

```
public class Punto
{ private    Intero x;
             Intero y;
  public Punto()    { ….  }
  public Punto (Intero a, Intero b)  { ….. }
  public double Distanza (Punto P) { ….  }
  public void getP() {…}
  public void setP(Intero a, Intero b) {…}
}
```

Ad esempio:

…

Punto P = **new** Punto(2,0);

System.out.println (P.x); // dà errore in compilazione

Effetti dell'incapsulamento

Astrazione sui dati

L'utente usa gli attributi senza necessità di conoscere la loro rappresentazione in memoria.

Attributi

```
public class Punto
{ private    Intero x;
             Intero y;
  public Punto()    { ….  }
  public Punto (Intero a, Intero b)  { ….. }
  public double Distanza (Punto P) { ….  }
  public void getP() {…}
  public void setP(Intero a, Intero b) {…}
}
```

Ad esempio, non ci interessa quanti *byte* occupino in memoria un numero intero o un numero reale (caratteristica dipendente dalla piattaforma hardware o software) o da come questi siano allocati in memoria.

Effetti dell'incapsulamento

Astrazione sulle procedure

I metodi sono visti come scatole nere, il cui codice non è visibile all'esterno.

Sono attivabili soltanto mediante una interfaccia.

Le **interfacce** sono le firme dei metodi (nome, parametri e valore restituito).

Interfacce

```
public class Punto
{ private    Intero x;
             Intero y;
  public Punto()    { ….  }
  public Punto (Intero a, Intero b)  { ….. }
  public double Distanza (Punto P) { ….  }
  public void getP() {…}
  public void setP(Intero a, Intero b) {…}
}
```

Ad esempio, per modificare le coordinate di un punto P, le istruzioni
P.x = 5; P.y = -2;
danno errore. Si dovrebbe scrivere correttamente:
P.setP (5, -2);

Effetti dell'incapsulamento

Mascheramento dell'informazione

L'**occultamento delle informazioni** (*information hiding*) consente al progettista di riscrivere parte del codice senza modificare l'interfaccia e quindi senza cambiamenti per l'utilizzatore finale.

```
public class Punto
{ private    Intero x;
             Intero y;
  public Punto()    { ….  }
  public Punto (Intero a, Intero b)  { ….. }
  public double Distanza (Punto P) { ….  }
  public void getP() {…}
  public void setP(Intero a, Intero b) {…}
}
```

Ad esempio, per scopi di efficienza, il progettista potrebbe modificare l'implementazione di un metodo, senza modificarne l'interfaccia.

Effetti dell'incapsulamento

Indipendenza degli oggetti

Quando si istanzia più volte una medesima classe, si generano oggetti indipendenti, ciascuno con i propri valori degli attributi ossia con il suo stato.

Ad esempio

Punto P1, P2 (3,2);

crea due oggetti P1 e P2 che possono essere inseriti o eliminati da un'applicazione in modo molto semplice, poiché sono delle entità autonome.

```
public class Punto
{ private    Intero x;
             Intero y;
  public Punto()    { ….  }
  public Punto (Intero a, Intero b)  { ….. }
  public double Distanza (Punto P) { ….  }
  public void getP() {…}
  public void setP(Intero a, Intero b) {…}
}
```

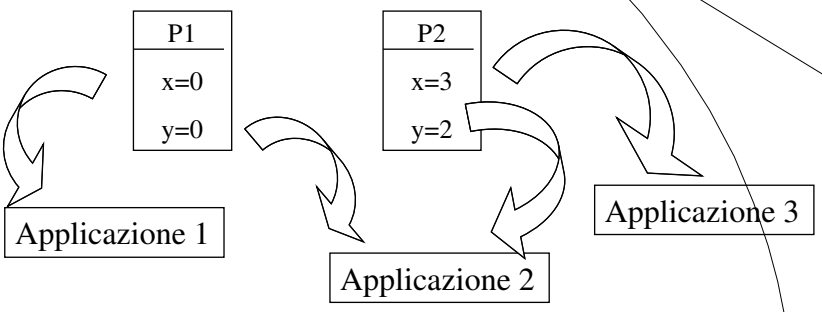
P1
x=0
y=0

P2
x=3
y=2

Effetti dell'incapsulamento

Riutilizzo del software

L'oggetto è visto come una scatola nera di cui si ignora la struttura interna e che può essere riutilizzato inserendolo in altri programmi.



Variabili di classe

Le variabili utilizzate nella programmazione **OOP** possono essere classificate come segue:

- **variabili locali**: sono le variabili definite all'interno dei metodi che hanno un tempo di vita pari al tempo di esecuzione del metodo;
- **variabili istanza**: sono gli attributi di una classe, che, al momento della sua istanza, assumono un ben determinato valore, diverso in generale da oggetto a oggetto della stessa classe;
- **variabili di classe (static)**: sono variabili dichiarate come attributo all'interno della classe e che mantengono lo stesso valore per tutte le istanze di questa classe. Si dichiarano all'interno della classe con la sintassi

static *tipo nomeattributo* = valore*;*

Variabili di classe

Le **variabili di classe** si usano quando si vuole che tutti gli oggetti della classe condividano lo stesso dato.

