Principi di ingegneria del software paradigmi di programmazione OOP

Principi

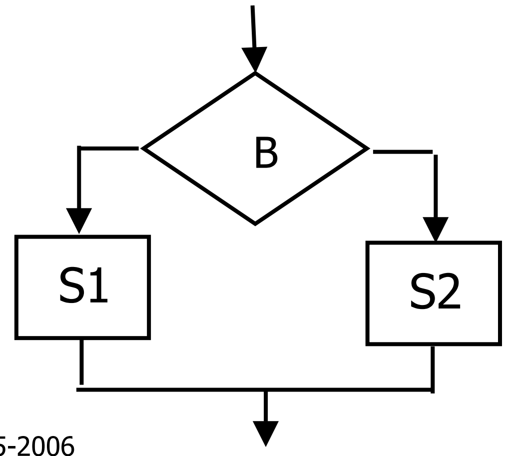
- Astrazione
- Scomposizione dei problemi (*divide et impera*)
- Struttura dei programmi
- Modularità
- Information Hiding (incapsulamento)

Astrazione

- Descrizione semplificata di un sistema che mette in evidenza alcuni degli aspetti e ne sopprime altri
- E’ un processo che la mente umana compie normalmente
- La difficoltà sta nel separare gli aspetti rilevanti da quelli ininfluenti

Astrazione di controllo

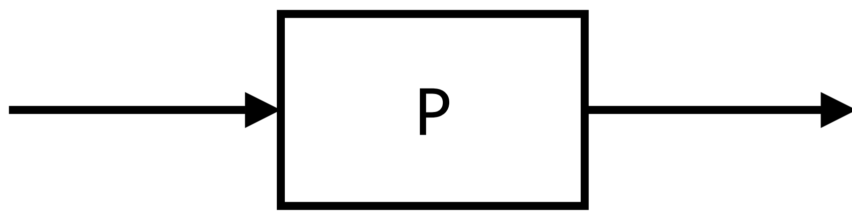
- E' la forma più elementare di astrazione consentita dai linguaggi di alto livello
  - Statement altrimenti indipendenti vengono raccolti in un unico statement
  - IF; IF THEN ELSE; CASE OF; WHILE DO



if B then S1 else S2

Astrazione procedurale

- I linguaggi di programmazione tradizionali (FORTRAN, Basic, Pascal, C) sono procedurali
  - Un programma/sottoprogramma viene visto come la trasformazione degli argomenti di ingresso (X) nei risultati (Y), secondo la procedura (la funzione) P



Astrazione procedurale

- Esempio Y = SIN (X)
  - Uno statement come il precedente introduce un'*astrazione procedurale:*
  - Il simbolo SIN nasconde tutti i dettagli relativi al calcolo del seno di X
  - SIN può essere impiegata come una sorta di “macchina” per la valutazione della funzione per ogni specifico X
  - Il modo in cui viene effettuato il calcolo è irrilevante

Astrazione

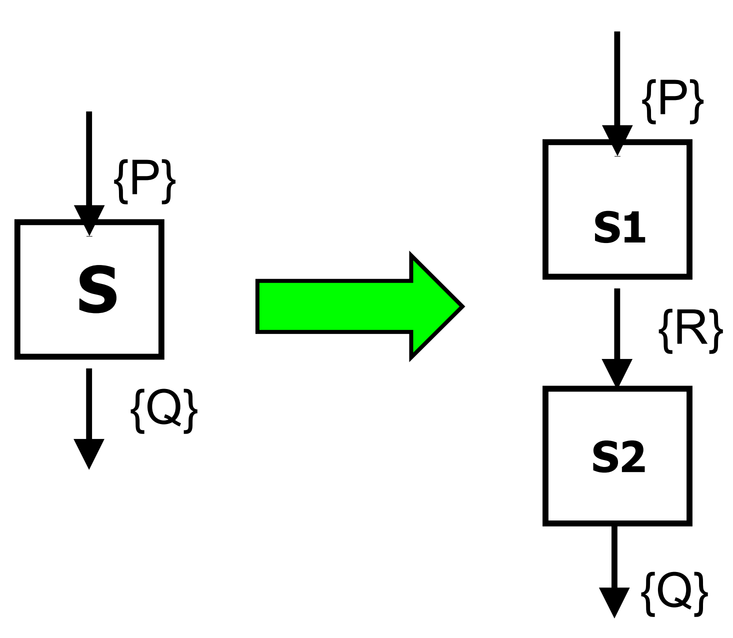
- L'astrazione di controllo e l'astrazione procedurale sono il fondamento della programmazione.
- Entrano comunque nello sviluppo software, qualunque metodologia si usi
- In tutti i linguaggi di programmazione

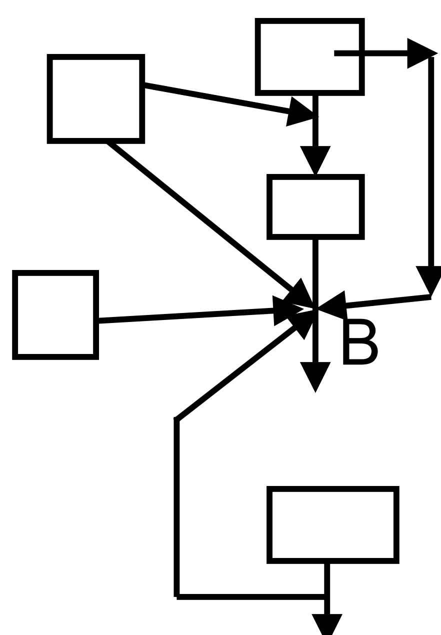
Progettare un algoritmo

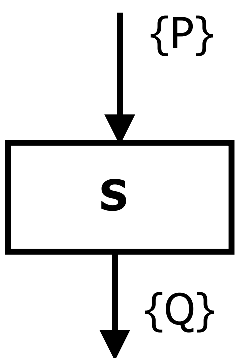
- Metodo: Decomposizione funzionale TOP-DOWN
  - Isolare i problemi/ragionare su programmi parziali
- Applicazione sistematica dell'astrazione procedurale e di controllo
- Riscorso a costrutti con un solo ingresso e una sola uscita (i moderni linguaggi non hanno che questi)

In modo intuitivo

- S viene diviso in S1 e S2; di norma S1 e S2 sono più semplici di S
- Evitare grafi come quello di destra







Formalmente

- P e Q sono due predicati (due condizioni) sui valori delle variabili rispettivamente in ingresso e in uscita allo statement S.
- Vengono dette ASSERZIONI
- Si scrive {P} S {Q}
- La notazione {P} S {Q} significa:

se lo stato delle variabili soddisfa il predicato P e viene eseguito lo statement S, allora, al termine dello statement S, lo stato delle variabili soddisfa il predicato Q

Ricerca anni '70

- Mirava a stabilire assiomi e regole per dare prova rigorosa delle proprietà dei programmi
  - verificare formalmente i programmi (come se si trattasse della dimostrazione di un teorema)
  - sintetizzare i programmi
- Bella costruzione teorica di scarsa utilità pratica
- Di pratico ha prodotto la *programmazione strutturata*

Programmazione strutturata

- Prima vera *tecnologia* per la costruzione di programmi (anni ‘70)
  - isolare i problemi (*divide et impera*)/ragionare su programmi parziali
  - dare struttura ai programmi
- Concetti recepiti dai linguaggi di programmazione

- I salti (GO TO) sono stati banditi dai linguaggi di programmazione correnti

Programmazione difensiva

```
<type> P(p1,p2,…,pn){
if (PRE(p1,p2,…)==false)
return(error);
...corpo della procedura;

if (POST(., ...,) == false) {
          ..cura il malanno;
  }
}
```

Programmazione difensiva

- Si assicura la congruenza degli ingressi rispetto alla specifica della procedura
- Si effettuano le azioni per forzare la postcondizione oppure si prende un provvedimento di altro genere ma che eviti la prosecuzione come se niente fosse

- Concetti recepiti nei moderni linguaggi di programmazione EXCEPTION

Limiti della programmazione strutturata

- Va bene per la progettazione di un algoritmo a misura di singolo programmatore
- Non va bene per la progettazione di sistemi
  - non realizza il concetto di *componente software*
  - riusabilità, modificabilità, manutenibilità basse

- Motivo: dati e strutture di controllo che agiscono su di essi sono sostanzialmente separati

Esempio

- Lavoro diviso tra due programmatori:
  - Tizio: progetta la parte del sistema che comprende la definizione della struttura Data
  - Caio: progetta una parte che usa Data

Tizio

```
 struct Data {
      int giorno;
      int mese;
      int anno;}

Data d;

if (d.mese==9)
 printf("Sett");
```

Caio

```
Data d;

 if (d.mese==9)
   qualchecosa();
```

…..Esempio

- Tizio decide di cambiare la definizione di Data

Tizio

```
  struct Data {
      int giorno;
      char mese[4];
      int anno;}

Data d;

 printf("%s",d.mese);
```

Per Caio è un **disastro!!!**

- Il suo codice non funziona più
- Va cambiato:

```
Data d;
if (strcmp(d.mese=="Sett"))
   qualchecosa();
```

Cosa accade

P1

D

P1

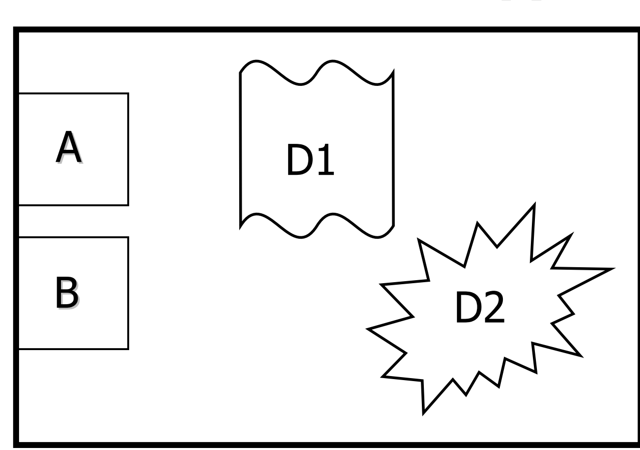
A D hanno accesso sia P1 sia P2, in modo incontrollato

Bisogna disciplinare l’accesso alla struttura dati

Che fare?

- Nascondere i dati entro i moduli di programma, rendendoli invisibili all'esterno
- imporre che tutte le interazioni avvengano solo attraverso le procedure visibili dall'esterno: queste procedure definiscono l'**interfaccia al modulo**
  - Operazioni di lettura
  - Operazioni di scrittura/modifica

Information hiding



- I dati (le strutture dati) all'interno di un modulo costituiscono le *variabili di stato* del modulo stesso
- I dati all'interno del modulo si leggono/modificano solo attraverso l'esecuzione delle procedure che fanno parte dell'interfaccia del modulo

Un modulo (stile ADA)

Module M is A, B;
procedure A is …
procedure B is …

Coesione e accoppiamento

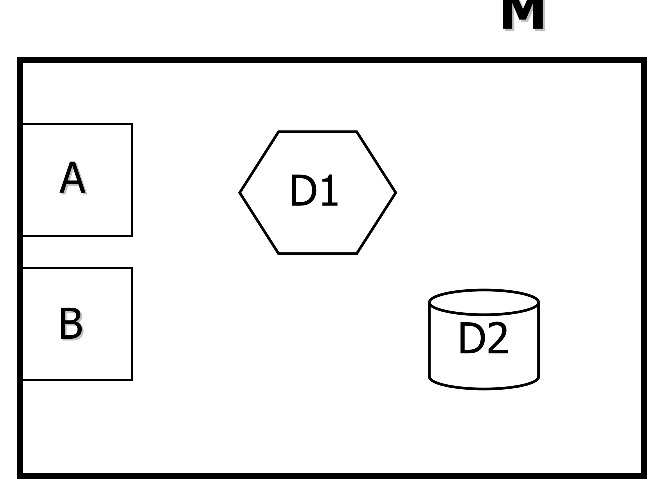
M

A

D1

B

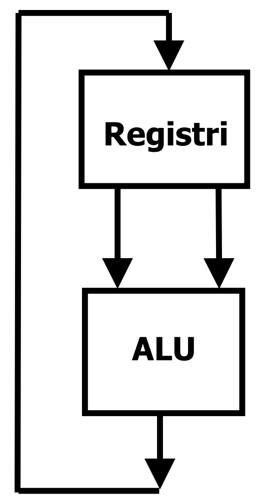
D2

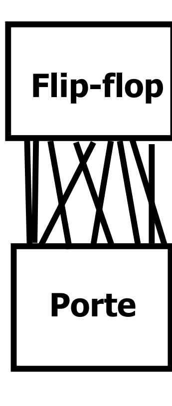


Buoni e cattivi

- Di per sé i moduli non sono né buoni né cattivi

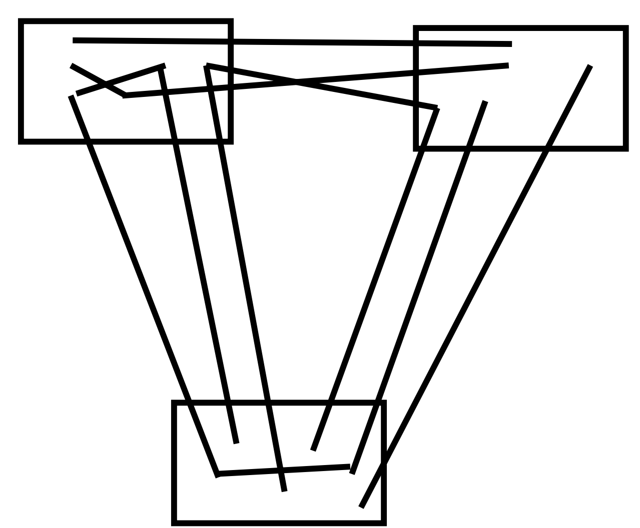
- La soluzione a destra
    - difficile da capire
    - difficile localizzare gli errori
    - difficile da modificare/estendere
    - difficile da riusare
    - costosa da manutenere

