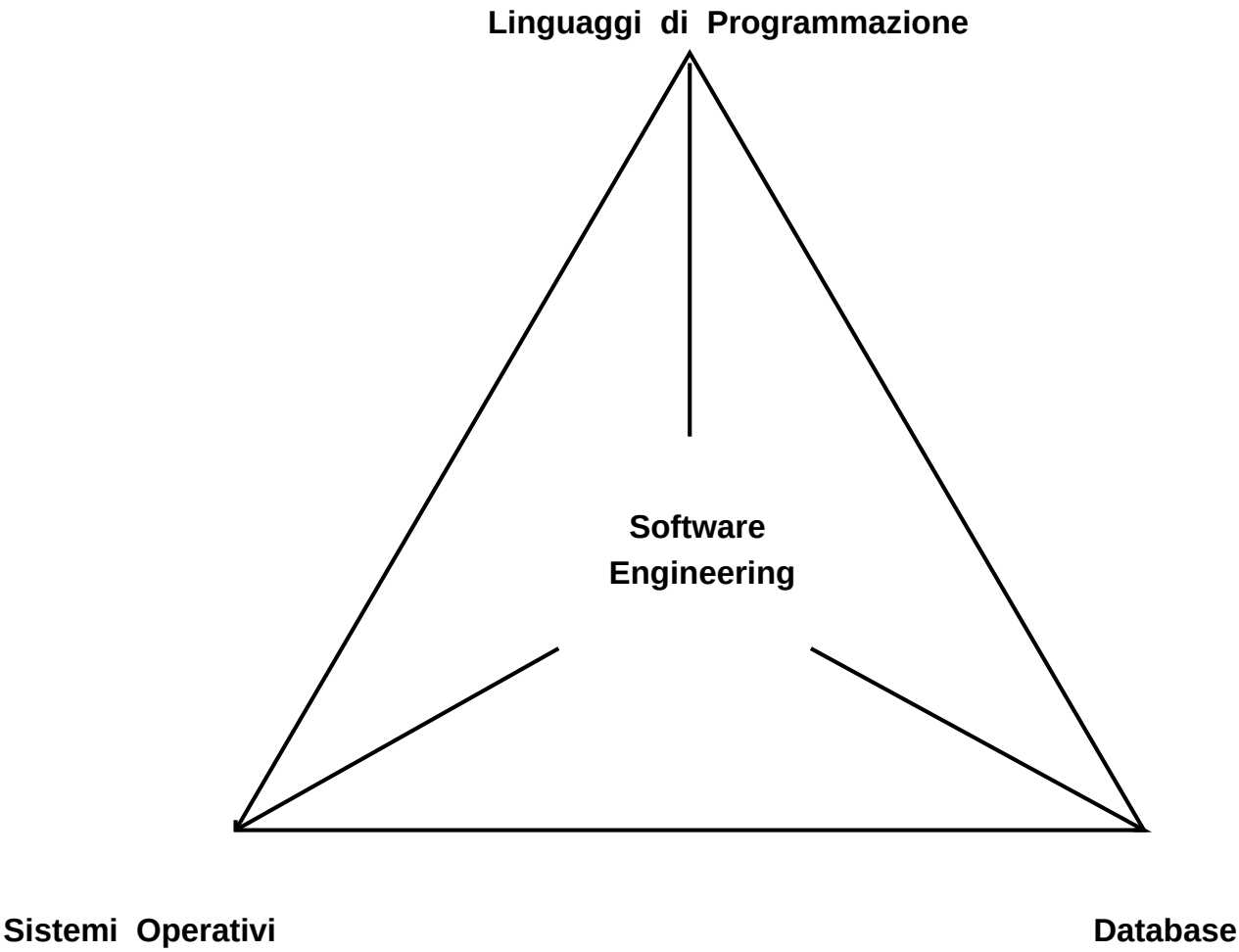
Il Paradigma Object-Oriented in C++

Massimo Ancona Walter Cazzola

11 marzo 1997



Paradigma ad oggetti

Dimensioni del Paradigma Object-Oriented

- *Oggetti*
- *Classi*
- *Identità d'Oggetto*
- *Astrazione e Encapsulation*
- *Interfaccia*
- *Ereditarietà*
- *Tipi e Type Checking*
- *Polimorfismo*
- *Overriding e Late Binding*
- *Overloading*
- *Polimorfismo Parametrico*
- *Metodi e Scambio di Messaggi*
- *Delegazione*
- *Clientship*

Oggetti

Un *oggetto* è l'astrazione di un'entità del mondo reale (ad es. una mela).

- Stroustrup: an object is "a region of storage"
- Meyer: objects are *machines* with an *internal state*, procedures and commands that change the state of the objects, and functions or queries on the state
- Cardelli and Wegner: objects are "data abstractions with an interface of named operations and a hidden local state"
- Wegner: an object is "an entity which has a set of *operations* and a *state* that remembers the effect of operations".
- Snyder: "An object is an identifiable entity that plays a visible role in providing a service that a client (user or program) can request... An object explicitly embodies an abstraction characterized by the behavior of certain requests."
- Booch: "Something you can do things to. An object has state, behavior and identity; the structure and behavior of similar objects are defined in their common class. The terms *instance* and *object* are interchangeable."
- Goldberg: an object is defined as an entity "composed of a private memory and a set of operations".

Quando un oggetto è un dato, cioè può essere passato come parametro o assegnato ad una variabile, allora viene detto oggetto di *prima classe* oggetto di *seconda* classe altrimenti.

Classi, Istanze e Metodi

> "*Classes* serve as templates from which objects can be created [Wegner]."

Le classi descrivono le caratteristiche comuni ad un certo gruppo di oggetti.

> "A class is a set of objects that share a common structure and a common behavior. [Booch]."

Queste caratteristiche possono essere qualitative e vengono dette ***attributi*** oppure comportamentali e sono dette operazioni o ***metodi***. Gli attributi caratterizzano lo stato interno di un oggetto.

> "A method is a procedure that performs services. Typically, an object has one method for each operation it supports [Snyder]."

Ogni metodo è strettamente connesso ad un oggetto.

Gli oggetti sono ottenuti istanziando la classe. Gli attributi possono essere: *variabili istanza* o *di classe*. Le variabili istanza caratterizzano un oggetto della classe, mentre le variabili di classe sono comuni a tutti gli oggetti della classe.

Identità d'Oggetto

Gli oggetti, analogamente alle entità che modellano, hanno una propria identità.

Oggetti istanza della stessa classe, sono oggetti diversi, anche se sono istanziati con gli stessi valori (oggetti uguali, ma non identici).

Classi e Metodi in C++

In C++ una classe è definita dalla keyword class. In C++ le struct sono un particolare tipo di classe, la differenza con quest'ultime, riguarda solo le regole di visibilità degli attributi che definisce.

Negli esempi si adotterà, prevalentemente, la keyword class.

```
class point {
  ...
 int x,y;
 void draw(void){...};
 point(int x, y){...};
 ~point();
   ...
}
```

La classe *point* descrive i punti tramite le coordinate (attributi x, y) ed una operazione (il metodo *draw*).

I metodi sono delle funzioni definite all'atto della definizione di classe, l'implemetazione può essere contestuale (inserita di seguito es. con *class*) o differita (come nel caso *struct*).

```
void point::draw(void) { ... };
```

Si noti la presenza dei metodi *point* e $\sim$*point*, rispettivamente, costruttore e distruttore delle istanze della classe *point*.

NB. Gli attributi C++, in generale, sono variabili d'istanza. È possibile definire variabili di classe, qualificando gli attributi con il qualificatore static. Un esempio è mostrato in seguito.

