Dipartimento di Informatica e Sistemistica
Antonio Ruberti

“Sapienza” Università di Roma

Object Oriented Software Design

Corso di Tecniche di Programmazione

Laurea in Ingegneria Informatica

(Canale di Ingegneria delle Reti e dei Sistemi Informatici - Polo di Rieti)

Anno Accademico 2007/2008

Prof. Paolo Romano

Si ringraziano il Prof. G. De Giacomo ed il Prof. Enrico Denti per aver reso disponibile il proprio materiale didattico sul quale si basano queste slides

Principi di base dell'orientazione agli oggetti

Funzioni od Oggetti ?

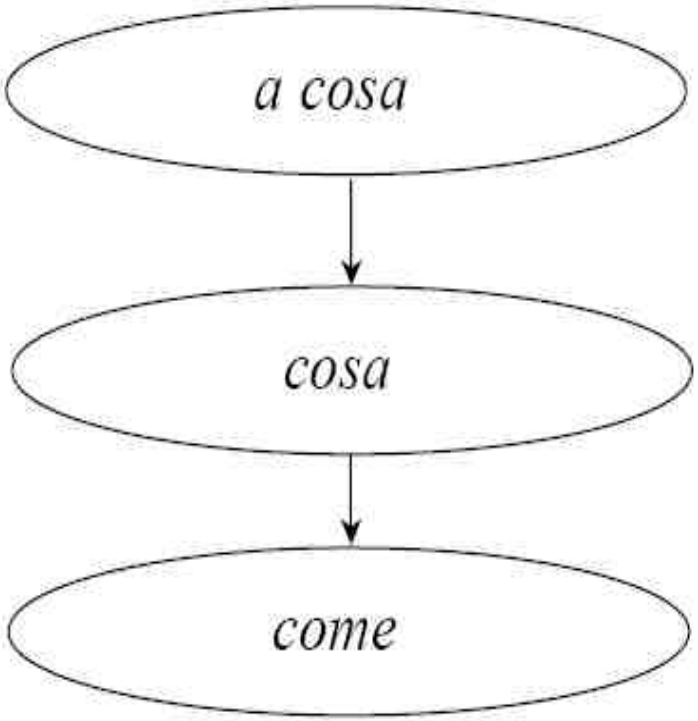
Molte delle tecniche tradizionali sono basate sull'idea di costruire un sistema concentrandosi sulle funzioni

Le tecniche Object-Oriented (OO) rovesciano questo rapporto: un sistema viene costruito partendo dalla classificazione degli oggetti da manipolare

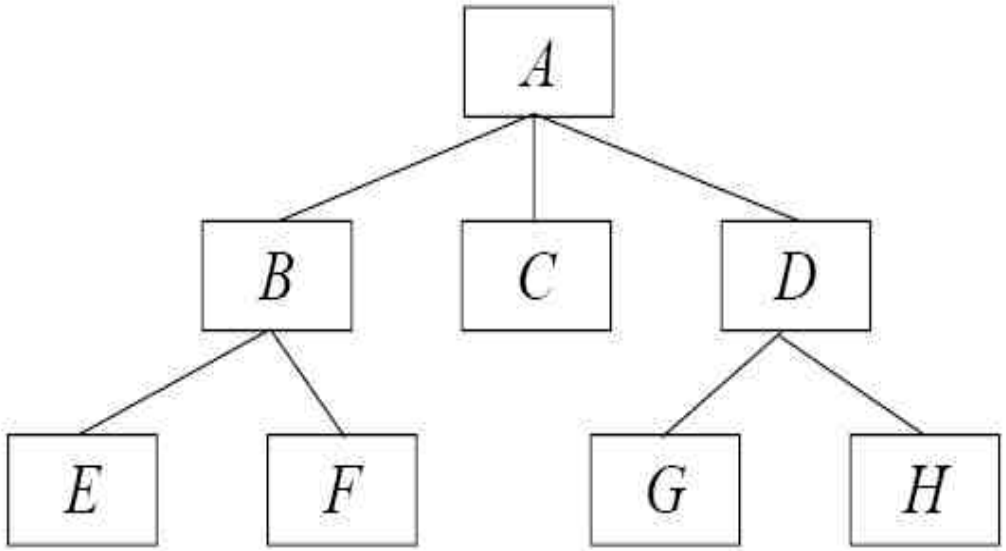
Quindi

La progettazione del SW con l'approccio OO è il metodo che conduce a concentrarsi sugli oggetti che il sistema deve manipolare piuttosto che sulle funzioni che deve realizzare.