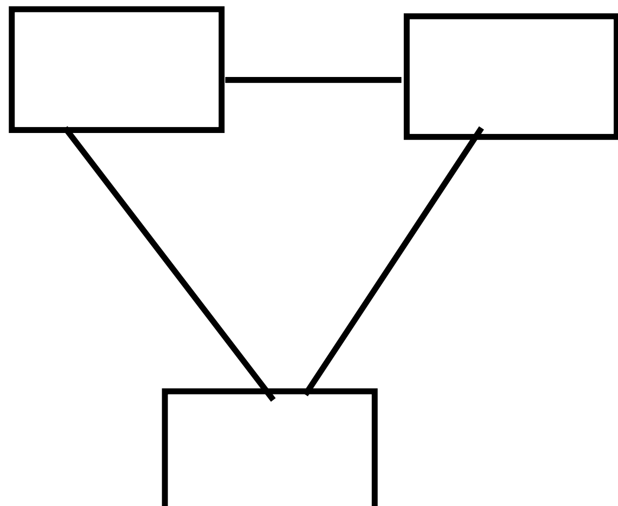




Accoppiamento

- L'accoppiamento tra moduli di programma dovrebbe essere **minimo:**
  - Nessuna assunzione da parte del chiamante





Accoppiamento

5 livelli di accoppiamento

- **Minimo**: i moduli si passano solo parametri scalari o strutture di cui vengono usati tutte le componenti
- **Massimo**: un modulo può modificare direttamente il contenuto di un altro
  - Esempio: il modulo A modifica le istruzioni contenute nel modulo B

Coesione

- I moduli dovrebbero essere altamente coesivi
  - all'interno dello stesso modulo componenti che sono strettamente legate l'una all'altra
  - in modo che un modulo costituisca una unità di programma ben identificata
- 7 livelli di coesione
  - Funzionale (MAX) … Incidentale (MIN)

Interfaccia

- Le procedure visibili dall'esterno costituiscono l'interfaccia
  1) operazioni di lettura delle variabili di stato del modulo
  2) operazioni di modifica dello stato del modulo

Principio fondamentale:

Programmare verso l'interfaccia non verso l'implementazione

Linguaggi convenzionali

- Il linguaggi convenzionali (FORTRAN, C, Pascal) non hanno costrutti di modularizzazione
  - Confinare i dati in “moduli di compilazione indipendenti”
  - Rendere visibili solo gli identificatori che danno il nome alle procedure che si vogliono rendere chiamabili dall’esterno

Esempio classico: uno stack

Un modulo che rappresenta uno stack può essere definito atraverso un numero limitato di procedure:

push

pop

init

is_empty

Il modulo stack

- Il modo in cui è definito è irrilevante (vettore, lista, …)
  - quel che conta è che il modulo risponda alle specifiche di interfaccia

Interfaccia

Specifica come il modulo si interfaccia

Corpo

Implementa la specifica

Stack in C

```
namespace Stack{
  const int dimensione_max=100;
  char v[dimensione_max];
  int top=0;

void push(char c) {...}

char pop()        {...}
}
```

Stack in C (compilazione separata)

File Stack.h: definisce l'interfaccia

```
namespace Stack{        //interfaccia
   void push(char);
   char pop();       }
```

File Stack.c

```
#include "stack.h"                 //include l'interfaccia
namespace Stack{
   const int max_dim=100;
   char v[max_dim];
   int top=0;
}
void Stack::push(char c) {..fa il push}
char Stack::pop()        {..fa il pop}
```

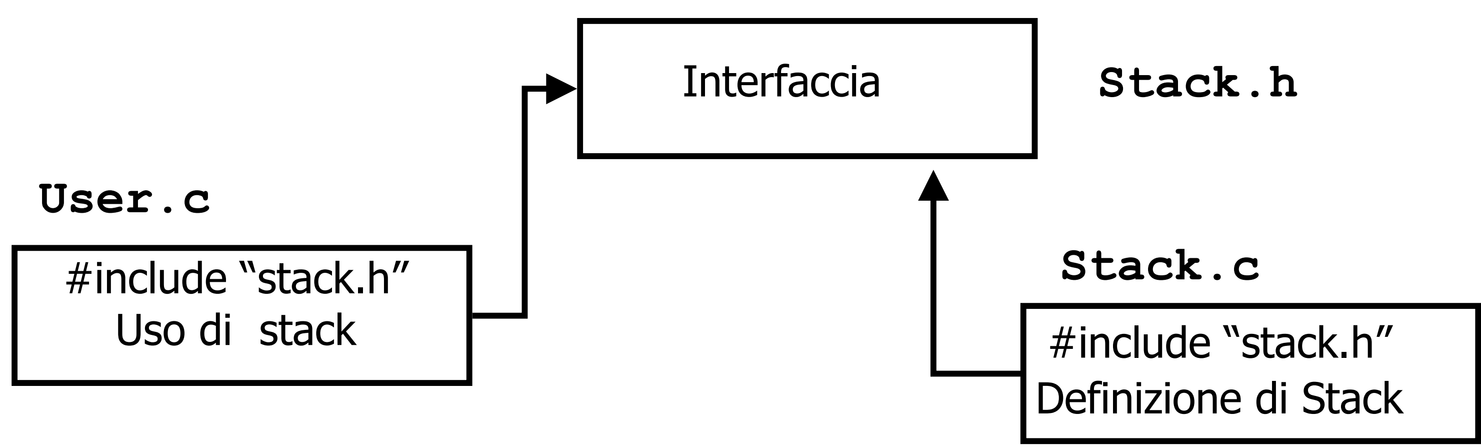
File user.c

```
include "stack.h"                  //prende l'interfaccia
void main()

   Stack::push('c');
    if (Stack::pop() != 'c') printf("impossibile");
```

Compilazione separata

- `Stack.c e User.c` condividono le informazioni presentate dall'interfaccia `Stack.h`
  - per il resto sono indipendenti e compilati separatamente.



Verso i dati astratti

- Un modulo può essere riguardato come il *manager* dei dati che esso racchiude al suo interno
- oppure come una sorta di *dato astratto,* cioè un dato sul quale sono definite le operazioni costituire dalle operazioni di interfaccia

Verso i dati astratti

- Il precedente stack è un dato astratto in versione unica
- Supponiamo che servano due stack:
  - duplicare tutto
  - Prevedere due strutture all’interno (usando il loro nome nelle chiamate)
  - Consentire la creazione (dinamica) delle strutture al tempo di esecuzione:

Lavoro individuale

- Da svolgere in C
  - Uno stack di numeri interi secondo modalità diverse (vettore, lista, ..)
  - Uno stack di numeri complessi
  - Più di uno stack di interi nello stesso programma

Il concetto di dato astratto

- Il linguaggio di programmazione deve permettere:
  - di definire un **nuovo** *tipo di dato*, cioè un dato *non primitivo* come gli interi, reali, etc.,
  - di definire variabili di quel tipo
- Il compilatore deve garantire il corretto accesso al nuovo tipo di dati (type checking)

- E' passato il termine di Abstract Data Type (ADT), anche se sarebbe meglio parlare di User Defined Type

ADT (Abstract Data Type)

Esempio ADA

- Viene definito il tipo di dato astratto **STACK**:

```
STACK is push, pop, init, is_empty;
```

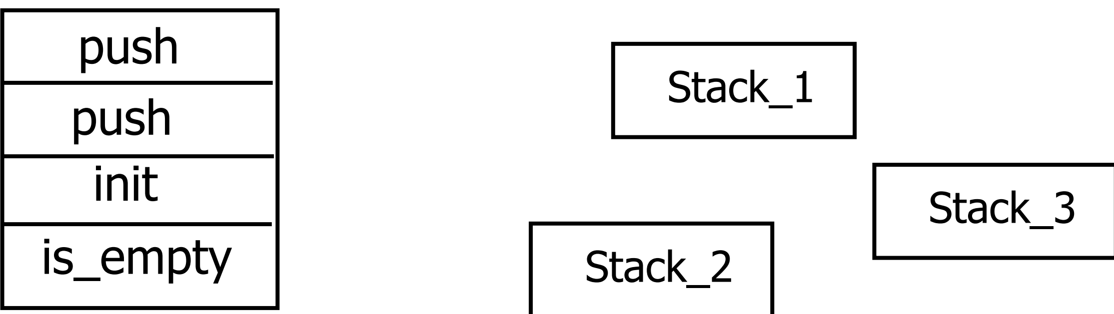
- Vengono dichiarate due variabili di tipo **STACK**

```
   Stack_1, Stack_2:  STACK;
```

- Si usano gli stack nominandoli nel testo di programma

```
  Stack_1.init; Stack_2.init;
  Stack_2.push(X);
```

ADT a run-time



- L'incapsulamento ora è di tipo concettuale (fisicamente i dati sono stati separati dalle procedure)
- Attuato attraverso il controllo dei tipi del compilatore che consente il riferimento a dati solo attraverso le procedure

- Dagli ADT agli oggetti il passo è breve

Programmazione con ADT

- Incapsulamento dati (data hiding)
- Controllo della congruenza tra i tipi
- Possibilità di creare istanze di un ADT
- I tipi di dati astratti sono diversi dai tipi primitivi:
  - Ogni *type-manager* deve definire un meccanismo per la creazione del tipo corrispondente

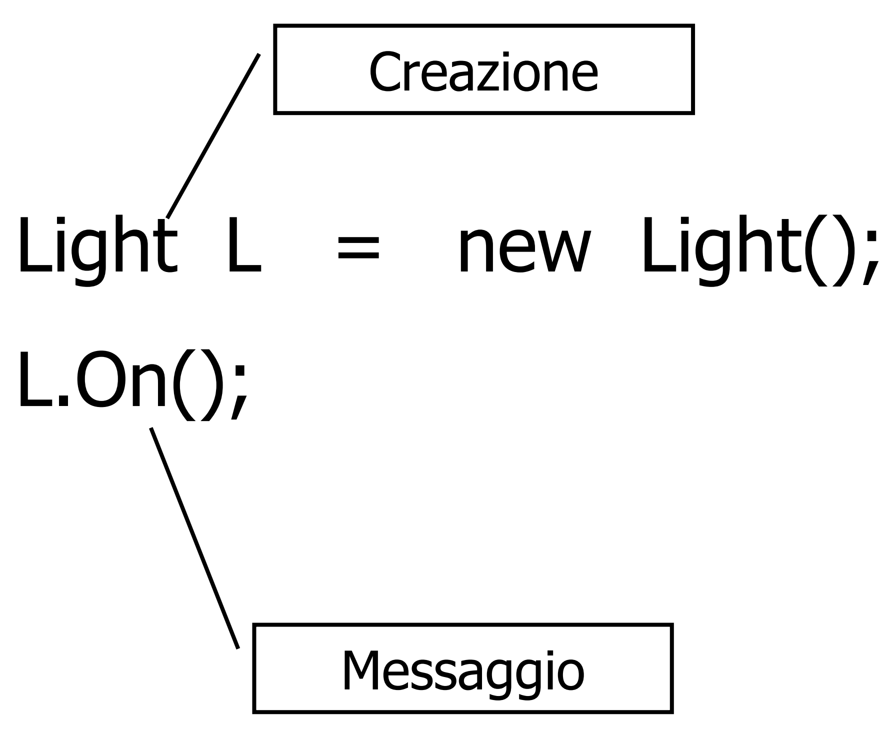
OOP

- Rispetto agli ADT il paradigma OO aggiunge:
  - Ereditarietà
  - Specializzazione
  - Associazione, aggregazione di oggetti