Creazione e Distruzione di Oggetti in C++

Il costruttore di una classe permette di inizializzare le istanze della stessa. Il distruttore permette di effettuare azioni sugli oggetti prima di eliminarli (ad es. garbage-collecting). È buona norma dotare ogni classe di costruttore e distruttore.

Si hanno i seguenti modi per istanziare degli oggetti:

- automaticamente: l'oggetto è creato all'atto della definizione. Ad esempio:

```
point p = point(0,1);
point q(0,1)
```

  I punti p e q vengono creati ed inizializzati. Il C++ non permette di creare un oggetto automatico (nel senso del C) non inizializzato.

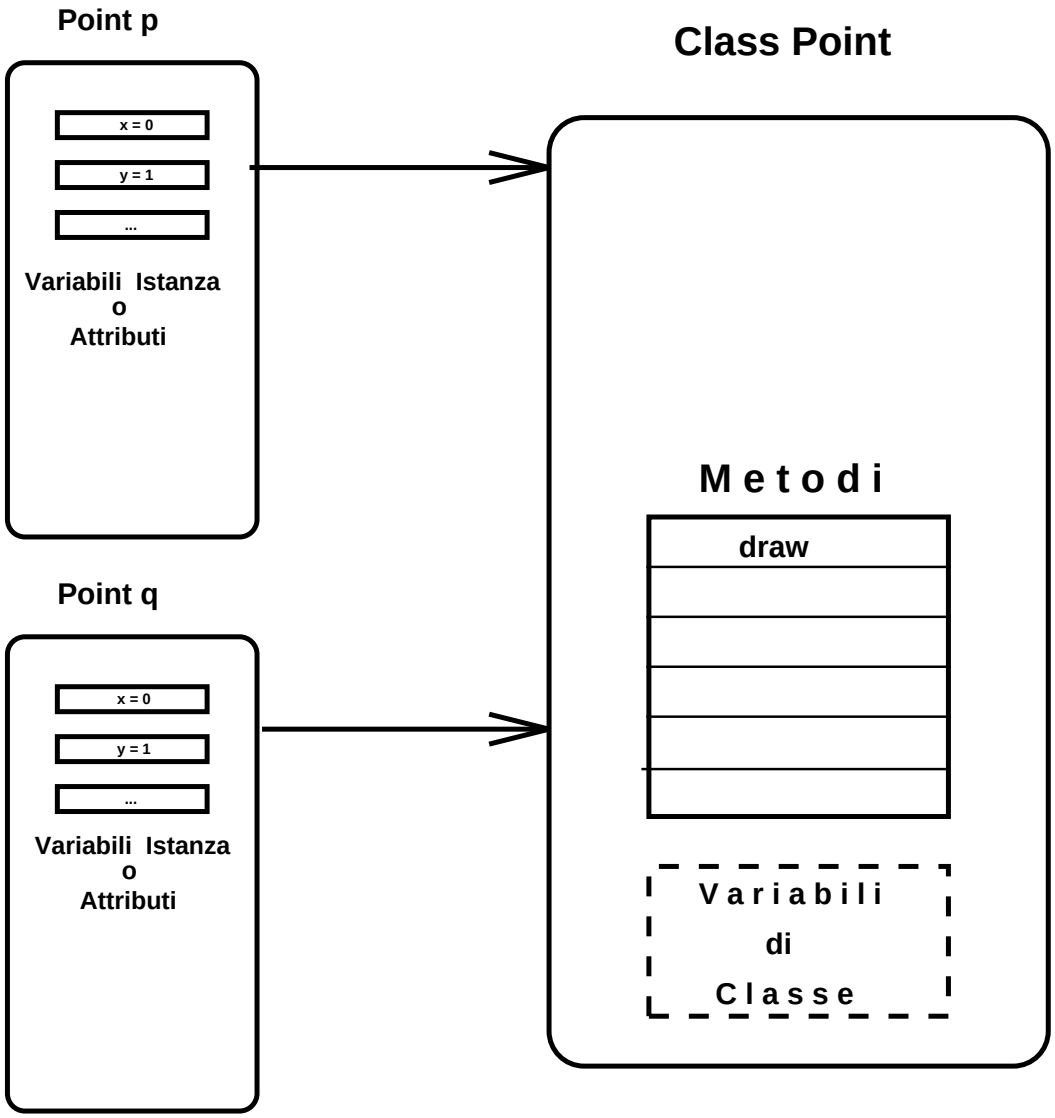
- in memoria dinamica: la variabile è definita come un puntatore e verrà allocata, successivamente tramite l'operazione *new*.

```
point *pp, *pq; pp = new point(0,1); pq = pp;
```

In entrambi i casi è attivato (per difetto) il costruttore della classe.

Gli oggetti automatici vengono distrutti nel momento in cui, nel programma, non si hanno più riferimenti attivi agli stessi. Gli oggetti creati con *new* devono essere distrutti esplicitamente con *delete*. In entrambi i casi viene eseguito il distruttore della classe.

NB. A causa del concetto di identità d'oggetto l'espressione `p == q` ritorna `false`, mentre `pp == pq` ritorna `true`.


Point p
Class Point
x = 0
y = 1
...
Variabili Istanza
o
Attributi
Metodi
draw
Point q
x = 0
y = 1
...
Variabili Istanza
o
Attributi
Variabili
di
Classe

Object e Class Based ed Object-Oriented

Un linguaggio è detto *object-based* se supporta gli oggetti come una primitiva del linguaggio. Es. Ada, Modula-2, gli oggetti sono i packages ed i moduli rispettivamente.

Un linguaggio è detto *class-based* se è object-based e se ogni oggetto ha una classe. Es. CLU, le classi sono i moduli, mentre gli oggetti sono le variabili del tipo associato al modulo.

Un linguaggio è detto *object-oriented* se è class-based e se le classi hanno una struttura gerachica definita da una relazione di *ereditarietà*. Es. C++ e SmallTalk sono object-oriented.

Per poter essere considerato object-oriented, un linguaggio class-based, necessita della presenza di una relazione di ereditarietà che strutturi, le classi, in una gerarchia.

Ereditarietà

L'ereditarietà è un meccanismo che permette di combinare una o più classi, per formarne un'altra con le caratteristiche delle classi originali.

> "A class may inherit operations from 'superclasses' and may have its operations inherited by 'subclasses' [Wegner]."

La classe da cui si eredita è detta *classe base* o *madre*, la classe che eredita è detta *classe derivata* o *figlia*.

Si parla di *ereditarietà singola* se le classi ereditano solo da una classe. Si parla di *ereditarietà multipla* se le classi ereditano da più classi.

L'ereditarietà è un meccanismo utile per:

- Riusare|condividere informazioni tra classi, ad es. la classe circle che acquisisce dalla classe point le coordinate del centro.
- Specializzare le classi, come nel caso della classe person che viene specializzata nella classe student, aggiungendo i riferimenti al tipo di studi effettuati.

NB. Le istanze della classe derivata, in generale, sono anche istanze della classe base, non è vero il viceversa. È, quindi, corretto asserire che lo studente è una persona, ma è errato dire che una persona è uno studente.

Ereditarietà in C++

Considerata la solita classe *point* deriviamo la classe *circle*.