Ad esempio, per la classe *Automobile* si avrebbe:

```
public class Automobile
{   private Intero Marcia;
            Intero Serbatoio;
    static Intero nRuote=4;
    // metodi
    public void SetSerbatoio(Intero B)
    public void GetSerbatoio( )
    …..
}
```

L'attributo *nRuote* viene dichiarato e inizializzato all'interno della classe, ma si comporta come una variabile globale, visibile anche nei vari metodi.
Ogni oggetto creato dalla classe *Automobile* avrà *nruote=4*.

Metodi di classe

Analogamente alle variabili di classe possiamo avere i **metodi di classe.**

I metodi di classe

- sono caratterizzati dalla parola chiave **static**
- si definiscono come le funzioni normali, con eventuali parametri
- poiché non appartengono alla classe, questi metodi vanno istanziati indicando il nome della classe, anziché il nome dell'oggetto

Ad esempio, per la classe *Punto* il metodo per il calcolo della distanza è:
public double Distanza (Punto P)
ma si potrebbe assieme definire il metodo **statico** corrispondente con
static double Distanza (Punto P1, Punto P2);

Metodi di classe

Ovviamente, cambierebbe il modo di istanziare il metodo:

- il metodo non statico si istanzia con
  Punto p1 = new Punto(2, 4);
  Punto p = new Punto(4);
  System.out.println (p1.Distanza (p));
- il metodo **static** richiede, invece, entrambi i parametri, poiché non prevede alcun oggetto e si istanzia su *p* e *p1* precedenti con:
  System.out.println (Punto.Distanza (p, p1));

I punti tra cui si calcola la distanza, sono usati uno come parametro (*p*), l'altro (*p1*) come oggetto su cui si istanzia il metodo

L'istanza di un **metodo di classe** richiede il *nome della classe* (e non l'oggetto) e la *presenza di due parametri.*

Oggetti come parametri e valori di ritorno

Implicitamente, negli esempi precedenti, abbiamo visto che un oggetto può figurare nei metodi anche come:

- **parametro**
- **valore di ritorno**

In questo modo è possibile per un metodo agire su oggetti passati come parametri e restituire un oggetto come risultato della elaborazione. Ad esempio:

Il valore di ritorno è un oggetto di classe *Punto*

I due oggetti *p1* e *p2* sono passati come parametro al metodo.

public static Punto Pmedio (Punto p1, Punto p2)

restituisce un oggetto di classe *Punto*, che rappresenta il punto medio tra *p1* e *p2*.

Metodologia OOA e OOD

Per realizzare una applicazione **OOP** è necessario svolgere i seguenti passi:

- **Fase 1 - Analisi del problema**, indicata con **OOA** (*Object Oriented Analysis*);
- **Fase 2 – Definizione delle classi**, indicata con **OOD** *(Object Oriented Design)*;
- **Fase 3 – Implementazione**

Metodologia OOA e OOD

Fase 1 - Analisi del problema

- si sviluppa l'analisi del testo;
- si individuano le specifiche;
- si individuano le classi che il problema deve gestire e si assegna loro un nome per referenziarle. E' utile:
  - il concetto che abbiamo di oggetto reale
  - pensare agli oggetti da individuare, senza tenere conto di "cosa fa" il sistema, ma "con cosa" opera;
- si valuta l'opportunità di riuso di classi preesistenti già collaudate

Metodologia OOA e OOD

Fase 2 – Definizione delle classi

In questa fase è utile tenere presente che:

- *una classe non fa qualcosa*, ma *una classe è qualcosa*;
- *occorre disegnare i diagrammi delle classi* nei quali:
  - <u>specificare gli attributi</u>, (di solito espressi da aggettivi);
  - <u>specificare i metodi</u>, definendone i prototipi (di solito espressi da verbi). E' indispensabile fornire almeno i metodi di input e output e costruttori per poter utilizzare gli oggetti;
  - <u>specificare eventuali funzioni complementari</u>, ossia funzioni che usano oggetti come parametro o come valori di ritorno . In generale si tratta di funzioni di libreria che servono a facilitare l'uso degli oggetti;
  - <u>specificare i livelli di visibilità</u> pubblico o privato.

Metodologia OOA e OOD

- può essere utile, a volte, scrivere in pseudocodice gli algoritmi che, attraverso la comunicazione di messaggi tra gli oggetti, realizzano l'applicazione desiderata.

Metodologia OOA e OOD

Fase 3 – Implementazione

Successivamente, si affronta l'implementazione delle classi in un linguaggio **OOL** (*Object Oriented Language*).

Argomenti

- Effetti dell'incapsulamento
  - Protezione dei dati
  - Astrazione sui dati
  - Astrazione sulle procedure
  - Mascheramento dell'informazione
  - Indipendenza degli oggetti
  - Riutilizzo del software
- Variabili di classe
- Metodi di classe
- Oggetti come parametri e valori di ritorno
- Metodologia OOA e OOD

Altre fonti di informazione

- P.Gallo, F.Salerno – Informatica Generale 1, ed. Minerva Italica
- M.Romagnoli, P.Ventura – Linguaggio C/C++, ed. Petrini
- A. Garavaglia, F.Petracchi, S.Forte
  Strutture dati e programmazione per oggetti, ed. Masson Scuola