- Spesso si usa l'equazione:

OOP = ADT + Ereditarietà

Programmare a oggetti



Light
On() **Off()**

Paradigma OO

- Il sistema software è basato sul concetto di **oggetto**
- Le entità nel programma sono oggetti
  - Un oggetto è dotato di comportamento
  - Il comportamento è caratterizzato dalle azioni che l'oggetto subisce o che richiede ad altri oggetti

- Modellano oggetti reali, ma anche oggetti non esistenti nella realtà
  - Pulsante: Fisico o concettuale

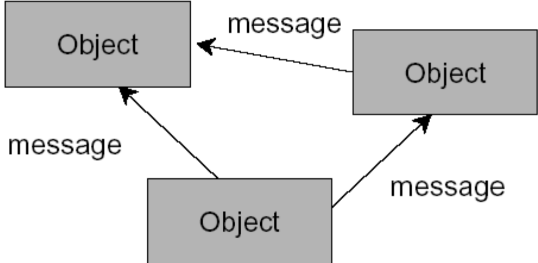
Oggetti

- Un oggetto è fatto di
  - **variabili (attributi)**: definiscono il suo stato
  - **procedure (metodi)**: definiscono il suo comportamento

- **Variabili:**
  - scalari, strutturate o aggregazioni di altri oggetti
  - non visibili dall'esterno
  - lette o modificate solo attraverso le procedure

Oggetti

- **Metodi:**
  - sono le operazioni che possono essere eseguite sul nuovo tipo di dato. L'insieme dei metodi rappresenta l'**interfaccia** (degli oggetti di quella classe verso il resto del sistema)
  - *Method invocation*
  - Essendo un oggetto un’entità autonoma si raffigura invocazioni dei metodi come passaggio di messagi
    - trasmettere e ricevere messaggi anziché passare parametri



5 regole per i linguaggi a oggetti

- Qualunque entità è un oggetto
- Un programma è un insieme di oggetti che si parlano attraverso messaggi
- Ogni oggetto ha una sua memoria fatta di altri oggetti
- Ogni oggetto ha un tipo
- Tutti gli oggetti dello stesso tipo possono ricevere lo stesso messaggio

Oggetti

Notazione UML

IS: Corso
codice_corso= 217 nome_corso=Ingegneria del software
esame()

Collaborazione tra Oggetti

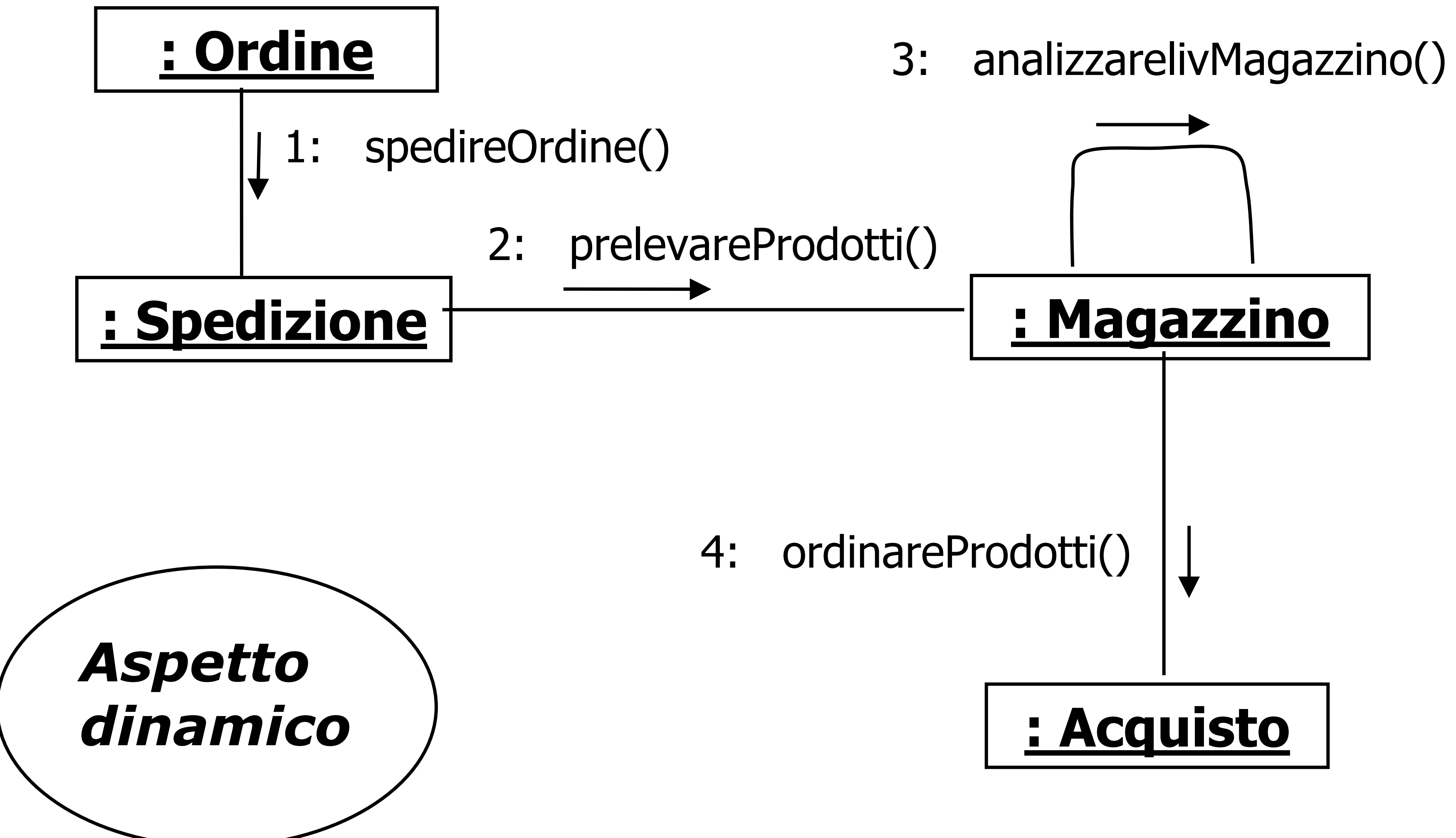
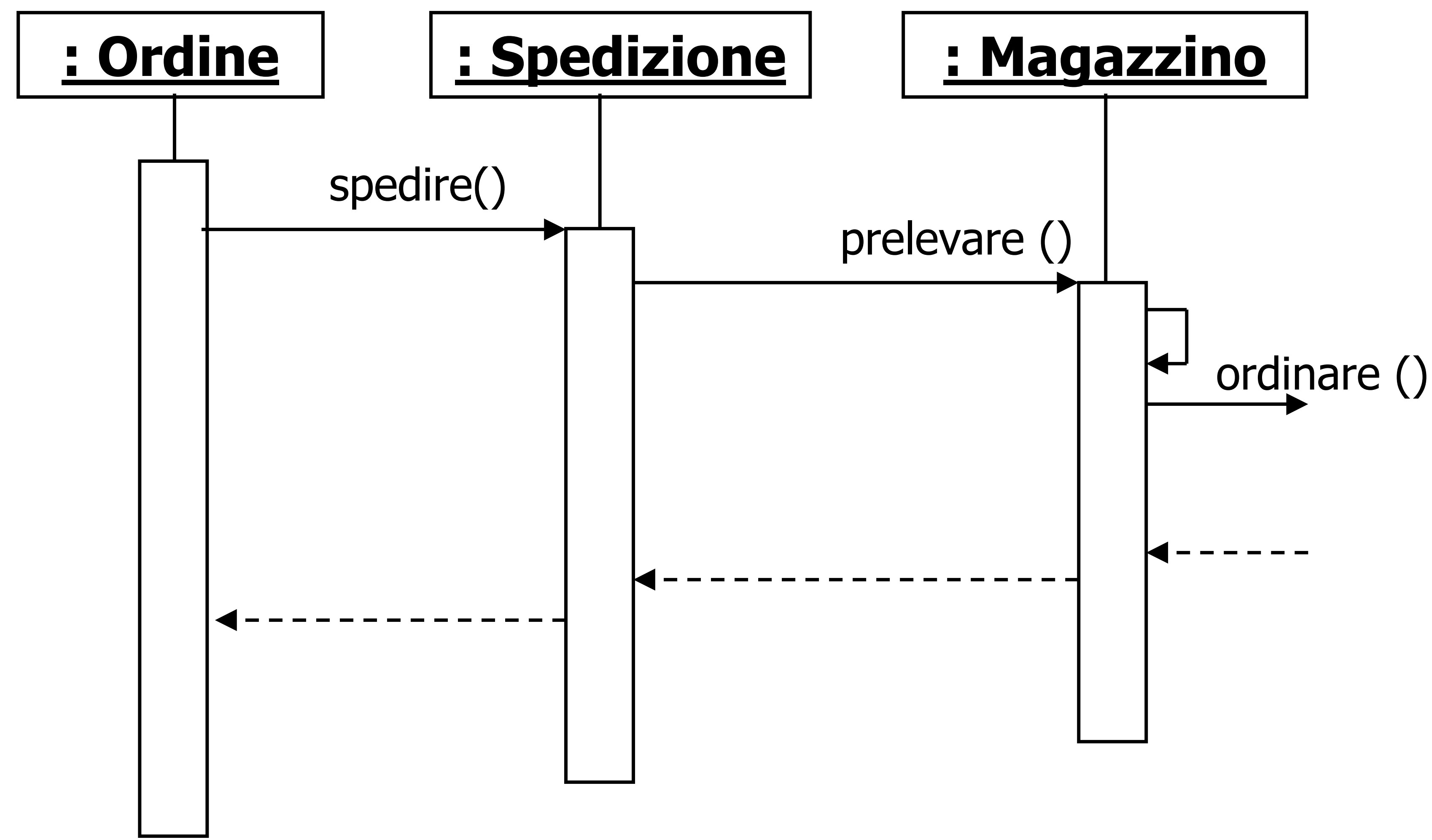


Diagramma di sequenza



Oggetti

- Oggetti dello stesso tipo (stesso comportamento) formano una <u>classe</u>
  - "Giovanni Rossi" appartiene alla classe "Uomo"
    (è un uomo)
  - "C1" appartiene alla classe "Corso"
    (è un corso)

Classe

- Oggetti con lo stesso comportamento appartengono alla stessa classe
- La classe è il costrutto attraverso il quale si definiscono *nuovi tipi di dati*
- Attraverso il meccanismo di istanziazione si creano tante istanze (tanti oggetti) di quella classe quanti ne servono (*Costruttore*)

Esempio

- La classe `Conto_Corrente` rappresenta i conti correnti trattati dal sistema.
  - Ogni istanziazione della classe rappresenta uno specifico conto corrente, con propri attributi (i valori delle sue variabili di stato)
  - Ogni specifico conto corrente è generato e manipolato in accordo al modello rappresentato dalla classe

Rappresentazione di una classe

Nome

Rettangolo

Nome
Attributi
Metodi

Rettangolo
base altezza posizione
sposta() area()

Rappresentazione di una classe

Dettaglio per la fase di implementazione

Rettangolo
- base: int - altezza: int - posizione: Point
+Rettangolo(p1: Point,p2: Point) +sposta(pos: Point) +area(): int

Attributi privati (non visibili dall'esterno)

Metodi pubblici (visibili dall'esterno)

Esempio *C++*

```
Class point
{
private:
     double x,y; //coordinate public:
  point(double xx, double yy){..}

  double get_x(){..}
  double get_y(){..}
  public void move(double xx,
               double yy){..}
}
```

Esempio C++

- Da qualche parte si possono definire uno più punti:

```
point P1,P2,P3;
```

- `P1, P2, P3`: sono tre istanze della classe `point`;

- l'istanziatione è statica

Esempio Java

```
public class Cerchio{
  double x, y; //coord del centro
  double r;            // raggio

  public Cerchio(double x, double y,
  double r) {
     this.x = x; this.y = y;
    this.r =r;
    }
```

Cerchio Java (segue)

```
// metodi che valutano circonferenza e area

public double circonf() {
        return 2 * 3.14 *r
   }
public double area() {
        return 3.14 * r * r}
   }
```

Cerchio Java (segue)

Istanziazione

```
Cerchio c;     //  c è dichiarato come
             //  Cerchio

c = new Cerchio(10.2, 0.0, 5.6);
               // istanziazione
```

Oppure

```
Cerchio c = new Cerchio(10.2, 0.0, 5.6);
```

Cerchio Java (segue)

- La dichiarazione di `c` dà semplicemente il nome "`c`" a un oggetto `Cerchio`
- **NON** istanzia l'oggetto
- In Java tutti gli oggetti devono essere creati **dinamicamente**
  - attraverso il costrutto `new`

L'operatore **new**

- Istanzia dinamicamente un nuovo oggetto della classe specificata
- Alloca la memoria
- Chiama il dovuto costruttore
- Restituisce un riferimento/puntatore (Java/C++) all'oggetto creato
- Con l'istanziazione dinamica si può avere un numero di oggetti variabili.
- Java è dotato di un *Garbage Collector* per recuperare (e riorganizzare) le aree occupate da oggetti non più in uso (orphans)

Esempio di uso (Java)

- Una volta che l'oggetto è stato creato si possono usare i suoi metodi.
  - Si può assegnare alla variabile “a” il valore dell'area del cerchio “c”

```
double a; a = c.area();
Cerchio c2 = new Cerchio(4.,1.,1.);
a= c2.area();
```

- L'attenzione è sull'oggetto non sulla chiamata di funzione. non c'è da passare un parametro. La sintassi dice che si lavora sull'oggetto e che l'area è calcolata per tale oggetto.