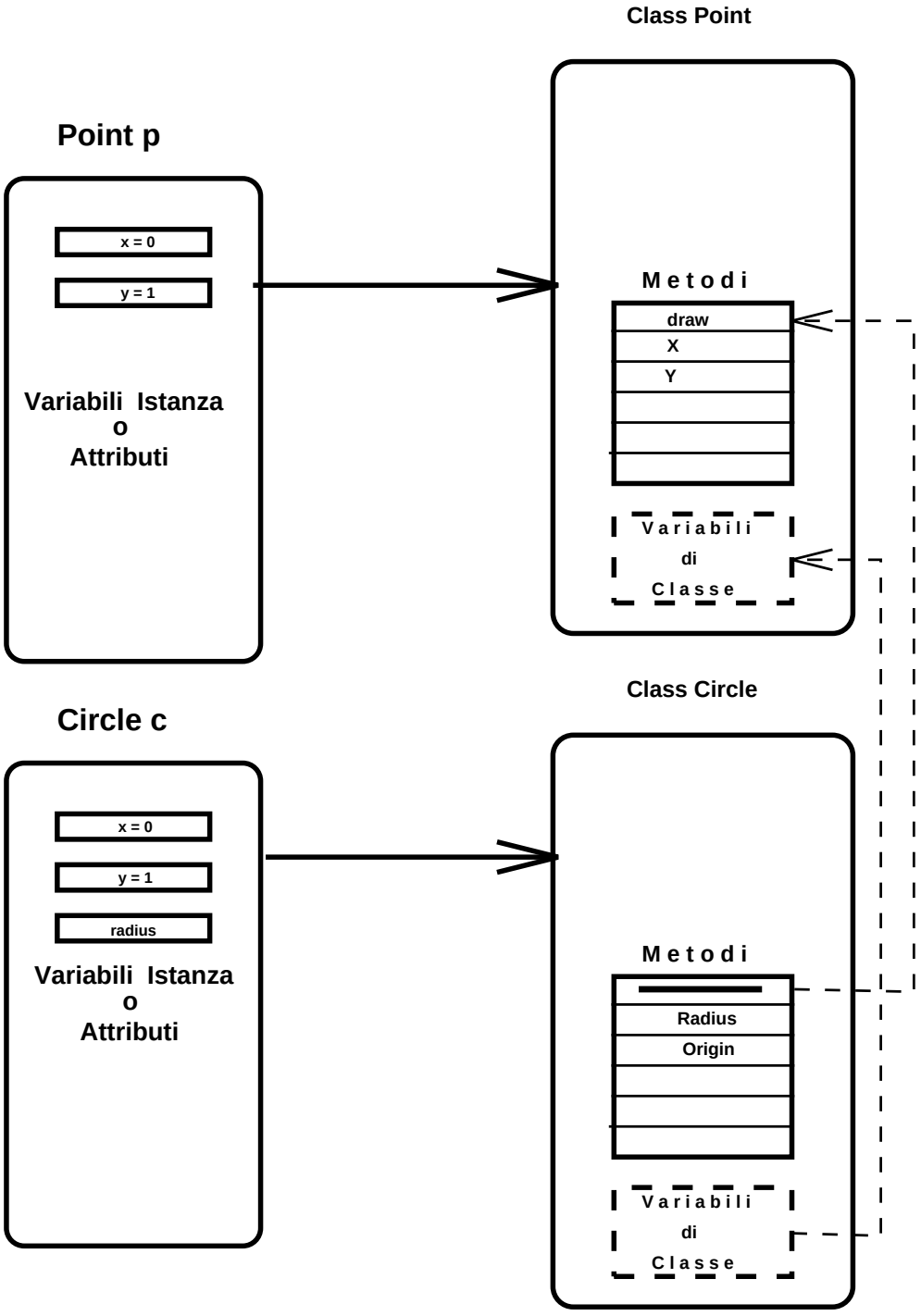
```
class circle : point {
  int radius;
       ...
  circle( int x, int y, int r );
  ~circle() : ~point() {};
       ...
}

circle::circle( int x, int y, int r )
          : point(x,y), radius(r) {};
```

La classe *circle*, possiede tutti gli attributi ed i metodi definiti nella classe *point*, a cui aggiunge l'attributo *radius*.

Sia il costruttore che il distruttore, di una classe derivata, devono richiamare, rispettivamente, il costruttore ed il distruttore della classe base e prevedere anche i parametri necessari all'attivazione di quest'ultimi.

NB. Il termine `radius(r)` è un'abbreviazione dell'espressione `radius = r`, utilizzabile solo nel costruttore.


Class Point
Point p
x = 0
y = 1
Variabili Istanza
o
Attributi
Metodi
draw
X
Y
Variabili
di
Classe
Class Circle
Circle c
x = 0
y = 1
radius
Variabili Istanza
o
Attributi
Metodi
Radius
Origin
Variabili
di
Classe

Astrazione ed Encapsulation

L'astrazione e l'encapsulation sono due dei concetti fondamentali della metodologia object-oriented.

> "Abstraction is the process of focusing upon the essential characteristics of an object [Booch]."

> "Encapsulation is the process of hiding all the details of an object that do not contribute to its essential characteristics. The structure of an object is hidden, as well as the implementation of its methods [Booch]."

Encapsulation e information hiding sono concetti interscambiabili e sono relativi alla visibilità dei dati e delle operazioni.

Concettualmente si hanno due livelli di encapsulation, detti di prima e seconda classe:

- l'encapsulation di prima classe è relativa a nascondere i dettagli implementativi,
- l'encapsulation di seconda classe è relativa ad impedire gli accessi sconsiderati ai dati.

Nei linguaggi object-oriented le unità di incapsulamento sono gli oggetti: il loro stato è accessibile solo tramite i metodi messi a disposizione dall'oggetto e l'implementazione stessa dei metodi è separata dalla parte dichiarativa della classe.

Interfacce

"All information about a module should be private to the module unless it is specifically declared public. [Meyer]"

"An interface is the information about a function that caller need to know including: its expected arguments, the jobs it does, and its returned values [Keene]."

Interfaccia degli Oggetti

"Each class must have two parts: an interface and an implementation. The *interface* of a class captures only its outside view. The implementation of a class comprises the representation of the abstraction as well as the mechanisms that achieve the desired behavior.[Booch]"

L'interfaccia è un aspetto che discende direttamente dai concetti di astrazione ed encapsulation.

Grazie all'astrazione è possibile progettare l'interfaccia di una classe attraverso le sue caratteristiche essenziali che la distinguono da ogni altra classe.

Grazie all'encapsulation vengono nascosti, dietro all'interfaccia della classe, tutti i dettagli implementativi.

Solamente l'interfaccia di un oggetto è visibile agli altri oggetti, e solo attraverso i servizi offerti dall'interfaccia è possibile interagire con l'oggetto.

Encapsulation in C++

Il C++ implementa sia l'encapsulation di prima che di seconda classe.

L'encapsulation di prima classe è ottenuta grazie alla possibilitià di separare (su più file) l'implementazione dei metodi dalla definizione degli stessi (usando l'operatore di scope `::`).

L'encapsulation di seconda classe è ottenuta tramite l'uso dei qualificatori d'accesso: `private, protected` e `public`.

- `private:` le informazioni definite private sono inaccessibili all'esterno della classe, compreso alle classi derivate,
- `protected:` le informazioni sono visibili solo nelle classi derivate,
- `public:` le informazioni pubbliche possono essere utilizzate in ogni contesto.

Le informazioni pubbliche formano l'interfaccia delle istanze di una classe, con cui gli altri oggetti possono interagire.
Le informazioni pubbliche e protette formano l'interfaccia della classe verso le altre classi (clienti e discendenti).

Il qualificatore `protected` permette di escludere dall'interfaccia delle classi derivate, quanto ereditato dalla classe base.

NB. Per default, le classi introdotte dalla keyword `class` definiscono i propri dati come privati, viceversa le classi introdotte dalla keyword `struct` definiscono i propri dati come pubblici.