*Non chiederti cosa fa il sistema ma **a cosa** serve, di **quali oggetti** è composto, e **come** si opera su di essi!*



Tecnica tradizionale: sviluppo funzionale



- Introduzione di funzioni troppo astratte/innaturali
- Declassamento degli aspetti relativi ai dati
- Attenzione agli aspetti meno stabili (metodi di manipolazione) rispetto a quelli più stabili (informazioni da gestire)
- Creazione di moduli validi solo in un contesto
- Estendibilità e riusabilità (qualità esterne) difficili da raggiungere

Principi di base dell'approccio OO

- Modellare gli aspetti della realtà di interesse nel modo più diretto ed astratto possibile (strutturazione), mediante le nozioni di oggetto, classi e relazioni
- Costruire il programma in termini di moduli, sulla base del principio che ogni classe è un modulo (modularità)
- Legare gli aspetti comportamentali a quelli strutturali
- Proteggere le parti delicate del SW permettendo solo un accesso controllato a dati e funzioni (concetto di interfaccia)

SISTEMI A OGGETTI

- **Un *sistema a oggetti* è composto da (classi e) oggetti che interagiscono fra loro**

Alcune domande:

- **Cosa vuol dire "sistema *composto da* oggetti"?**
  - **Come sono fatti gli oggetti "complessi" ? Incorporano oggetti più semplici?**
  - **Come interagiscono tutti questi oggetti fra loro?**
- **Progettare un *sistema* a oggetti**
  - **o anche solo un *oggetto complesso***

richiede un *modello chiaro* di cosa si vuol fare.

RELAZIONI FRA OGGETTI

Per *modellare un oggetto complesso*

- **e quindi per estensione anche un intero *sistema***

bisogna chiarire bene le *relazioni fra gli oggetti* del dominio

- **le descrizioni parole possono essere fuorvianti..**

Qualche esempio:

- una flotta ***ha*** un ammiraglio
- una flotta ***ha*** delle navi
- un libro ***ha*** delle pagine
- un esagono ***ha*** sei vertici

Il verbo "avere" NON HA LO STESSO SIGNIFICATO nelle varie frasi!

Per modellare bene un oggetto complesso occorre chiarire bene i diversi possibili significati

RELAZIONI FRA OGGETTI: ESEMPI

Ragioniamo su queste frasi:

- una flotta ***ha*** un ammiraglio
- una flotta ***ha*** delle navi
- un esagono ***ha*** sei vertici

- **La flotta non è <u>fatta di</u> ammiragli…**
- **…però, è in relazione con uno (e un solo) ammiraglio che la comanda.**

- **La flotta è invece <u>fatta di</u> navi…**
- **…ma <u>nessuna è indispensabile</u> per l'esistenza della flotta!**
- **Però, la flotta è tale finché (e solo se) c'è almeno una nave.**

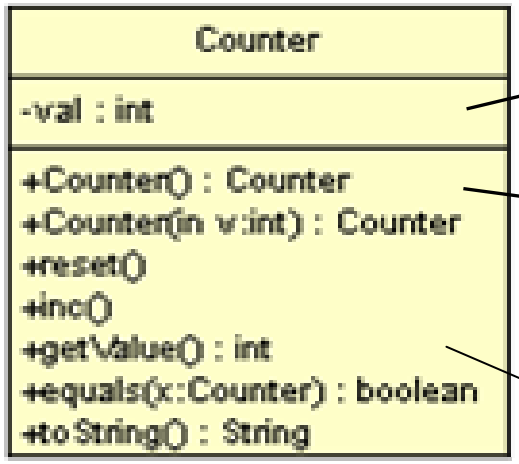
- **L'esagono non è <u>fatto di</u> vertici, ma ne è caratterizzato in modo forte**
- **<u>Tutti i vertici sono essenziali</u>:**
  se manca un vertice o ce n'è uno in più, non è più un esagono!