Riferimenti e oggetti (JAVA)

```
c  = new Cerchio()
```

- **`c`** è il riferimento all'oggetto **NON** è l'oggetto
- Con la sequenza

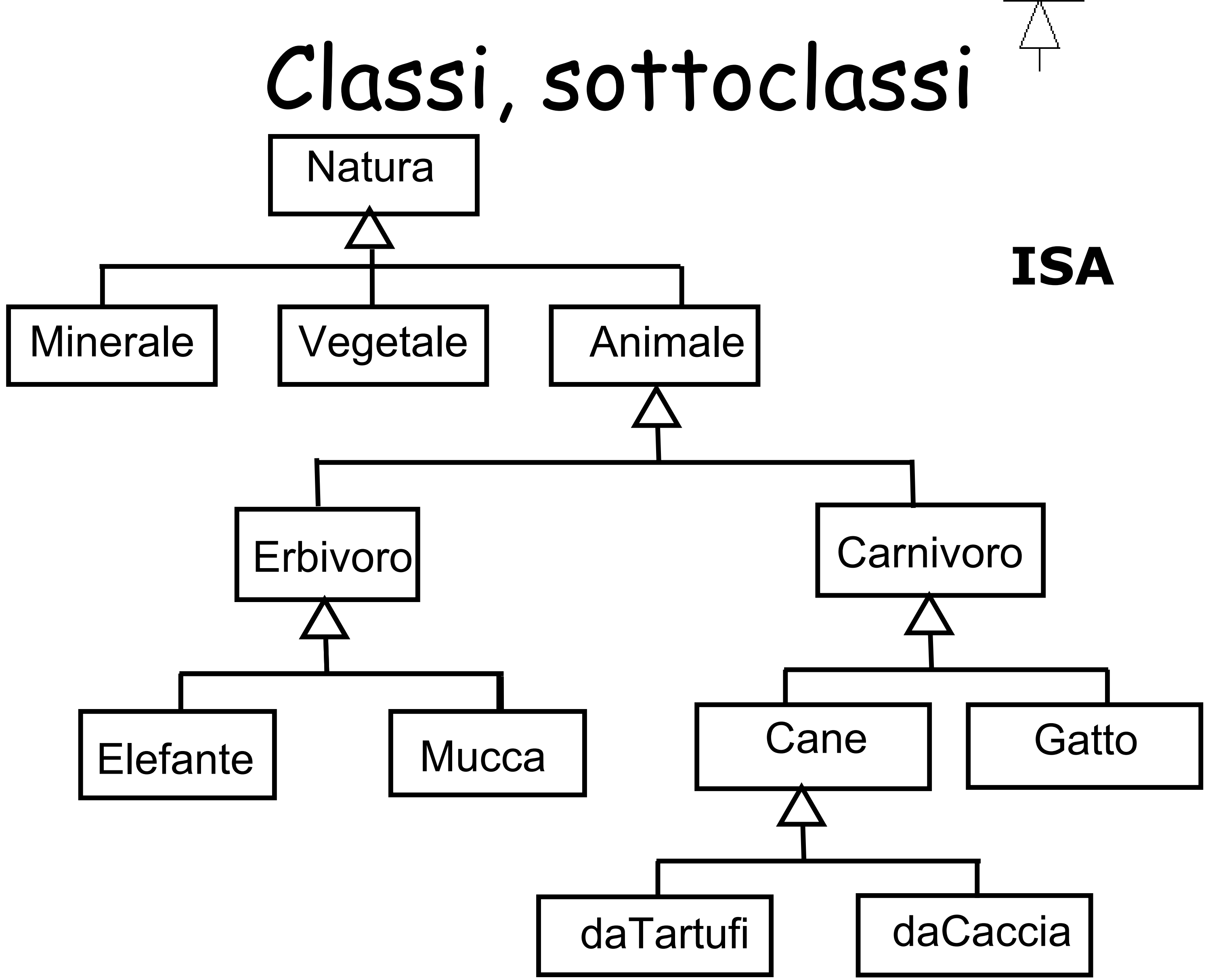
```
Cerchio d = new Cerchio(..);
c = d;      //c  d individuano lo stesso oggetto
```

il precedente oggetto riferito da **`c`** è perso !!

Cosa si è guadagnato?

- Per ora non c'è differenza con il concetto di dato astratto
  - è stato definito un nuovo tipo di dato
  - sono state definite le operazioni su di esso
  - si creano istanze a piacere di elementi della stessa classe

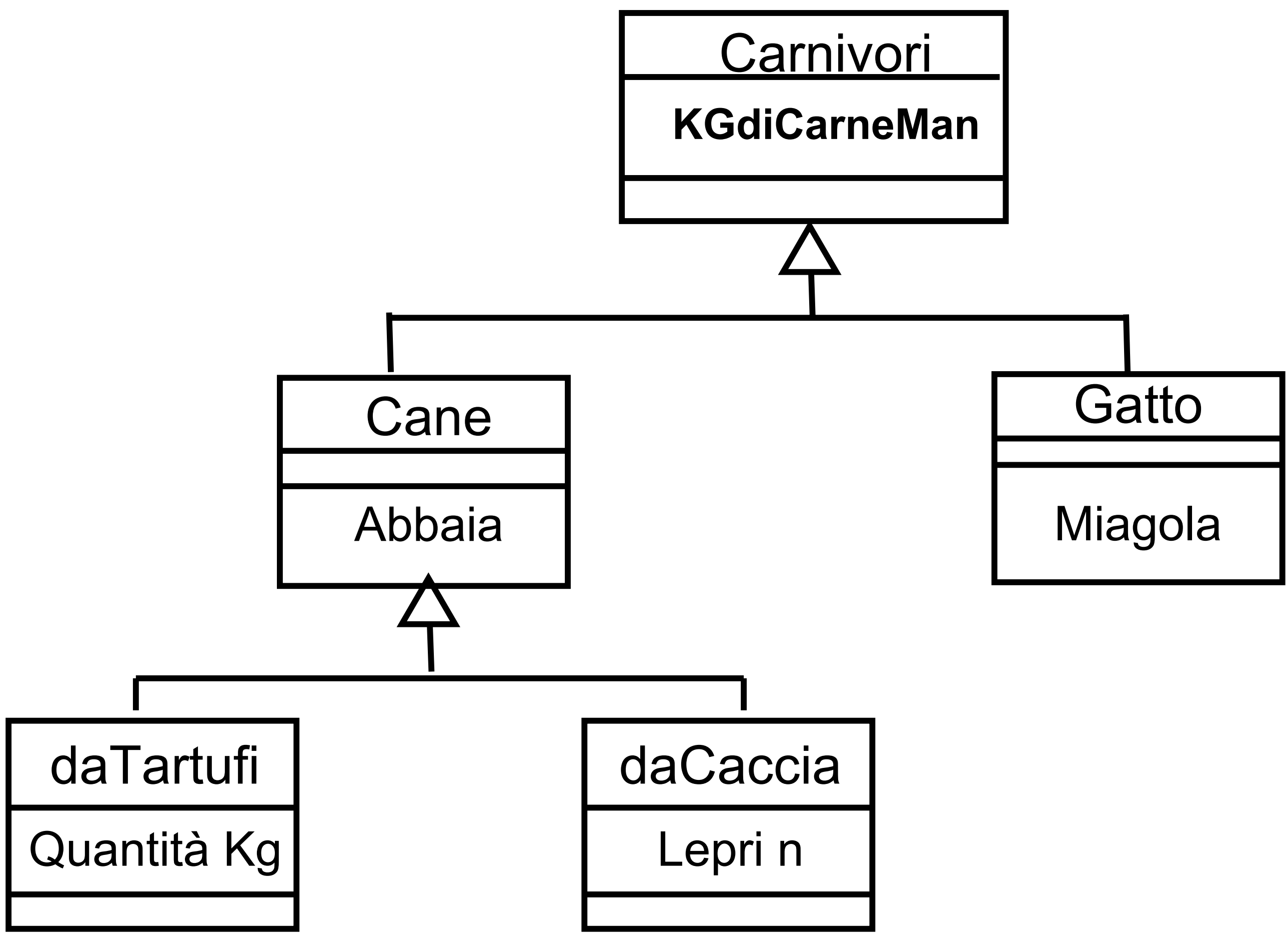
Classi, sottoclassi



Comunalità e specializzazione

- Esempio:
  - Tutti i carnivori mangiano carne e quindi per tutti si potrebbe definire l'attributo KGdiCarneMangiata
  - I gatti miagolano
  - I cani abbaiano
  - I cani da tartufi trovano una Quantità
  - I cani da caccia scoprono Lepri

… Comunalità e specializzazione

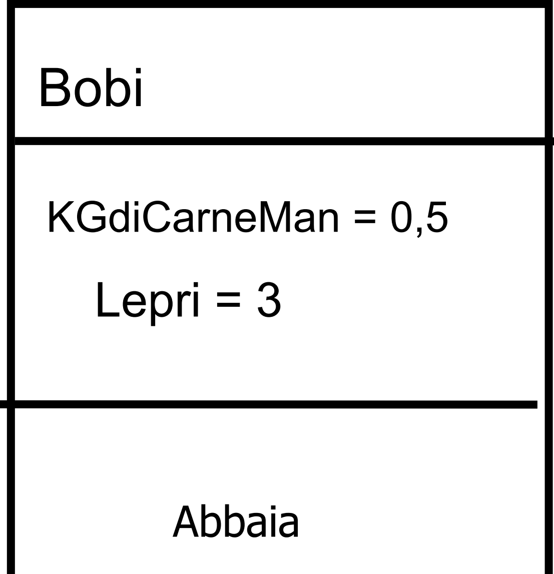


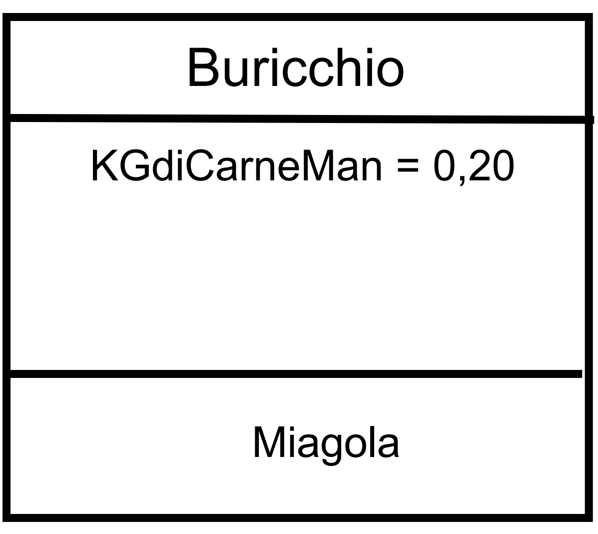
Ereditarietà

- Si dice che le sottoclassi ereditano le proprietà delle superclassi
  - Sia Cane che Gatto ereditano **`KGdiCarneMan`** da Carnivori