Encapsulation in C++ (segue)

È buona norma di programmazione definire lo stato degli oggetti come privato, permettendone l'accesso solo tramite metodi specifici, detti *selettori* e *modificatori*

```
class point {
  int x,y;
 public:
  void draw(void);
  point(int x,y):x(x),y(y){};
  ~point() {};
  // selector
  int X() {return x;};
  int Y() {return y;};
 protected:
  // modifier
  void set_x(int X) {x=X};
  void set_y(int Y) {y=Y};

    ...
}
```

```
class circle : public point {
  int radius;
 public:
  double area() {pi*radius*radius;};
  circle(int x,y,r): point(x,y), radius(r){};
  ~circle() : ~point() {};
  // selector
  int Radius() {return radius;};
  point Origin() {return point(X(),Y());};
 protected:
  // modifier
  void set_radius(int r) {radius=r};
  void set_origin(point p) {set_x(p.X());
                            set_y(p.Y());};}
    ...
}
```

L'interfaccia della classe *point* è: draw, X, Y, set_x, set_y più il costruttore ed il distruttore; mentre l'interfaccia delle sue istanze è uguale all'interfaccia della classe escluso set_x e set_y.

L'interfaccia delle istanze della classe *circle* è draw, X, Y, area, Radius, Origin più costruttore e distruttore; l'interfaccia della classe è la stessa con in più set_radius, set_origin.

Encapsulation in C++ (segue)

Comportamento dell'encapsulation (p è istanza di point e c è istanza di circle):

- p.x è illegale! x è un attributo privato di p
- p.X() è corretto! X è il selettore di p ed è dichiarato pubblico.
- p.set_x(...) è illegale! set_x è definito *protect*, può essere usato solo nelle classi discendenti da *point* (vedi set_origin).
- c.X() è corretto! X è un metodo pubblico ereditato dalla classe *point*.
- c.area() è corretto! area è un metodo pubblico della classe *circle*.
- c.set_radius(...) è illegale! set_radius è un metodo protetto definito dalla classe *circle*; può essere usato nelle classi discendenti

Tipo

"A *type* is a precise characterization of structural behavioral properties which a collection of entities all share [Deutsch]."

"The terms class and type are usually interchangeable; a class is a slightly different concept than a type, in that it emphasizes the importance of hierarchies of classes.
Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways [Booch]."

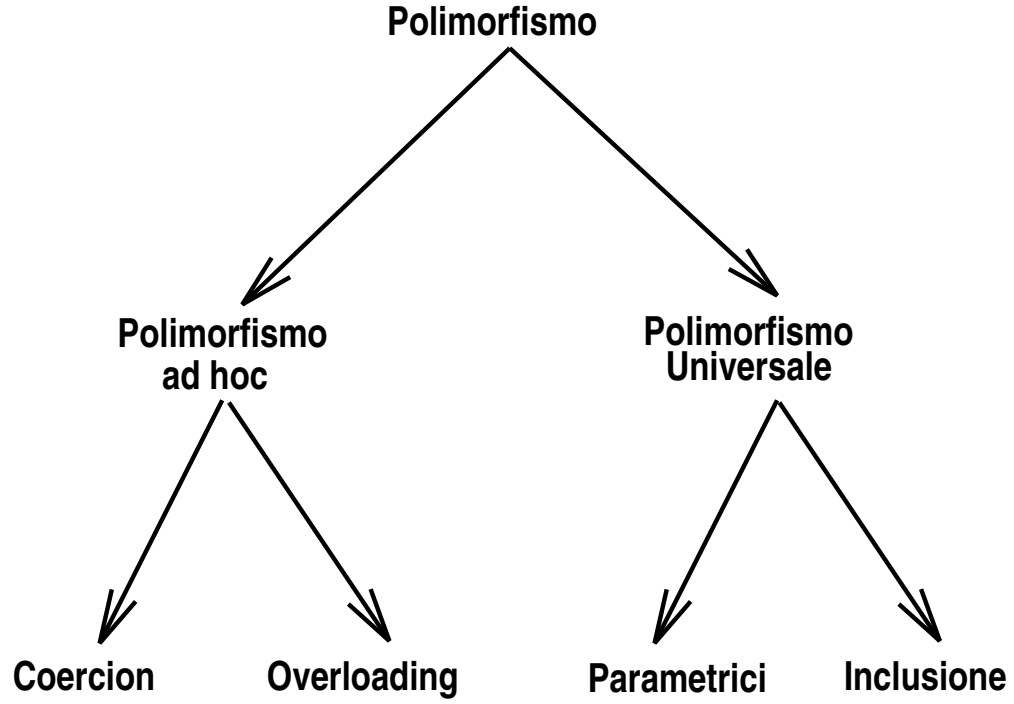
Strong Typing, Static and Dynamic Binding

Strong typing riguarda la consistenza dei tipi, mentre il *binding* riguarda il momento in cui le informazioni sui tipi vengono collegate ai nomi.

- *static binding*: i tipi delle variabili e delle espressioni sono fissati al momento della compilazione;

- *dynamic binding* (detto anche *late binding*): i tipi delle variabili e delle espressioni non sono note fino all'esecuzione.

Ada è un linguaggio fortemente tipato con binding statico.
SmallTalk è un linguaggio non tipato con binding dinamico.
Object Pascal e C++ sono linguaggi fortemente tipati con binding dinamico.



"*Polymorfic languages* are those in which some values and variables may have more than one type. [Cardelli-Wegner]"

In un regime di *polimorfismo universale*, le entità polimorfe sono applicabili ad numero infinito di tipi, e per ogni tipo eseguono sempre lo stesso codice (ad es. ML).

In un regime di *polimorfismo ad hoc*, le entità polimorfe sono applicabili ad un numero limitato di tipi e per ogni tipo possono eseguire codice diverso (ad es. C++).