UML

- ***UML (Unified Modeling Language)*** **è un linguaggio grafico per** ***esprimere il modello di un sistema a oggetti***
  - oggi ampiamente usato in tutte le fasi – analisi, progettazione, implementazione – di un sistema
  - supportato da molti strumenti per analisi, progetto…
  - … ma anche reverse engineering e generazione semiautomatica di codice (Java, C++, C#…)
- **Fra i molti diagrammi, particolare importanza assume il** ***diagramma delle classi***

UML: CLASSI

Una classe è rappresentata in UML mediante un disegno come il seguente:



campi dati:
"-" se privati, "+" se pubblici,
 # se package o protetti

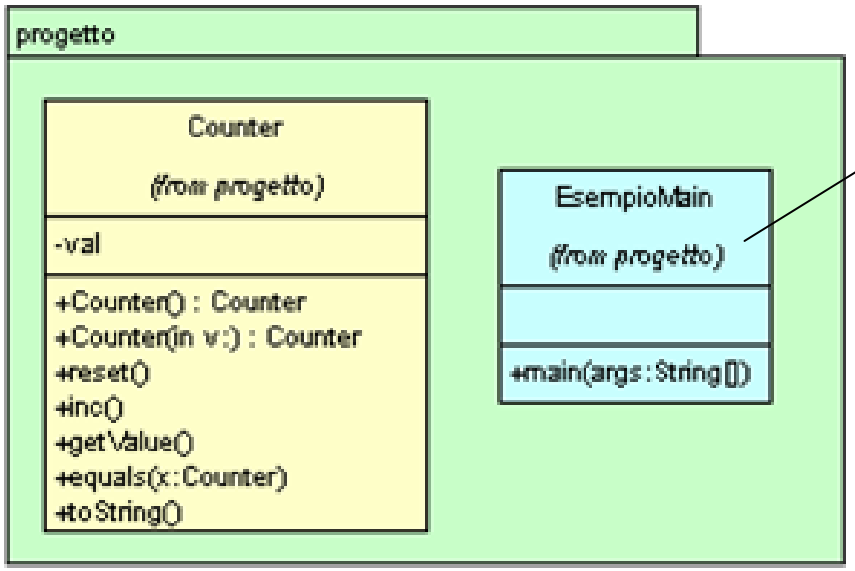
costruttori e metodi:
"-" se privati, "+" se pubblici,
 # se package o protetti

Eventuali dati o metodi statici sono tipicamente sottolineati

Esistono strumenti sia per produrre automaticamente il diagramma a partire dal codice Java (*reverse engineering),* sia per generare lo scheletro del codice dal modello.