- Le sottoclassi "specializzano" le superclassi

Due Oggetti di classi diverse



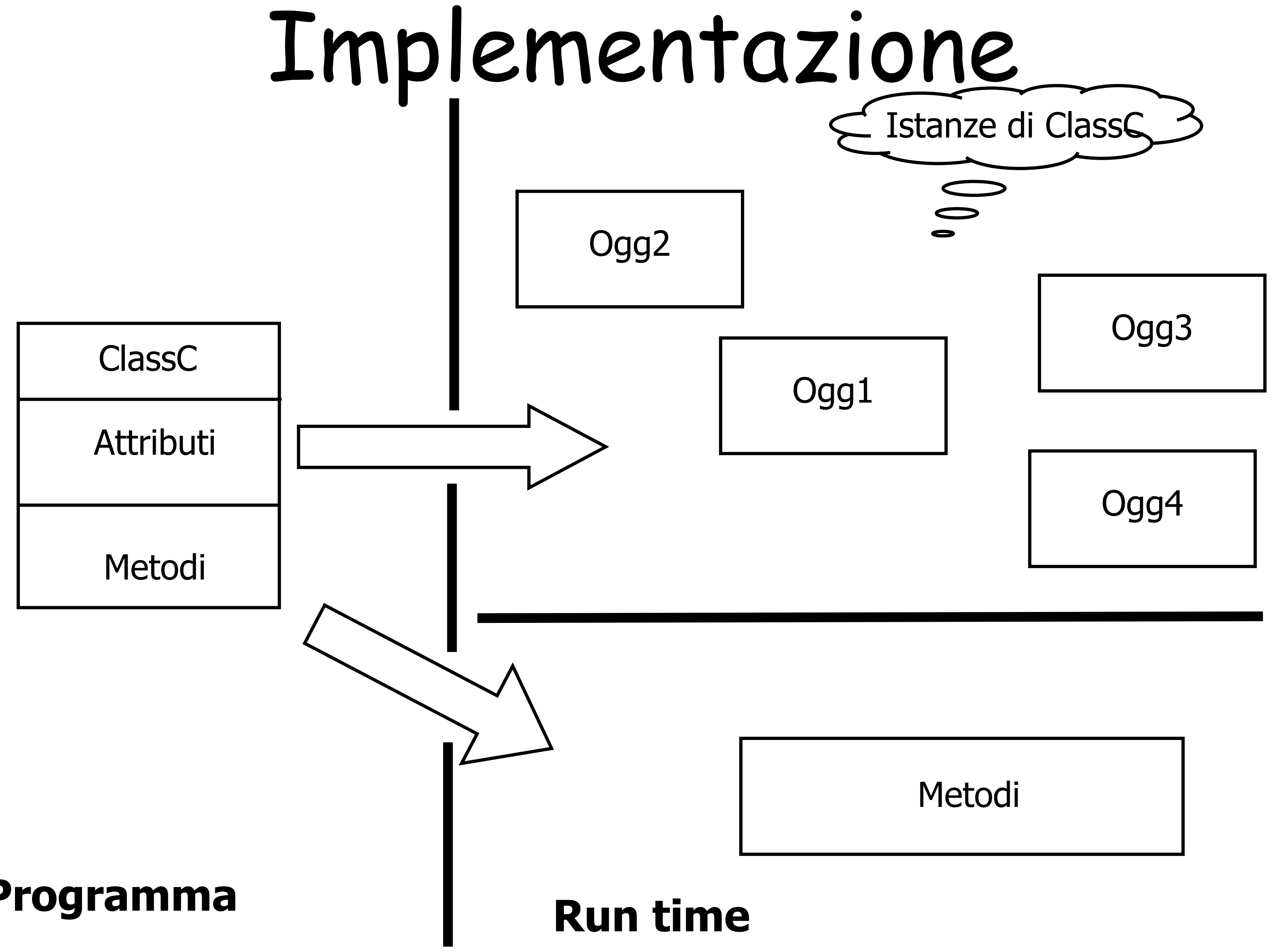


- I sostantivi sono gli attributi, i verbi sono i metodi

Ereditarietà

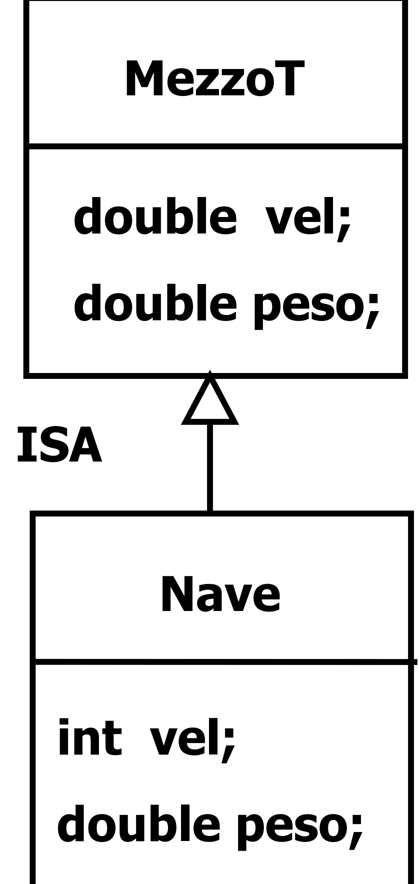
- Le sottoclassi condividono il comportamento delle superclassi ma possono aggiungere degli aspetti nuovi di comportamento o modificare quelli della superclasse
  - **`Abbaiare`** e **`miagolare`** sono aspetti nuovi del comportamento di cani e gatti

Implementazione



Overriding degli attributi

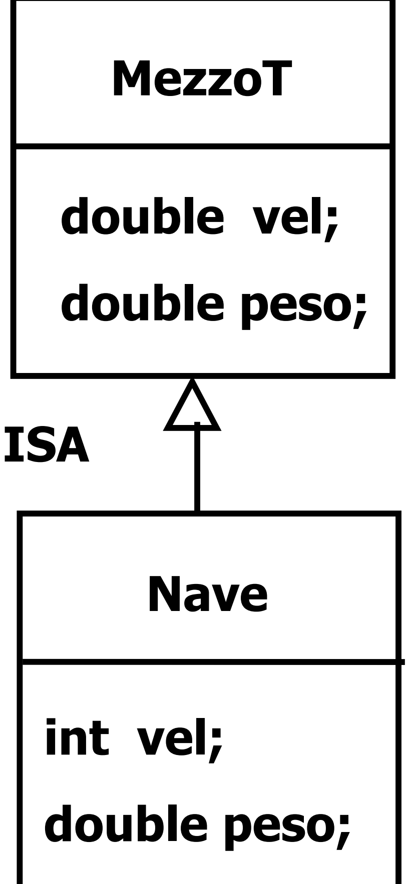
- Un attributo di una superclasse può essere *sovrascritto* da un attributo di una sottoclasse con identico nome
- L’attributo ridefinito può essere o no dello stesso tipo di quello della superclasse
- Il compilatore è in grado di capire a quale classe appartiene vel o peso e di generare i riferimenti giusti



Overriding degli attributi

```
MezzoT NN = new MezzoT();
Nave  Titanic = new Nave();


NN.vel;
Titanic.vel;
```



Overloading dei metodi

- In una stessa classe si possono definire più metodi con identico nome, ma differenziati dai parametri.
- E' il caso tipico dei costruttori.

  Esempio: la classe Cerchio (Java)

Overloading dei metodi

```
public class Cerchio{
  public double x, y;          // centro
  public double r;          // raggio

  public Cerchio();

  public Cerchio(double x,
               double y, double r)
     {
     this.x = x; this.y = y;
     this.r =r;}
```

Overloading dei metodi

```
Cerchio c1= new Cerchio();
Cerchio c2, c3;
c2= new Cerchio();
c3= new Cerchio(10.2,0.0,5.6);
```

- Il compilatore sceglie il giusto costruttore in base alla chiamata

Costruttori

- Se non viene dichiarato un costruttore Java ne provvede uno di *default*
  - Equivale a **`Cerchio()`**
- Per un costruttore come **`cerchio()`** Java inizializza gli attributi (**`zero`** o **`false`**)

… Overloading degli operatori

- Gli operatori comuni (+, *, -, ==, ...) possono essere ridefiniti
  - L'operatore + ha significato di somma ed è definito per i tipi primitivi del linguaggio:

<int> + <int>

<float> + <float>

... Overloading degli operatori

- Può essere utile usare lo stesso simbolo per indicare un’operazione di somma in cui vengono ordinatamente sommati, ad uno ad uno, gli elementi di una struttura dati:

<stru> + <stru>

… Overloading degli operatori

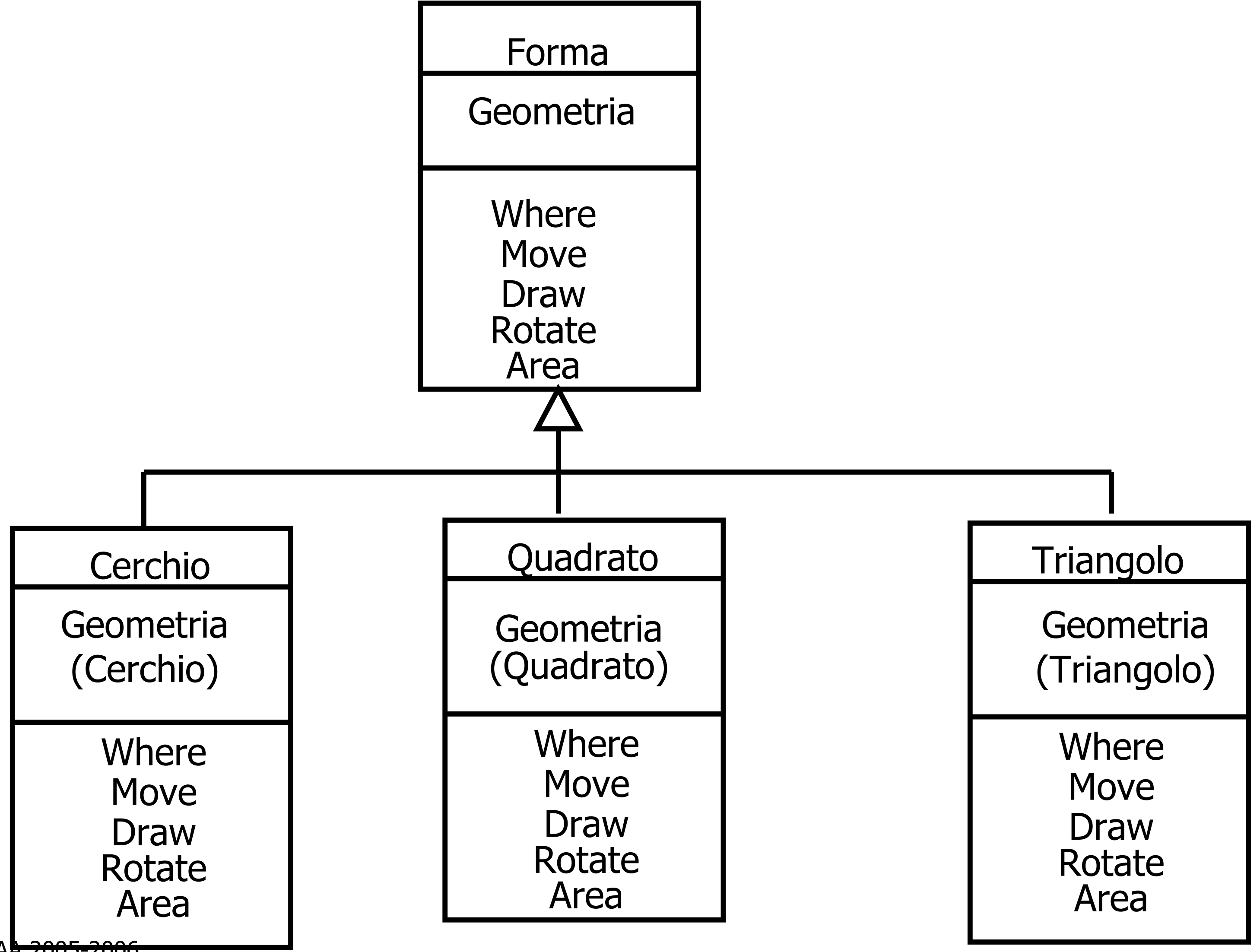
- <stru> è una classe al cui interno è stato ridefinito l’operatore “+”

- E’ il compilatore che decide quale operatore utilizzare basandosi sul contesto in cui il simbolo (+) di operazione appare

Polimorfismo

- Capacità di presentare comportamenti diversi
- Basato sulla presenza di un'interfaccia comune per differenti classi di oggetti

Polimorfismo



Polimorfismo

- Cerchio, Quadrato e Triangolo sono Forme
- Ogni forma ha un “centro”
- Ogni forma ha una sua geometria specifica:
  - raggio
  - lato
  - base, altezza

..Polimorfismo

```
class Point {......}
class Color {......}


abstract class Forma {
      Point      center;    //Geometria
      Color      col;



\\segue
```

..Polimorfismo

```
//segue
```

Interfaccia

```
public Point where ()
        {return center;}
public void  move (Point to)
{center= to; draw();}
public abstract void draw ();
public abstract void rotate(int g);
public abstract double area();
```

..Polimorfismo

- L'interfaccia presenta 5 metodi : **`where`**, **`move`**, **`area`**, **`draw`**, **`rotate`**
- **`area`**, **`draw`** **e** **`rotate`** sono astratti (virtuali)
  - la loro implementazione è rimandata a una classe derivata da **`Forma`**.