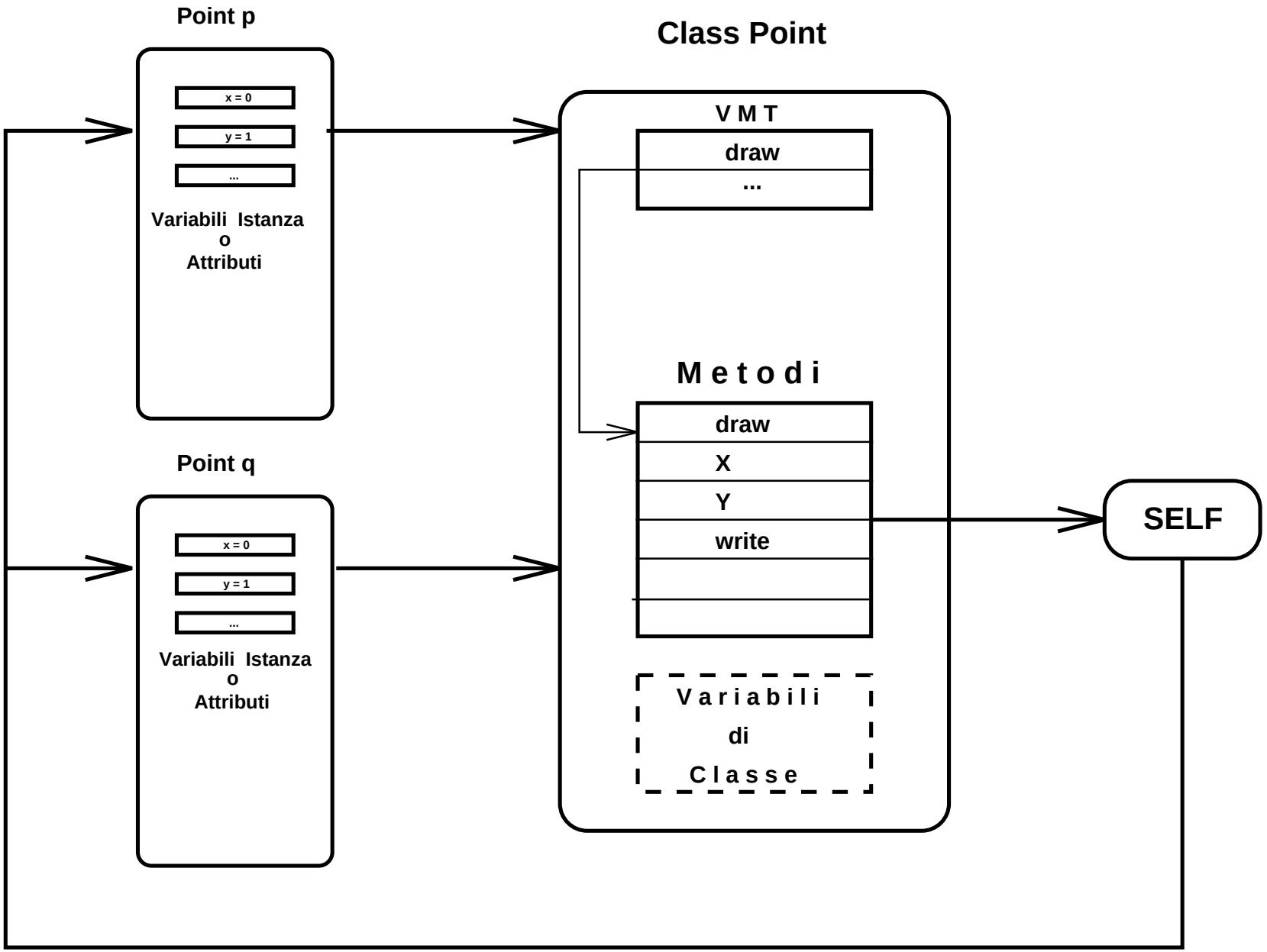
Proprio dei linguaggi object-oriented è il polimorfismo per inclusione, trascendente dal tipo e legato al comportamento. A seconda del contesto un'entità polimorfa può presentare un comportamento differente (*ridefinizione*).

Overriding (Ridefinizione)

Una classe derivata, può ridefinire (override) uno o più metodi ereditati dalla classe base, fornendone, quindi, una nuova implementazione.

La ridefinizione è il meccanismo che sta alla base del polimorfismo comportamentale; infatti, a seconda dell'istanza viene attivato un'implementazione del metodo ridefinito piuttosto che un'altra.

Per poter funzionare correttamente è necessario che i nomi dei metodi vengano legati dinamicamente (late binding) all'implementazione.


Point p
x = 0
y = 1
...
Variabili Istanza
o
Attributi
Point q
x = 0
y = 1
...
Variabili Istanza
o
Attributi
Class Point
V M T
draw
...
M e t o d i
draw
X
Y
write
V a r i a b i l i
di
C l a s s e
SELF

Ridefinizione e Metodi Virtuali in C++

In C++ la ridefinizione avviene semplicemente ridefinendo il metodo in questione nella classe derivata.

```
class point {                    class circle : point {
  int x, y;                        int radius;
 public:                          public:
  virtual void draw(void);         virtual void draw(void);
  void write(void);                void write(void);
       ...                              ...
};                               };
void point::draw(void) {cout << "I'm a point\n";};
void point::write(void) {cout << "I'm point's write\n";};}
void circle::draw(void) {cout << "I'm a circle\n";};
void circle::write(void){cout <<"I'm circle's write\n";};}

point p(1,2), *pp = new point(0,1);
circle c(0,1,2), *pc = new circle(3,2,7);
```

Nell'esempio il metodo `draw` viene ridefinito nella classe *circle* per far stampare il tipo di oggetto che si sta usando.

I metodi introdotti dalla keyword virtual sono detti *metodi virtuali* e per loro viene utilizzato il binding dinamico.

Il binding dinamico è attivato solo per oggetti creati dinamicamente.

```
p.draw(); //- risultato: I'm a point
c.draw(); //- risultato: I'm a circle
p = c;    //- un cerchio e' anche un punto
p.draw(); //- risultato: I'm a point
```

NB. Siccome una classe derivata è sottotipo della classe base, le sue istanze si possono assegnare ad istanze della classe base.

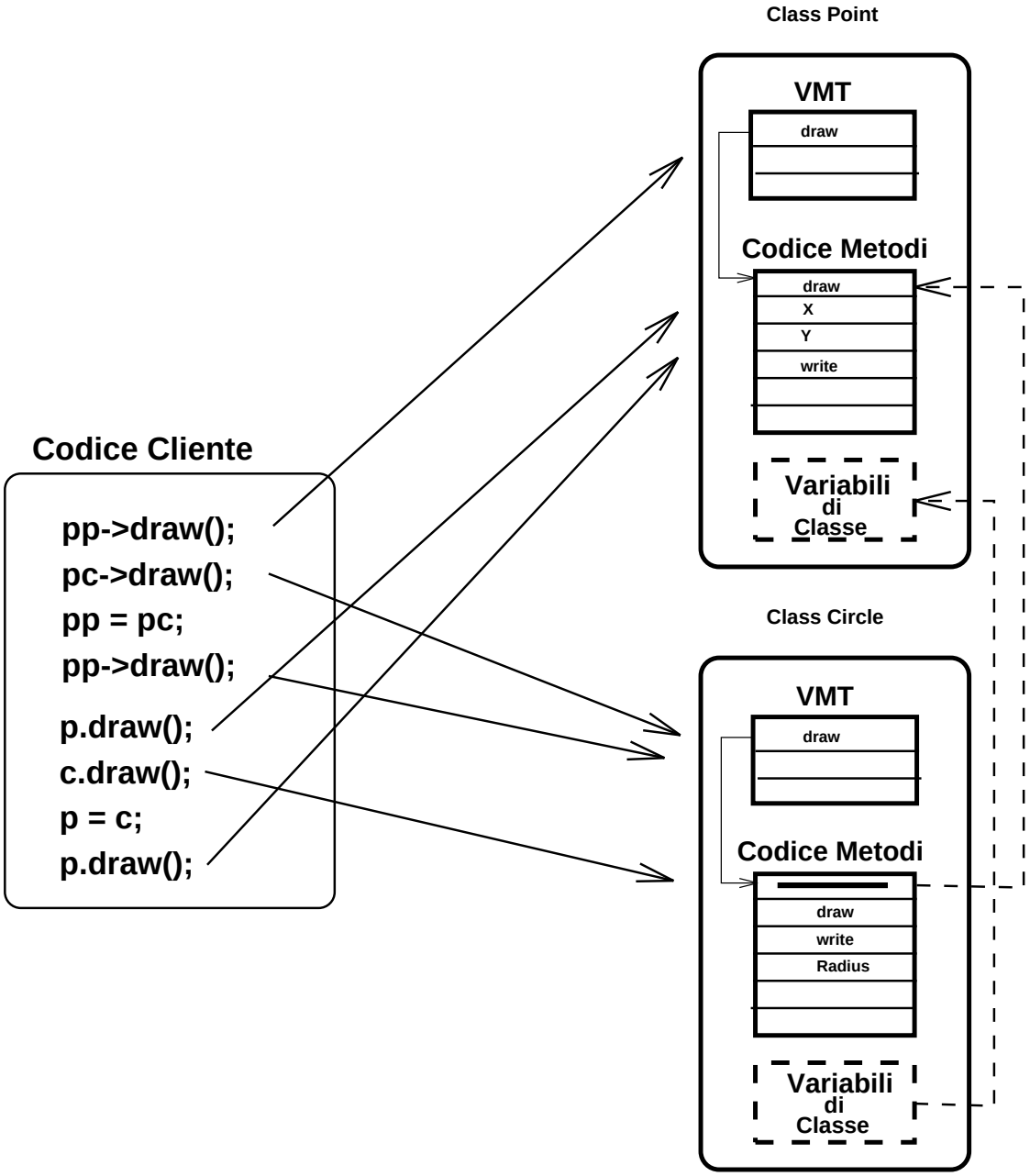
Ridefinizione e Metodi Virtuali in C++ (segue)