UML: PACKAGE

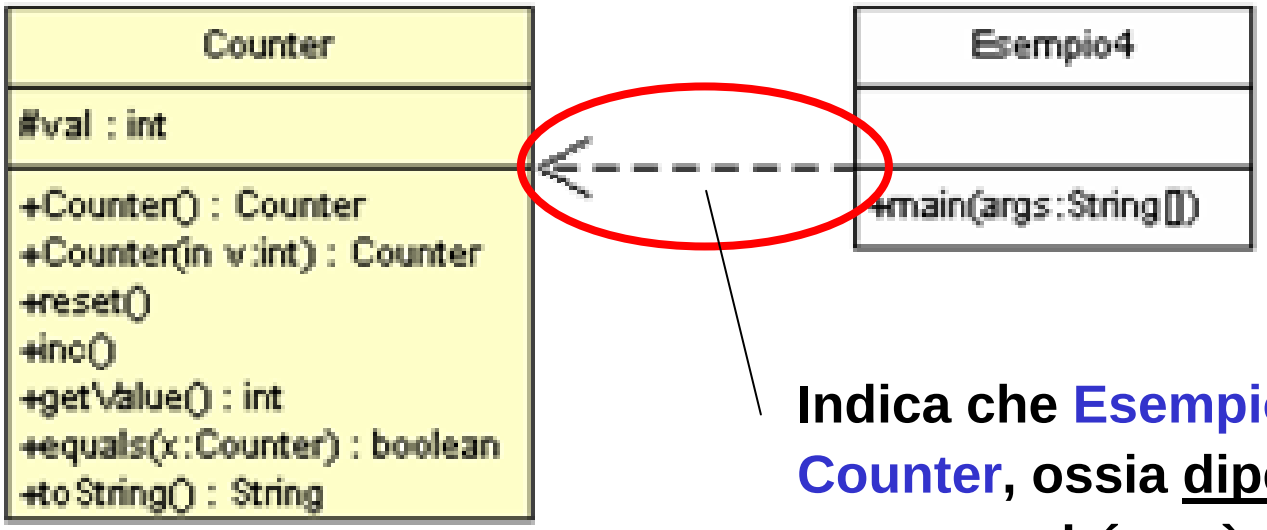
Un package è rappresentato in UML mediante un disegno come il seguente:



Fra parentesi si specifica il package di cui la classe fa parte.

UML: RELAZIONE DI USO

Quando una classe ne usa un'altra, si dice che vi è una *relazione di uso o di dipendenza,* rappresentata da una freccia tratteggiata.



Indica che Esempio4 usa Counter, ossia dipende da esso, perché ne è *cliente.*

UML: ASSOCIAZIONI

Quando una entità rappresentata da una classe “possiede” una o più entità rappresentate da un’altra classe, si parla di *associazione* fra classi.

Esempi di *tipi diversi* di *associazioni*:

- una flotta *ha* un ammiraglio
- una classe *ha* degli studenti
- un libro *ha* delle pagine
- un uomo *ha* un cuore, due polmoni, etc

ASSOCIAZIONI INTERESSANTI

Casi di associazioni di particolare interesse sono le *Aggregazioni* e le *Composizioni.*

- ***Aggregazione***
  - esprime che un oggetto "complesso è un *aggregato* di altri oggetti (le "parti"), nessuna delle quali però è essenziale per l'esistenza dell'oggetto complesso o per il suo funzionamento
  - ESEMPIO: *una flotta ha delle navi*
- ***Composizione***
  - esprime che l'oggetto "complesso" è *composto o caratterizzato* da "parti" tutte essenziali: se ne manca anche una sola, l'oggetto composto cessa di esistere.
  - ESEMPIO: *un triangolo ha tre vertici*

RIFLETTENDO SU ALCUNI ESEMPI…

- **una flotta *ha* un ammiraglio → associazione**
  - **la flotta non è un aggregato di ammiragli**
  - **ma non è neppure composta da ammiragli**
- **una classe *ha* degli studenti → aggregazione**
  - **la classe è fatta di studenti, ma nessuno è essenziale**
  - **si può toglierne uno e la classe continua a esistere**
- **un libro *ha* delle pagine → composizione**
  - **il libro è fatto di pagine, tutte essenziali**
  - **se ne togliamo una, non si può più leggere il libro, che quindi logicamente cessa di esistere.**
- **una linea *ha* dei punti → composizione**
  - **una linea è fatta di punti e sono tutti indispensabili**
  - **se ne manca uno, non c'è più la linea, ci sono due semirette!**

Oggetti composti e delegazione

PROGETTAZIONE INCREMENTALE (1/2)

Spesso capita di aver bisogno di un componente *simile* a uno già esistente, ma *non identico*

- **avevamo Orologio, ma lo volevamo con display**
- **abbiamo il contatore che conta in avanti, ma ne vorremmo uno che contasse anche indietro**

Altre volte, *l'evoluzione dei requisiti* comporta una corrispondente *modifica dei componenti:*

- **necessità di nuovi dati e/o nuovi comportamenti**
- **necessità di modificare il comportamento di metodi già presenti**

Come evitare di dover riprogettare tutto da capo?