..Polimorfismo

```
Public class Cerchio extends Forma{
  double r;  //geom. del cerchio

  public double area() {
          return r*r*3,14;}
  public void draw () {..}
  public void rotate (int g) {}
}
```

..Polimorfismo

```
public class Quadrato extends Forma
{
  double L;

  public  double area(){
          return L*L;}
  public void draw () {..}
  public void rotate (int) {..}
}
```

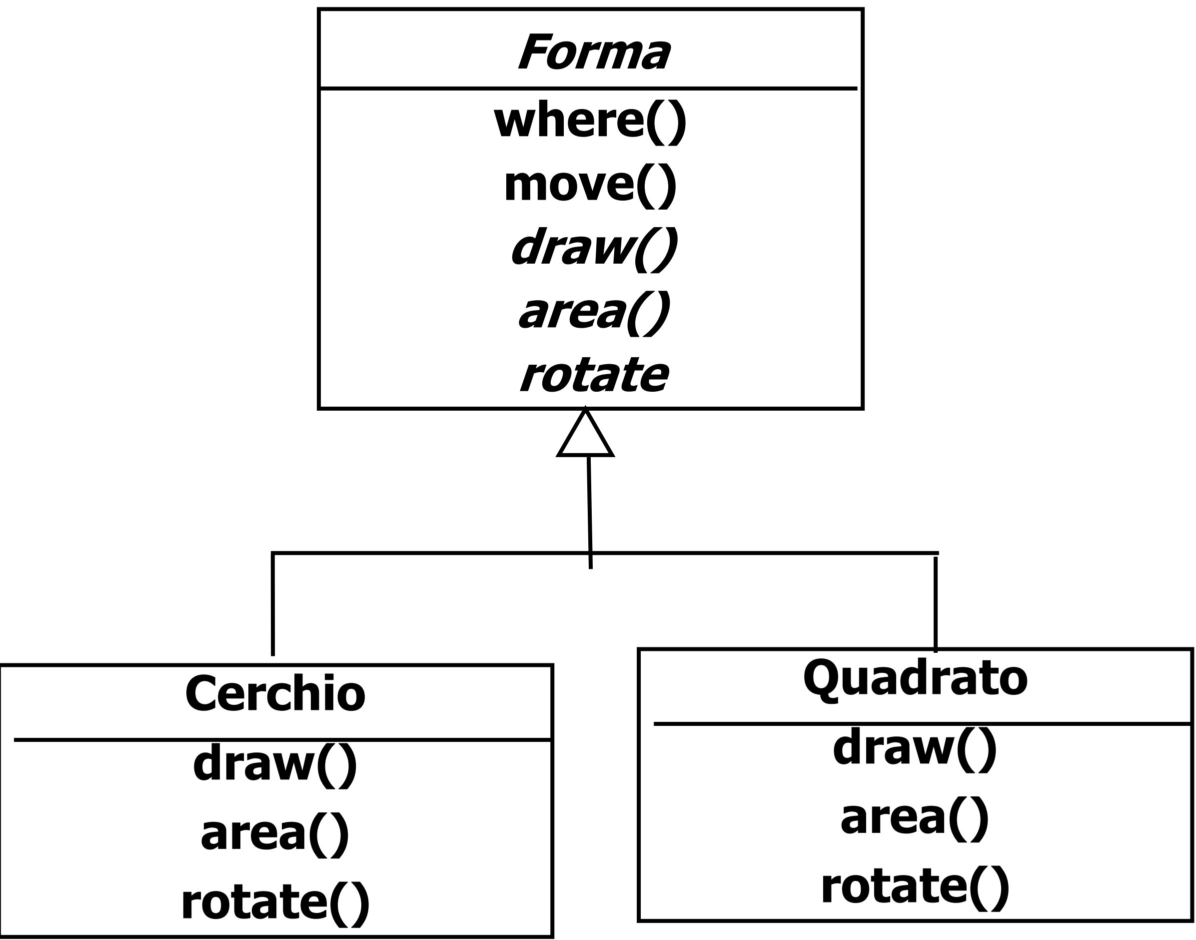
Il programma chiamante:

```
point P1, P2, P3;
Cerchio C1, C2;
Quadrato Q1, Q2;
Triangolo T[100];

  .... Istanziazione qui

  C1.move(P1);
  Q1.move(P2);
  P3= T[10].where();
  Q2.move(P3);
  Q1.rotate(30);
```

La gerarchia



L'interfaccia comune

```
Forme[] ff = new Forme[10];
   ff[0] = C1; ff[1] = Q1;
   ff[2] = new Quadrato(...);
   ff[3] = new Triangolo(..);    ff[4] = ..;

   ff[0].draw();  //disegna C1
   ff[1].draw();  //disegna Q1
   ff[3].draw();  //disegna il triangolo
   int i = 0; double a= 0;
   while (ff[i] != null){
            a= a+ff[i].area(); i++;}
```

..per sommarizzare

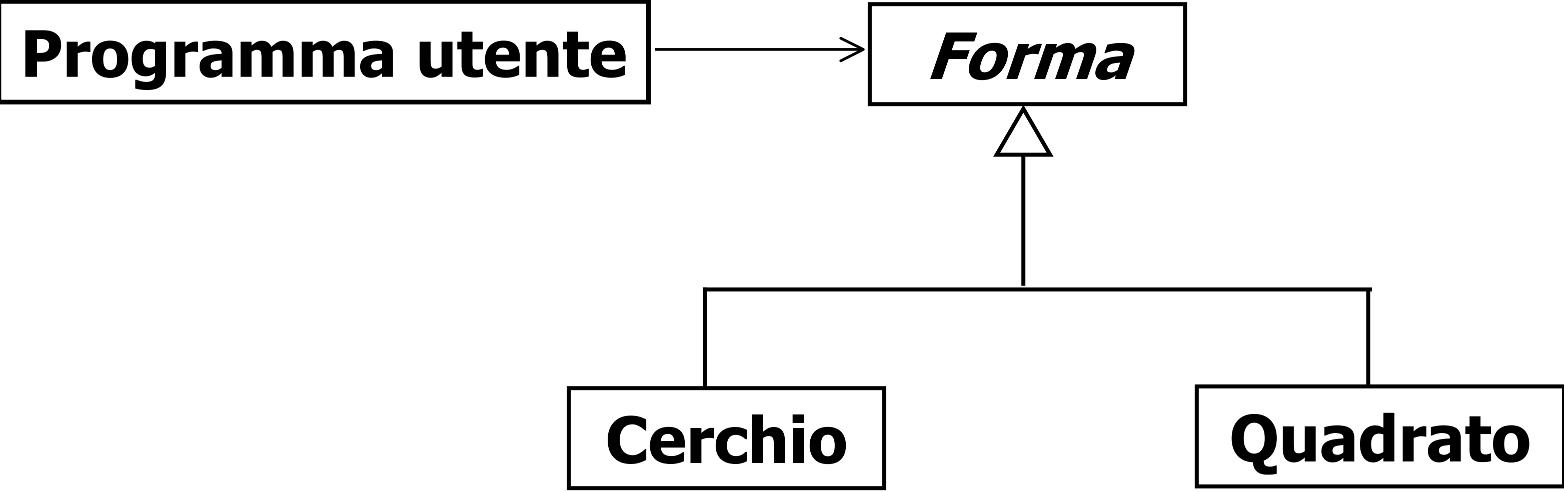
- Gli aspetti comuni sono stati definiti solo una volta
- Gli aspetti diversificati sono stati ridefiniti per ogni specifica forma
- Se il quadrato deve essere modificato in un rettangolo, le modifiche vanno apportate solo alla classe Quadrato
- Se si deve aggiungere un'altra figura non si tocca niente dell'esistente.

In passato…

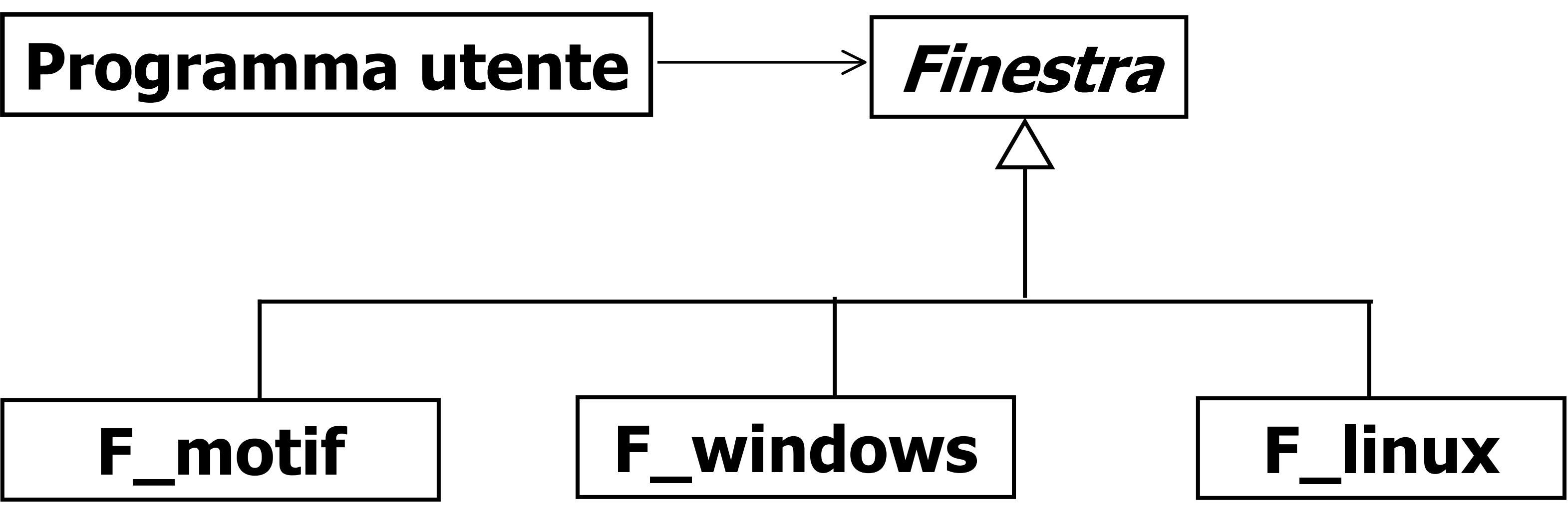
- Si usava la parametrizzazione !!!

```
void Move (elemento el, tipo t);
if (t == cerchio)    {/*  spostamento del
  cerchio*/}
if (t == triangolo)  {/*  spostamento del
  triangolo*/}
if (t == rettangolo) {/*  spostamento del
  rettangolo*/}
```

Il nostro caso



L'interfaccia comune è **molto** di più



Cosa si è ottenuto

- Un programma utente viene sviluppato con riferimento all’interfaccia comune
- Il programma comune non si accorge di quale specifico oggetto è collegato
  - Una finestra Windows?
  - Una finestra Linux?
  - Una finestra Motif?

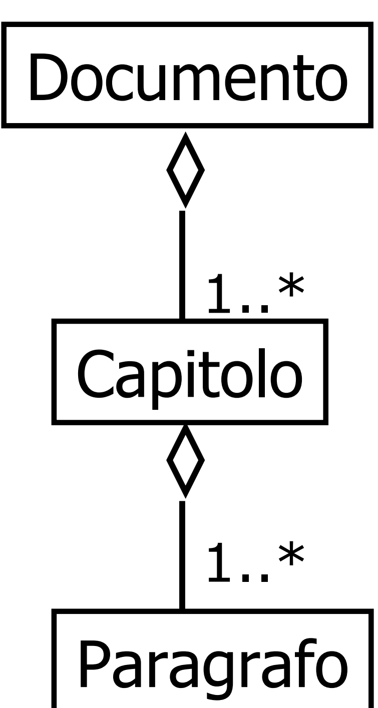
L'ereditarietà è solo uno degli aspetti

- Gli oggetti sono il relazione tra loro
- Associazioni
- Aggregazione/Composizione
  - per il momento focalizziamo su questa

Aggregazione/composizione

- Un oggetto può aggregarne altri
- Essere composto da altri

- Costruzione di oggetti complessi



Aggregazione/composizione

- Aggregazione
  - quando un oggetto si compone di altri che hanno vita autonoma
- Composizione
  - quando un oggetto si compone di altri oggetti che esistono solo in funzione del primo

Aggregazione/composizione

```
class Point {......}
class Color {......}

abstract class Forma {
     Point       center;
     Color       col;
```

- **center** e **col** sono due oggetti
- Sono aggregati o sono componenti?