Siccome `write` non è un metodo dinamico ad ogni sua attivazione verrà utilizzata l'implementazione relativa al tipo statico dell'oggetto attivatore.

```
pp->write(); //- risultato: I'm point's write
pc->write(); //- risultato: I'm circle' write
pp = pc;
pp->write(); //- risultato: I'm  point's write
```

Siccome `draw` è un metodo dinamico ad ogni sua attivazione verrà scelta l'implementazione relativa al tipo dinamico dell'oggetto attivatore.

```
pp->draw(); //- risultato: I'm a point
pc->draw(); //- risultato: I'm a circle
pp = pc;
pp->draw(); //- risultato: I'm a circle
```


Class Point
VMT
draw
Codice Metodi
draw
X
Y
write
Variabili
di
Classe
Codice Cliente
pp->draw();
pc->draw();
pp = pc;
pp->draw();
p.draw();
c.draw();
p = c;
p.draw();
Class Circle
VMT
draw
Codice Metodi
draw
write
Radius
Variabili
di
Classe

Overloading

Quando, nello stesso scope, più dichiarazioni di funzioni sono specificate per un singolo nome, quel nome è detto *overloaded*.

Al momento dell'uso la funzione appropriata viene selezionata confrontando il tipo degli argomenti attuali con il tipo degli argomenti formali.

> "In *overloading*, the same variable name is used to denote different and the context is used to decide which function is denoted by a particular instance of the name. [Cardelli-Wegner]"

L'overloading è semplicemente un modo sintattico per usare lo stesso nome per oggetti semanticamente diversi.

Esempio classico di operazioni overloaded è rappresentato dalle operazioni matematiche classiche, come il `+` e il `*`; queste possono essere usate sia con operatori interi che reali, nello stesso modo e sarà poi il compilatore a discriminare quale implementazione è più corretto usare.

Molti linguaggi (C++ e ML) permettono al programmatore di effettuare l'overloading sia di operazioni predefinite che user-defined.

Overloading in C++

In C++ è possibile sovraccaricare la maggioranza degli operatori predefiniti (compreso (), [], =, & e ->) e le operazioni user-defined.

L'overloading degli operatori è realizzato definendo una nuova segnatura per l'operatore in questione più la relativa implementazione.

```
point point::operator +(point p) { ... };
```

Gli operatori, in C++, sono particolari metodi introdotti dalla keyword operator.

NB. Oltre alla possibilità di sovraccaricare gli operatori esistenti, il C++ permette di definire nuovi operatori (come ad es. **).

L'overloading di metodi è ottenuto, semplicemente, definendo più metodi con segnatura diversa. Sarà il tipo ed il numero degli argomenti a permettere di discriminare il metodo da attivare.

Esempi di metodi user-defined overloaded ci sono forniti dai costruttori, che spesso sono forniti in più versioni, utilizzate oltre che per creare oggetti anche per convertirli da un tipo in un altro.

```
circle::circle(int x, int y, int r) ...;
circle::circle(point p, int r) ...;
```

Ad esempio, si può dotare la classe *circle* di due costruttori, uno che crea un cerchio a partire dalle singole componenti, e l'altra che lo crea a partire dalle coordinate dell'origine più il raggio.

Polimorfismo Parametrico

> "In parametric polymorphism, a polymorphic function has an implicit or explicit type parameter which determines the type of the argument for each application of that function. [Cardelli-Wegner]"

Le entità che esibiscono un polimorfismo parametrico sono anche dette entità *generiche* o *templates*.

Una classe parametrica, offre la possibilità ottenere oggetti distinti solo per il tipo di certi attributi e metodi (ad es. stack di interi, reali, caratteri ...) senza dover scrivere codice ad hoc che le implementi.

Diversi linguaggi di programmazione presentano un polimorfismo parametrico (ad es. Ada, Eiffel e C++).

Templates – Polimorfismo Parametrico in C++

Il C++ prevede, tramite i *templates*, la possibilità di definire classi e funzioni parametriche.

Le classi parametriche descrivono la struttura delle singole classi, analogamente a come le classi descrivono la struttura degli oggetti.

Ad esempio la classe parametrica (su T) list è definita come segue:

```
template<class T> class list {
  T *info;
  list<T> *next;
 public:
  list(){...};
  T& Info( void ) {return info;};
       ...
```

Una classe parametrica per poter essere usata deve essere istanziata.

```
list<int> int_list;
list<char> char_list;
list<int_list> int_list_list;
```

Due nomi di classi parametriche riferiscono la stessa classe se i loro template sono identici ed i loro argomenti hanno valori identici.

In modo analogo si definiscono funzioni parametriche.

```
template<class T> void sort(list<T>);
```

Le funzioni parametriche definiscono una famiglia di funzioni simili, che possono essere usate passandogli i parametri appropriati.

```
int_list il;
sort(il);
```

Ereditarietà Multipla

Spesso le entità del mondo reale che si intende modellare con la metodologia object-oriented, hanno una natura tale che ben si prestono ad essere definite tramite ereditarietà multipla.

L'ereditarietà multipla permette di definire una classe combinando due o più classi.

Il principale problema che si incontra operando con più classi base è il fenomeno noto come *clash dei nomi*.

Una classe può ereditare da due classi diverse che definiscono un attributo od un metodo con lo stesso nome. L'utilizzo da parte degli eredi di tale nome risulta ambiguo. Ad esempio A e B definiscono entrambe il metodo x, C eredita sia da A che da B, l'espressione x all'interno di un metodo di C è ambigua.

Il clash dei nomi può essere dovuto al fatto che una classe derivata eredita più volte dalla stessa classe base. Ad esempio A definisce x, B eredita da A e C eredita sia da A che da B, l'espressione x in un metodo di C risulta ambigua.

Il problema del clash dei nomi può essere risolto in diversi modi: imponendo un ordinamento tra le classi base, definendo delle regole di visibilità o dei meccanismi per disambiguare esplicitamente le espressioni.

Ereditarietà Multipla in C++

La classe *polygon* può essere descritta in termini di un centro, il numero di lati e la lunghezza degli stessi

```
class polygon : virtual point {
  int vertex_number;
  int side_lenght;
 public:
  polygon(int vn, int sl, int x, int y);
  ~polygon(){};
  virtual void draw() {cout << "I'm a polygon\n";};
       ...
};

polygon::polygon(int vn, int sl, int x, int y) :
   point(x,y), vertex_number(vn), side_lenght(sl) {};
```

La classe *polygon_in_circle* che descrive i poligoni inscritti in un cerchio, può essere vista in termini di cerchi e poligoni

```
class polygon_in_circle : public circle, public polygon {
 public:
  polygon_in_circle(int vn,int sl,int x,int y,int radius);
  virtual void draw() {cout<<"I'm a polygon in a circle\n";};
};

polygon_in_circle::
  polygon_in_circle(int vn,int sl,int x,int y,int radius) :
    point(x,y), circle(x,y,radius),polygon(vn, sl, x, y){};
```

La classe *polygon_in_circle* è definita utilizzando l'ereditarietà multipla.