PROGETTAZIONE INCREMENTALE (2/2)

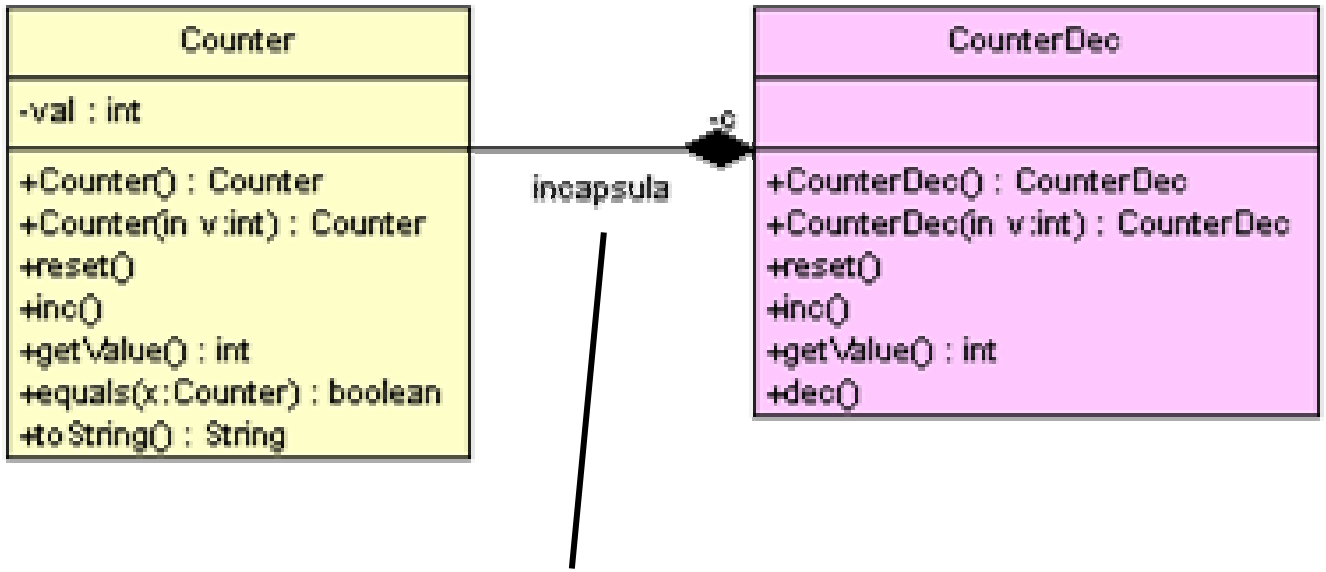
Finora, abbiamo solo due possibilità:

- ***ricopiare manualmente il codice* della classe esistente e *cambiare quel che va cambiato***
- ***creare un oggetto "composto"***
  - **che incapsuli il componente esistente...**
  - **… gli "inoltri" le operazioni già previste…**
  - **… e crei, *sopra di esso*, le nuove operazioni richieste (eventualmente definendo nuovi dati)**
  - ***sempre che ciò sia possibile!***