Aggregazione/composizione

- Due costruttori di **`Forma`**

```
Forma(Point p, Color c){
  centro = p;
      col = c;
  } // aggrega centro e col

Forma(){
      centro = new Point(0.0,0.0);
      col = new Color(0);
  } // costruisce due componenti
```

Aggregazione

- Costruttore di **`Cerchio`**

```
Cerchio(double r, Point p, Color c)
  {   super(p,c);
      this.r= r;   }
```

- questo costruttore costruisce un cerchio che aggrega il centro e il colore

.. Aggregazione

- Nel chiamante

```
Point p1 = new Point(11,11 );
Color c1 = new Color(1);

Cerchio c = new Cerchio(1., p1, c1);
```

c viene costruito aggregandogli un centro e un colore. Due oggetti che hanno esistenza autonoma, indipendente da **c**

..Composizione

- Costruttore di **`Quadrato`**

```
Quadrato(double l) {
      super();
      L=l;
}  // L è il lato
```

- il costruttore di quadrato costruisce al suo interno centro e colore

.. Composizione

- Nel chiamante

```
Quadrato q = new Quadrato(10);
```

q viene costruito in modo che esso stesso costruisca il suo centro e il suo colore; l’esistenza di questi due oggetti dipende dall’esistenza di **q**

Aggregazione/composizione

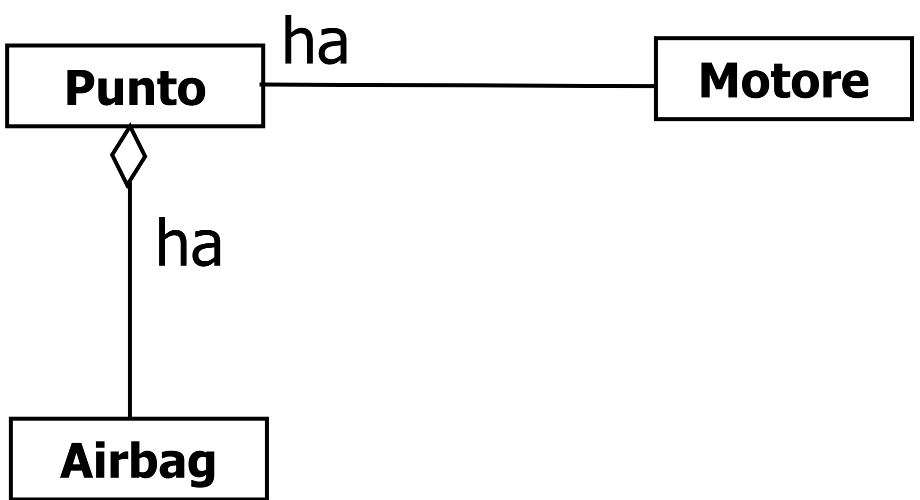
- *Fare molta attenzione: non sempre si fa questa distinzione; spesso si trattano in modo indifferenziato*

- E’ un caso particolare di associazione

Associazione

La Punto ha:

- un motore
- più airbag



Ereditarietà e polimorfismo non bastano….

- L'ereditarietà è statica l'associazione è dinamica
- L'associazione consente di cambiare gli attributi in un qualunque momento
- Se il motore è un attributo lo si può cambiare (anche per un oggetto già esistente, come per una macchina)

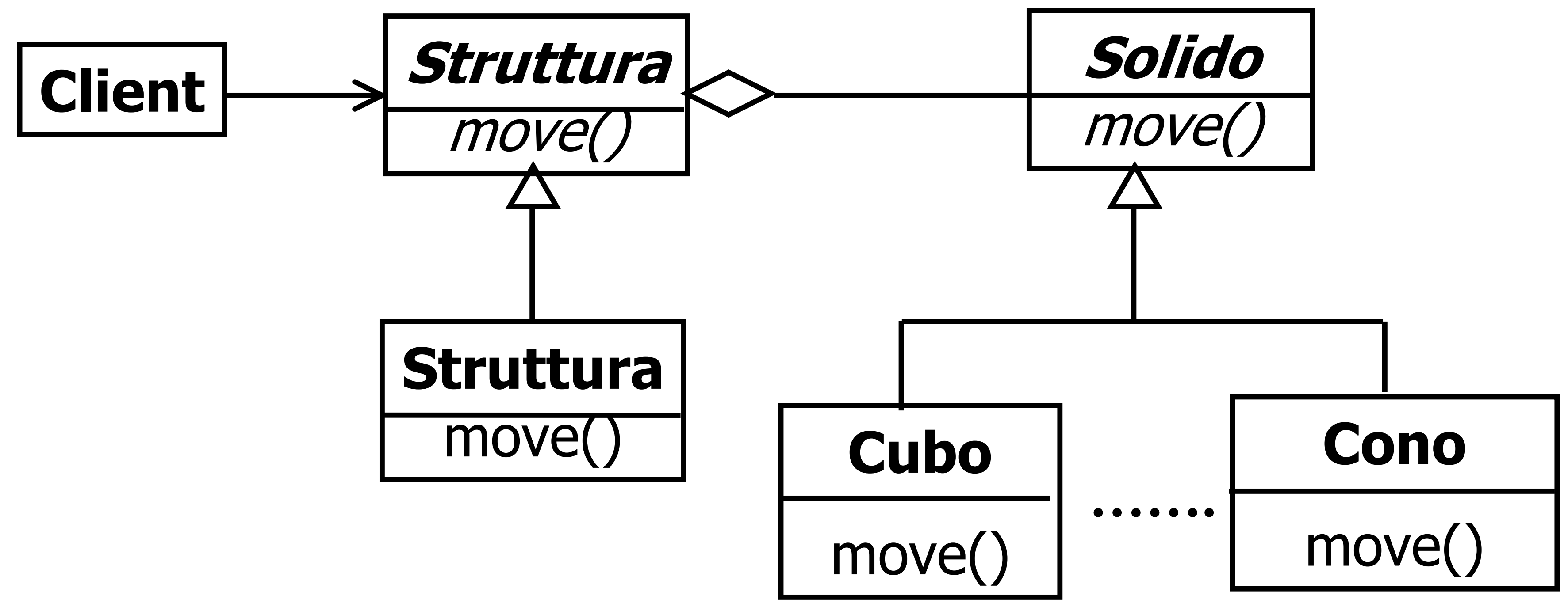
Ereditarietà e polimorfismo non bastano….

- Motore è una classe a parte
- Un'Automobile ha un motore
- Possiamo costruire nuovi motori senza toccare le auto
- Possiamo cambiare un vecchio motore con un nuovo

Composizione/Aggregazione

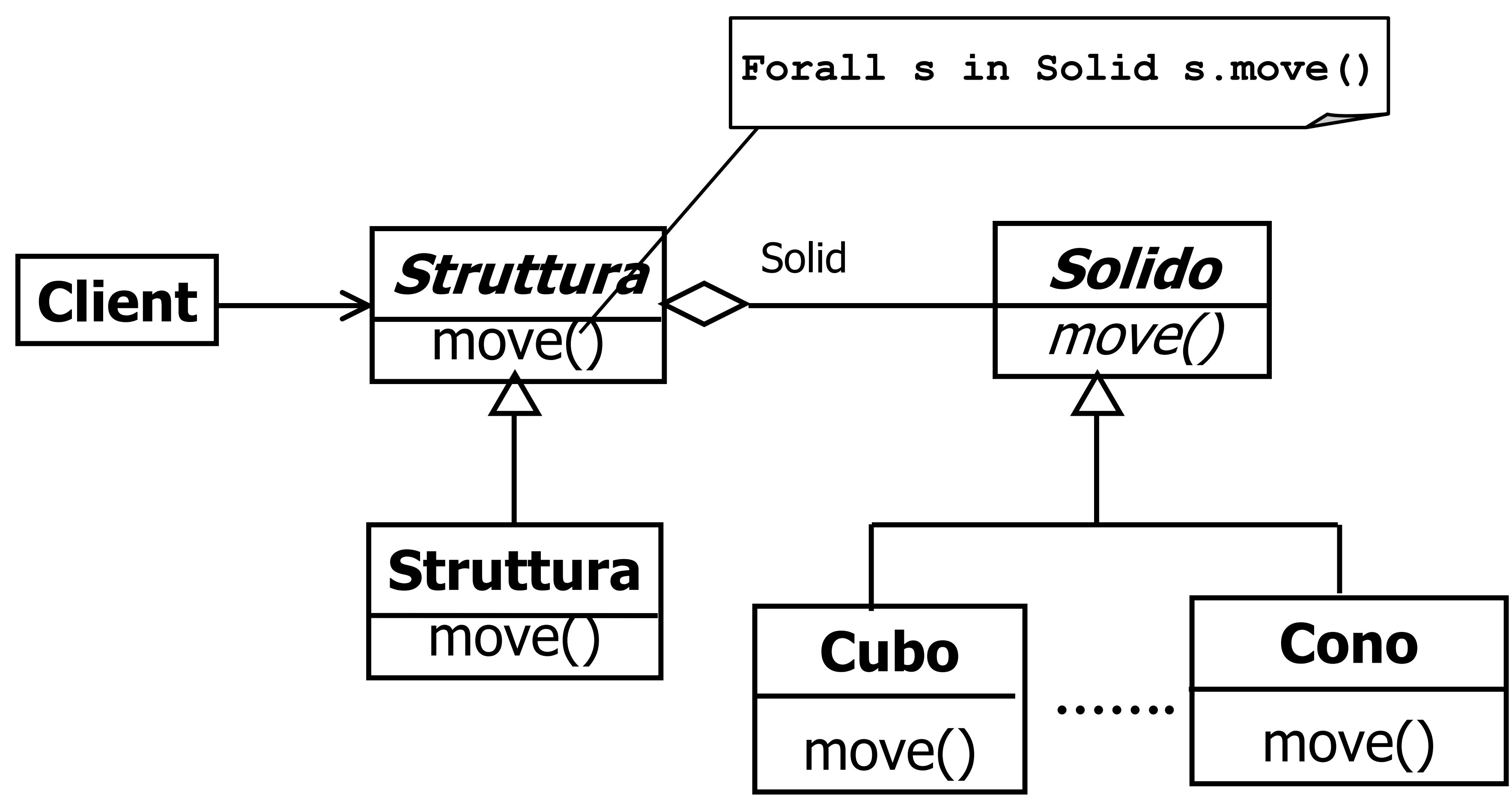
- L'oggetto composto ha un comportamento che dipende dalla somma dei comportamenti dei componenti.
- Una struttura fatta di cubi, coni, piramidi ecc. si muove facendo muovere opportunamente i suoi componenti