Ereditarietà Multipla in C++ (segue)

Il C++ offre due modi per disambiguare le espressioni ed evitare il clash dei nomi.

Il primo metodo consiste nel disambiguare esplicitamente le espressioni usando l'operatore di scope ::. Ad esempio la classe *polygon_in_circle* eredita due volte dalla classe *point*; quindi l'accesso al metodo X() risulta ambiguo e può essere disambiguato usando circle::X() o polygon::X().

Questa soluzione non è completamente soddisfacente, infatti un'istanza della classe *polygon_in_circle* presenta al suo interno due punti e non un solo punto come è lecito attendersi.

Il secondo metodo per evitare il clash sui nomi consiste nel fondere le classi ereditate più volte, in modo da non avere più ambiguità.

La fusione delle classi ereditate più volte è ottenuta ereditandole in modo virtuale, cioè introdotte dalla keyword virtual (come nell'esempio). Tramite la keyword virtual si informa il compilatore di inserire una sola volta la classe in questione tra le classi base della classe che si sta definendo. In questo caso si ha una sola istanza di *point* all'interno delle istanze di *polygon_in_circle* e l'espressione X() non è più ambigua.

Delegazione

> "*Delegation* is a mechanism that allows objects to delegate responsability for performing an operation or finding a value to one or more designated 'ancestors' [Wegner]."

La delegazione è definita indipendentemente dal concetto di classe e l'ereditarietà può essere vista come una specializzazione del concetto di delegazione.

Self è un linguaggio basato esclusivamente sulla delegazione (manca l'ereditarietà).

SmallTalk presenta, oltre all'ereditarietà, anche la delegazione. L'interpretazione di un messaggio inviato ad un oggetto, se non fa parte dell'interfaccia della classe di cui è istanza, è delegata alle sue antenate.

Clientship

> *Clientship* describes a relationship among objects.

La relazione di clientship, definisce le entità che possono usare le risorse definite in una classe senza necessariamente farne parte.

Esempi di linguaggi che implementano la clientship Eiffel e C++.

Friends – Clientship in C++

I *friend* di una classe sono funzioni che non sono membri della stessa, ma a cui è permesso usare i membri privati e protetti della classe.

Il nome del friend non fa parte dello scope della classe e non è attivata con i costrutti (. e ->) per l'accesso ai membri della classe.

```
class X {
  int a;
  friend void friend_set( X*, int );
 public:
  void member_set(int);
};

void friend_set( X* p, int i) { p->a = i };
void X::member_set(int i) {a=i;}
```

Dall'esempio si nota che la funzione friend_set può modificare gli attributi privati della classe di cui è friend. Inoltre quando si scrive l'implementazione non si deve usare il qualificatore di scope.

NB. Le funzioni friend non rompono l'encapsulation della classe in quanto fanno parte dell'interfaccia della stessa e ne sottostanno alle regole di visibilità e accesso.

Exception Handling

Exception handling è un meccanismo che permette di trasferire delle informazioni ed il flusso di controllo dell'esecuzione da un punto ad un'altro dell'esecuzione.
Il cambio di contesto è effettuato in seguito al verificarsi di un certo evento (generalmente un errore – eccezione) ed il punto dell'esecuzione a cui si salta è associato ad un gestore (handler) dell'eccezione.

In C++, le eccezioni vengono sollevate tramite l'istruzione throw.

```
throw "Overflow Error";
```

E vengono intercettate dalle istruzioni catch poste all'interno del *try-block* in cui è sollevata l'eccezione.

```
try{
   ...
}
catch (const char *p) {
  if (p=="Overflow Error") //gestione overflow
  else // gestione altre eccezioni
       ...
}
```

NB. Essendo catch una funzione è possibile definire più handler (grazie all'overloading) e raffinarli per gestire situazioni molto particolari.

Run-Time Type Information (RTTI)

RTTI è un meccanismo per gestire il tipo degli oggetti come un'informazione manipolabile durante l'esecuzione di un'applicazione.

RTTI del C++ si compone di tre parti:

- un operatore, dynamic_cast, usato per effettuare il cast dinamico da un puntatore ad un oggetto di classe base ad un puntatore ad un oggetto di classe derivata; l'operatore fornisce un valore diverso da 0 solo se il tipo dinamico del puntatore di cui si vuole fare il cast è effettivamente del tipo a cui si vuole fare il cast,
- Un operatore typeid, usato per identificare il tipo dinamico di un oggetto a partire dal puntatore ad un oggetto di classe base,
- Una struttura type_info, utilizzato come riferimento per ottenere ulteriori informazioni sul tipo a run-time.

Si consideri il solito esempio con i punti ed i cerchi, si hanno un puntatore ad un cerchio (`pc`) ed un puntatore ad un punto (`pp`).

`pc` può essere assegnato a `pp` tramite un'istruzione del tipo `pp = pc`, a questo punto espressioni del tipo:

```
pp->Radius();
```

sono errate anche se si è ben consci che `pp` punta ad un cerchio.

NB. Siccome `pp` ha tipo statico `point` non è possibile accedere al metodo `Radius` che fa parte della classe `circle`, cioè del suo tipo dinamico.

Run-Time Type Information (RTTI) (Segue)

Usando il meccanismo descritto è possibile scrivere codice che permetta di attivare i metodi del tipo dinamico derivato attraverso il puntatore ad un oggetto di tipo base.

```
if( circle *temp_circle = dynamic_cast<circle *>(pp) ) {
  // il cast ha avuto successo
  temp_circle->Radius();
} else // cast fallito non e' del tipo giusto
```

il test effettuato nell'if effettua il cast di `pp` al suo tipo dinamico `circle *`.

In alternativa è possibile realizzare codice analogo usando l'operatore di test del tipo dinamico typeid

```
if( typeid(circle) == typeid(*pp) ) {
  // il cast ha avuto successo
  circle *temp_circle = (circle *)pp;
  temp_circle->Radius();
} else // cast fallito non e' del tipo giusto
```

questa soluzione separa la fase di test da quella di cast introducendo la possibilità di commettere errori nella scelta del tipo su cui si effettua il test e di quello su cui si effettua il cast.

Input e Output in C++

In C++ le operazioni di input e output sono gestite da oggetti (cin e cout) speciali, detti *stream*.

Questi oggetti sono unici all'interno di tutto il programma e vengono creati prima della prima operazione di I/O e distrutti dopo l'ultima operazione di I/O.

Lo scheletro della classe di I/O è in linea generale:

```
class streamer {
  static count;
      ...
 public:
  streamer() { if ( count++ == 0 ) // initialize object };
  ~streamer() { if ( --count == 0 ) // destroy object };
           ...
};
```

e gli streamer standard sono definiti come segue:

```
 static streamer cin, cout;
```

Le operazioni di input e output avvengono tramite particolari operatori, >> per l'input e << per l'output. Questi due operatori sono overloaded per ogni tipo base del C++ (caratteri, interi, reali, ...) e devono essere definiti come friend (ed ovviamente overloaded) di qualunque classe che intenda fornire operazioni di input ed output per le proprie istanze.

Ansi C vs C++

Nonostante le analogie, non è detto che un programma C sia compilato correttamente da un compilatore C++ e funzioni come ci si attende.