ESEMPIO

Dal contatore solo avanti…

… *al contatore avanti/indietro.*

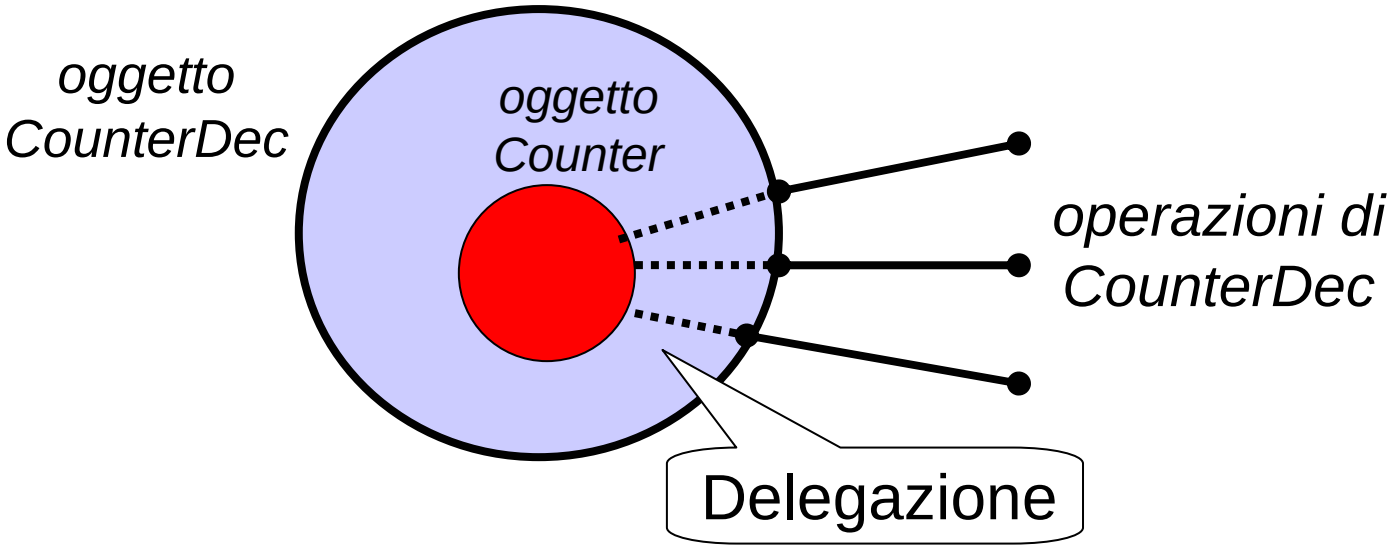


DELEGAZIONE (USO): un **`CounterDec`** è **composto da** un **`Counter`**, sebbene si tratti solo di una scelta implementativa: non si vuole dire che un contatore avanti/indietro sia sempre costituito da un contatore solo avanti !

IL CONTATORE AVANTI/ INDIETRO

Delegazione:

- **ogni oggetto `CounterDec` "ingloba" un oggetto `Counter` al suo interno…**
- **… a cui *delega* le operazioni richieste…**
- **… riformulando quelle che `Counter` non sa fare.**



IMPLEMENTAZIONE

Il *contatore avanti/indietro* *(con decremento)*

```
public class CounterDec {
 private Counter c;
 public CounterDec() { c = new Counter(); }
 public CounterDec(int v){ c = new Counter(v); }
 public void reset() { c.reset(); }
 public void inc()   { c.inc(); }
 public int getValue() { return c.getValue(); }
 public void dec() { ... }
}
```

Delegazione

Come definirlo?

ESEMPIO

Il metodo `dec()` può essere così definito:

- **recuperare il valore attuale V del contatore**
- **riportare il contatore in uno stato iniziale noto (0)**
- **riportarlo, tramite incrementi, al valore V' = V-1**

```
public void dec() {
 int v = c.getValue(); c.reset();
 for (int i=0; i<v-1; i++) c.inc();
}
```

Problema: e se `reset()` non fosse fornita??

ESEMPIO:VARIANTE

Alternativa:

- **sostituire l'attuale Counter c con uno nuovo …**
- **..inizializzato direttamente al nuovo valore desiderato.**

```
public void dec() {
 c = new Counter( c.getValue() – 1 );
}
```

BILANCIO

PRO:

- **Si può riusare un componente esistente per costruirne uno “alle differenze”**

MA

- **se i campi privati non sono accessibili, bisogna *riscrivere anche tutti i metodi che concettualmente rimangono uguali***
- ***inoltre,* non è detto che le operazioni già disponibili consentano di *ottenere qualsiasi nuova funzionalità***

Occorre poter riusare le classi esistenti
in modo più flessibile.

L'OBIETTIVO

- **Poter definire una nuova classe *a partire da una già esistente***
- **Bisognerà dire:**
  - ***quali dati*** **la nuova classe *ha in più* rispetto alla precedente**
  - ***quali metodi*** **la nuova classe *ha in più* rispetto alla precedente**
  - ***quali metodi*** **la nuova classe *modifica* rispetto alla precedente.**