Composizione-Polimorfismo

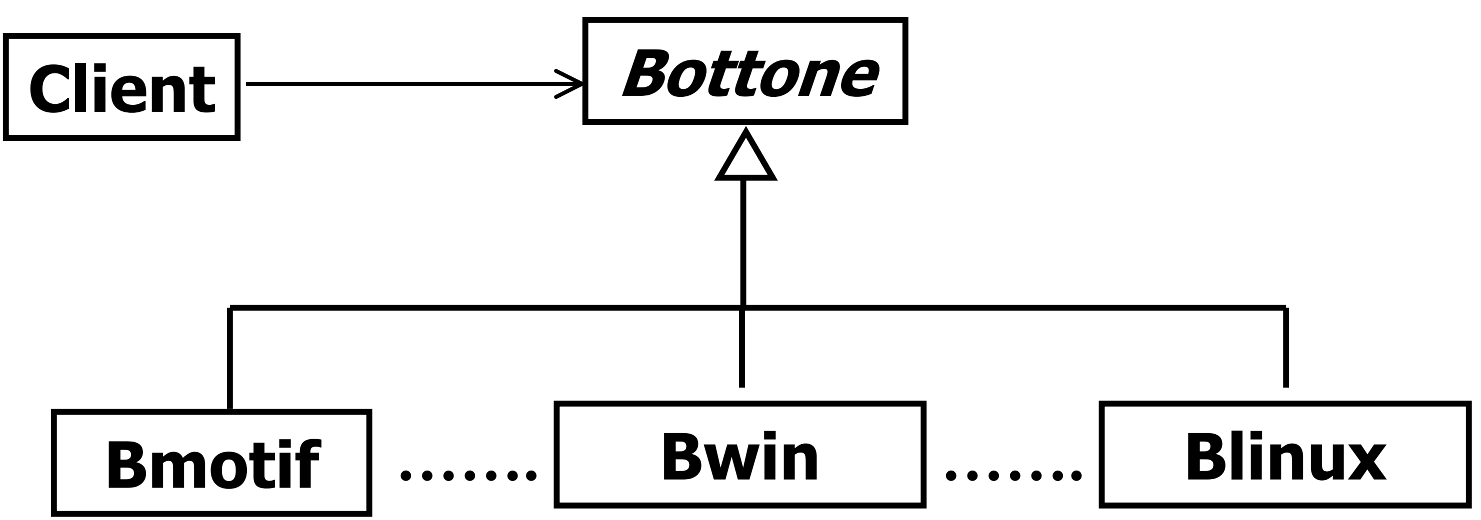


Delega

- L'oggetto che riceve una richiesta delega un altro o altri oggetti a gestire la richiesta stessa
  - La struttura delega i suoi componenti a effettuare il movimento
    - Il **`move()`** della struttura consiste nell'invocazione dei **`move()`** tutti i solidi che la compongono


Forall s in Solid s.move()
Client
Struttura
move()
Solid
Solido
move()
Struttura
move()
Cubo
move()
....
Cono
move()

Oltre il polimorfismo



Programmare verso l'interfaccia non verso l'implementazione

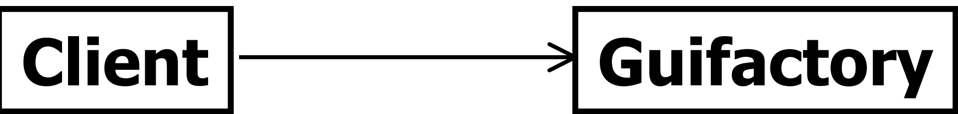
Creazione

- Nell'applicazione (Client):

```
Bottone B = new Blinux();
Finestra F = new Flinux();
```

- Problema:
  - Con un altro L&F si dovrebbe cambiare tutti gli statement di istanziazione
  - E' il caso di proteggere l'applicazione: non deve sapere quale tipo di bottone sta usando, in modo da svilupparla totalmente indipendente dal L&F

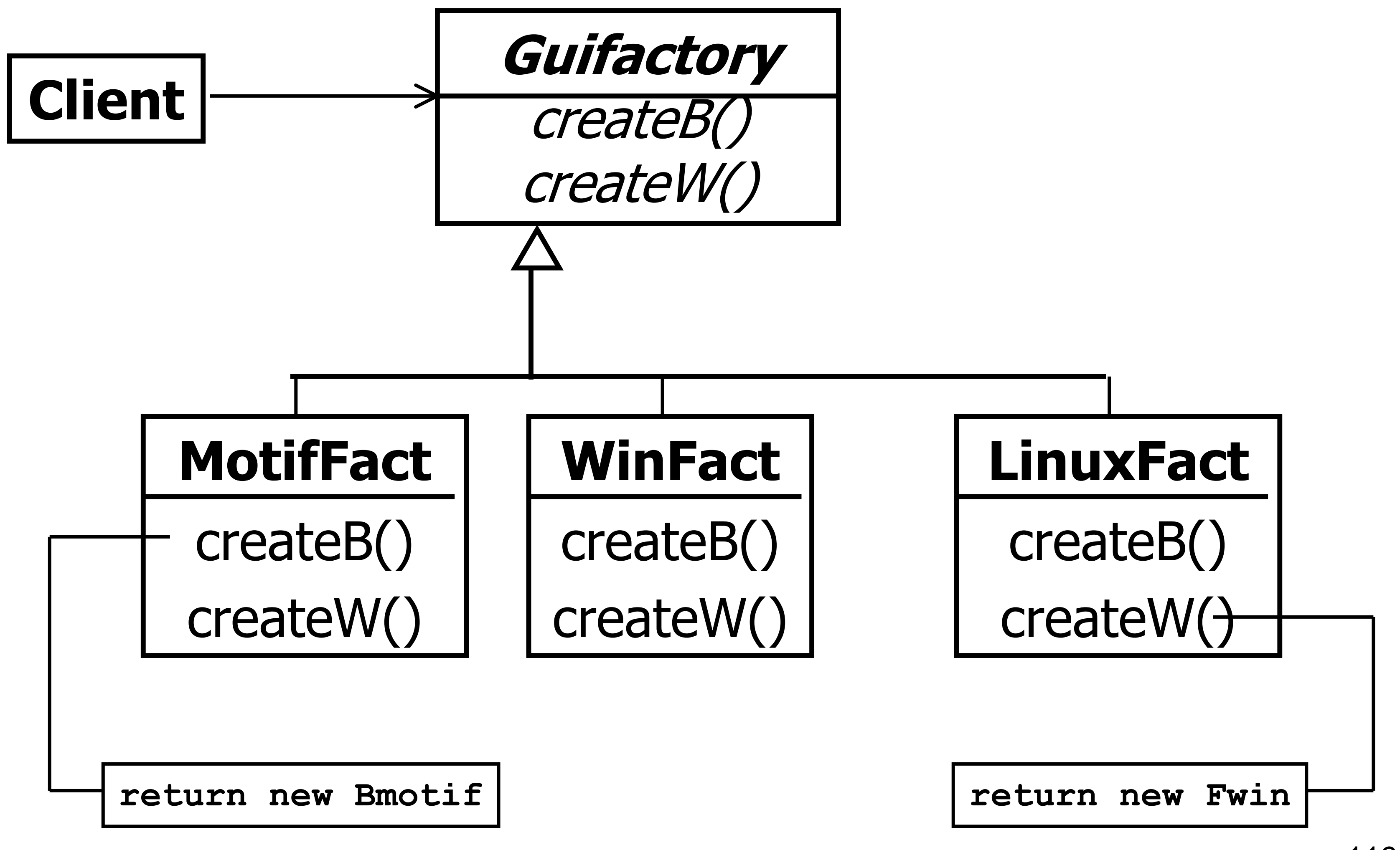
Una *Guifactory*



- Vogliamo portarci a questa situazione
  - una fabbrica di bottoni che sa quale tipo di bottone creare
  - il client non sa cosa c'è sotto

```
Bottone B = Guifactory.createB();
```

Una fabbrica astratta



Nel Client

- Se deve usare L&F Motif:

```
Guifactory guif = new  MotifFact();
```

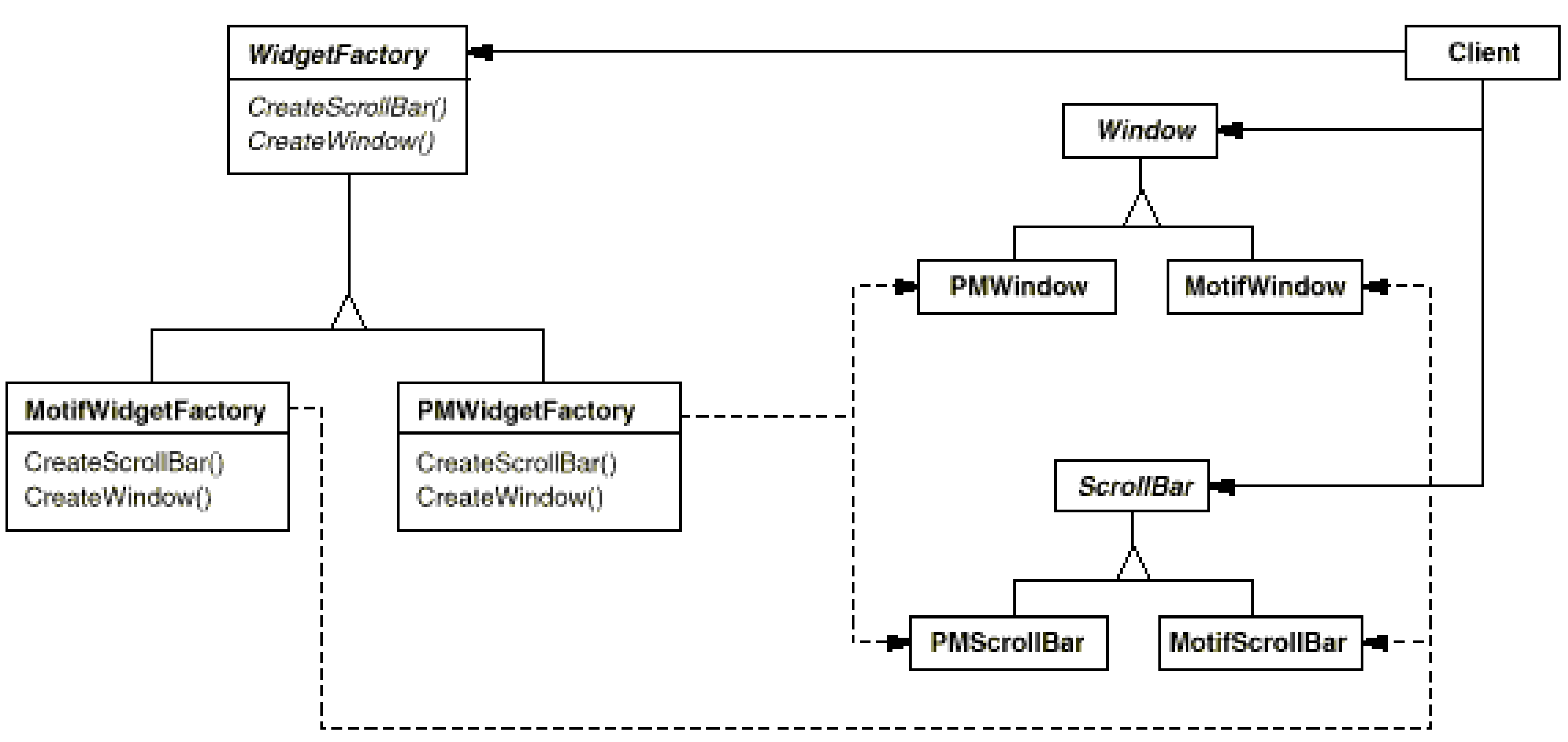
```
Bottone B guif.createB();
Finestra F guif.createW();
```

- Se si vuole un L&F diverso si cambia solo lo statement incorniciato

In conclusione

- L&F diversi non hanno alcun impatto sull'applicazione
- Basta modificare un solo statement per avere un nuovo L&F
- Garantisce un unico standard, evita misture di L&F diversi
- *Gli oggetti non vengono creati dal costruttore standard*
- *Cresce il numero di classi*

E' un Design Pattern



Ereditarietà o composizione?

- L'ereditarietà è statica definita al tempo di compilazione
- L'ereditarietà è facile da usare è parte del linguaggio; rende facile la modifica
- L'ereditarietà viola il principio dell'incapsulameto le classi figlie vedono quello che c'è nella classe padre (una modifica nella classe padre può avere conseguenze sulle classi figlio)

Ereditarietà o composizione?

- La composizione è dinamica attraverso i riferimenti acquisiti al tempo di esecuzione
- La composizione permette la sostituzione di un oggetto con un altro al tempo di esecuzione
- La composizione presuppone che gli oggetti rispettino le interfacce il principio di incapsulamento non viene mai violato

Ereditarietà o composizione?

- La composizione aiuta a tenere ogni classe ben focalizzata e coesa
- Evita la costruzione di gerarchie classi mostruose e ingestibili, ma porta a un aumento del numero degli oggetti

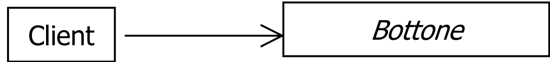
- **Favorire la composizione di oggetti rispetto all'ereditarietà**

- Analisi, progetto, modello concettuale, modello di specifica, modello implementativo ????

Dove sono i confini ?

- I confini non sono mai ben definiti
  - In analisi il modello concettuale applicativo
  - Nel progetto il modello di dettaglio implemetativo
  - Il modello di specifica è a cavallo

- Analisi, mod concettuale



- Modello di specifica

Client

Bottone

push()

Bmotif

push()

Bwin

push()

Blinux

push()

- Progetto, mod implementativo