Principalmente il C ed il C++ differiscono nei seguenti casi:

- in C *sizeof('a')* ha lo stesso valore di *sizeof(int)*, mentre in C++ ha il valore di *sizeof(char)*, che non è necessariamente uguale a *sizeof(int)*;
- in C++ il nome di una struttura dichiarata in uno scope interno, può nascondere il nome di un oggetto, di una funzione o di una funzione dichiarata nello scope esterno, ad es.

  ```
  int x[99];
  int f() { struct x {int a;}; return sizeof(x); };
  ```

  un'attivazione di f ritorna la dimensione della struttura x e non la dimensione dell'omonimo array;
- le funzioni devono essere dichiarate prima di poter essere usate;
- in C++, i nomi dichiarati all'interno di una struttura sono locali alla stessa, in C no;
- i caratteri costanti in C++ hanno tipo *char* in C hanno tipo *int*;

Inoltre esistono differenze di natura implementativa o dipendenti dal compilatore.

Il Crivello di Eratostene in C++

```
#include <iostream.h>

class element {
 public:
  element *source;
  element( element *src ) : source(src) {};
  virtual int emit( void ) {return 0;};
};

class counter : public element {
  int value;
 public:
  int emit( void ) {return value++;};
  counter( int v ) : element(0), value(v) {};
};

class sieve : public element {
 public:
  int emit( void );
  sieve( element *src ) : element(src) {};
};

int sieve::emit( void ) {
 int n = source->emit();
 source = new filter( source, n );
 return n;
}
```

Il Crivello di Eratostene in C++ (segue)

```
class filter : public element {
  int factor;
 public:
  int emit( void );
  filter( element *src, int f ) :
        element(src) { factor = f; };
};


int filter::emit( void ) {
  while(1) {
    int n = source->emit();
    if ( n % factor ) return n;
  };
}

void main() {
  counter c(2);
  sieve s(&c);
  int next;

  do {
    next = s.emit();
    cout << next  << ' ';
  } while( next < 101 );
  cout << '\n';
}
```

Liste Generiche in C++

Definizione Classe List

```
#include <iostream.h>

template <class T>
 class list {
   T info;
   list<T> *next;
  public:
   list( T elem ) : info(elem), next(0) {};
   virtual ~list(void);
   T Info() { return this->info; };
   virtual list<T> *Next(){return this->next;};
   virtual list<T> *add( T elem );
   int is_last() {return this->next == 0;};
   virtual list<T> *operator+ (list<T> b);
 };
```

Liste Generiche in C++ (Segue)

Implementazione Operazioni su Liste

```
template <class T>
 list<T> *list<T>::operator+( list<T> b ) {
  list<T> *aux_list, *root;
  aux_list = root = new list<T>(this->info);
  while( this->next != 0 ) {
   this = this->next;
   root->next = new list<T>( this->info );
   root = root->next;
  }
  root->next = new list<T>( b.Info() );
  while( b.Next() != 0 ) {
   b = *b.Next();
   root = root->next;
   root->next = new list<T>( b.Info() );
  }
  return aux_list;
 };

template <class T>
 list<T> *list<T>::add( T elem ) {
  list<T> *aux = new list<T>(elem);
   aux->next = this;
   return aux;
 }
```

Liste Generiche in C++ (Segue)

Overloading <<

```
template <class T>
 ostream & operator << (ostream &s, list<T> &l) {
  list<T> aux_list = l;
  s << "( ";
  do {
    s << aux_list.Info() << " ";
     aux_list = *(aux_list.Next());
  } while ( !aux_list.is_last() );
  s << aux_list.Info() << " )";
  return s;
 }
```

Liste Generiche in C++ (Segue)

Programma Principale

```
void main( void ) {
list<int> *il = new list<int>(1);
list<char> *cl_name = new list<char>('r');
cl_name = cl_name->add('e');
cl_name = cl_name->add('t');
cl_name = cl_name->add('l');
cl_name = cl_name->add('a');
cl_name = cl_name->add('W');
list<char> *cl_surname = new list<char>('a');
cl_surname = cl_surname->add('l');
cl_surname = cl_surname->add('o');
cl_surname = cl_surname->add('z');
cl_surname = cl_surname->add('z');
cl_surname = cl_surname->add('a');
cl_surname = cl_surname->add('C');
il = il->add(3);
il = il->add(7);
il = il->add(5);
il = il->add(25);
il = il->add(-1);
il = il->add(22);
cout  << *il << '\n' << *cl_name << ' ' << *cl_surname <<'\n';
cl_surname = *cl_name + *cl_surname;
cout << *cl_surname <<'\n';
}
```

Output

```
( 22 -1 25 5 7 3 1 )
( W a l t e r ) ( C a z z o l a )
( W a l t e r C a z z o l a )
```

Liste Generiche e Stack in C++ (Segue)

Classe Derivata Stack

```
template <class T>
 class stack : private lists<T> {
  public:
   stack() : list(), empty(TRUE) {};
   ~stack() {};
   stack<T> *push(T elem){return this->add(elem);};
   T pop(void);
   T top(void) {return this->info;};
   int is_empty();
 };

template <class T>
 T stack<T>::pop(void) {
   try{
     if (this->is_empty())
          throw "dangerous pop on empty stack";
     // codice dell'operazione di pop
                  ...
   } catch (const char *msg) {
     // azioni di recupero
     if (msg == "dangerous pop on empty stack")
         cout << "operazione illecita sullo stack vuoto\n";
     else  ...
   };
 };
```

Valutatore di Espressioni

Classe Astratta per le espressioni

```
virtual class skeleton_expr {
  public:
    skeleton_expr() {};
    virtual ~skeleton_expr() {};
    virtual int eval( void ) = 0;
};
```

Valutatore di Espressioni

Classe per espressioni zerarie, unarie e binarie

```
class zerary_expr : public skeleton_expr {
  int constant_value;
 public:
  zerary_expr(int value) : constant_value(value) {};
  ~zerary_expr() {};
  int eval( void ) { return this->constant_value; };
};

virtual class unary_expr : public skeleton_expr {
  char operator_symbol;
  skeleton_expr *sub_expr;
 public:
  unary_expr(char op, skeleton_expr *se1 )
     : operator_symbol(op), sub_expr(se1) {};
  skeleton_expr *operand( void ) {return this->sub_expr;};
  virtual ~unary_expr() {};
  virtual int eval( void ) = 0;
};

virtual class binary_expr : public unary_expr {
  skeleton_expr *sub_expr2;
 public:
  binary_expr(char op, skeleton_expr *se1, skeleton_expr *se2)
     : unary_expr(op,se1), sub_expr2(se2){};
  virtual ~binary_expr() {};
  skeleton_expr *left_operand(void) {return unary_expr::operand();};
  skeleton_expr *right_operand(void) {return this->sub_expr2;};
  virtual int eval( void ) = 0;
};
```

Valutatore di Espressioni(Segue)

Esempi di Classi Concrete Derivate

```
class minus_expr : public unary_expr {
 public:
  minus_expr(skeleton_expr *e) : unary_expr('-',e) {};
  ~minus_expr() {};
  int eval(void) {return - operand()->eval();};
};

class add_expr : public binary_expr {
 public:
  add_expr(skeleton_expr *e1, skeleton_expr *e2)
      : binary_expr('+', e1, e2){};
  ~add_expr(){};
  int eval(void)
      {return left_operand()->eval() + right_operand()->eval();};
};

class star_expr : public binary_expr {
 public:
  star_expr(skeleton_expr *e1, skeleton_expr *e2)
      : binary_expr('*', e1, e2){};
  ~star_expr(){};
  int eval(void)
      {return left_operand()->eval() * right_operand()->eval();};
};
```

Valutatore di Espressioni(Segue)

Un possibile Main

```
typedef skeleton_expr *expr;

void main(void) {
  expr e;
  char s[11] = "+2*-3*+567";
  cin >> e;
  cout << e << " = " << e->eval() << '\n';
}
```

Output

```
2+-3*((5+6)*7) = -229
```