IL PARADIGMA A OGGETTI

A livello di programmazione:

in-the-large ⇨ **modularità**
in-the-small ⇨ **astrazione**
emergere di ambienti distribuiti ed eterogenei ⇨ **incapsulamento**

CRISI DIMENSIONALE E GESTIONALE del software
⇨ necessità di una ulteriore evoluzione

Da aree di interesse diverse (quali i Linguaggi di Programmazione e l'Intelligenza Artificiale), è nata l'esigenza di una ***metodologia*** di strutturazione dei programmi che si basi sul ***riutilizzo*** delle informazioni

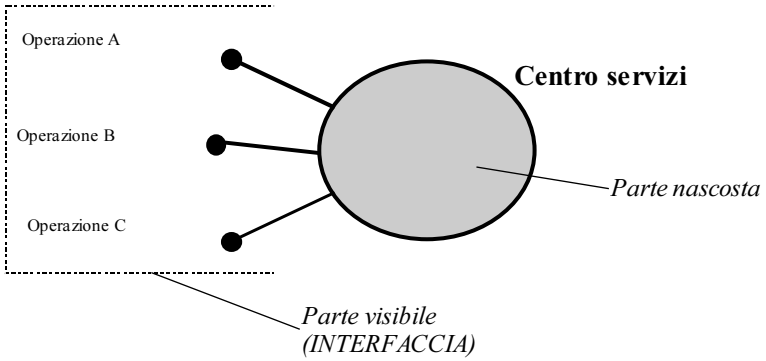
CONCETTO DI OGGETTO

Sulla base di questo paradigma sono stati sviluppati diversi **MODELLI** e quindi diversi linguaggi di programmazione:

- Smalltalk (capostipite)
- Eiffel
- C++
- Objective C
- Java
- etc.

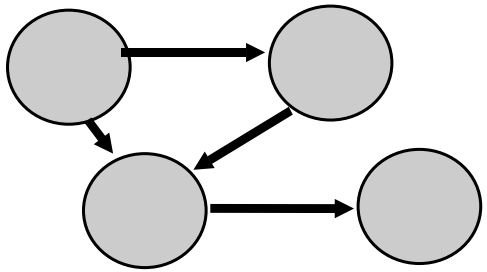
IL CONCETTO DI OGGETTO

Un ***centro di servizi***
con una *parte visibile* → *INTERFACCIA*
e una *parte nascosta*



ARCHITETTURA DI UN SISTEMA A OGGETTI

Un ***insieme di oggetti***
che **interagiscono gli uni con gli altri**
senza conoscere nulla delle rispettive implementazioni concrete



- un insieme di **oggetti** che si scambiano **messaggi**

DESCRIZIONE DI UN SISTEMA A OGGETTI

Ogni oggetto appartiene a ("è istanza di") una data ***classe*** (classe come "stampo" per la istanziazione di oggetti)

La classe *racchiude e incapsula la specifica* di
- **struttura** dell'oggetto (***dati***)
- **comportamento** dell'oggetto (***operazioni***)

Classi diverse possono essere correlate tra loro
→ *tassonomie* di ***ereditarietà***

IL CONCETTO DI CLASSE

Una classe, dunque, descrive le proprietà di un insieme di "oggetti" aventi:
- la medesima *struttura interna*
- lo stesso *protocollo di accesso* (insieme di operazioni)
- lo stesso *comportamento*

Riunisce le proprietà di:
- ***tipi di dato astratto***
  → funge da "stampo" per *creare* nuove *istanze* di oggetti
- ***moduli***
  → riunisce dati e relative operazioni
  → fornisce meccanismi di protezione

OBIETTIVI
- supporto a progettazione e sviluppo *incrementali*
- facilitazione di costruzione *cooperativa* di software
- possibilità di *rapida prototipazione e riuso*

PROPRIETÀ FONDAMENTALI DEL PARADIGMA A OGGETTI

- **ASTRAZIONE**
  OGGETTO:
  - contenitore informazioni
  - interfaccia (operazioni)
- **CLASSIFICAZIONE**
  - livello di META-descrizione
    relazione classe-istanza
  - entità descrittive capaci di racchiudere comportamenti comuni (a tempo di esecuzione)
- **EREDITARIETÀ**
  - relazione fra classi
  - riutilizzo di comportamenti comuni
- **COMUNICAZIONE**
  - tramite scambio di messaggi
    oggetti come ambienti locali
- **DINAMICITÀ**
  - creazione nuovi oggetti
  - valutazione a run-time della identità degli oggetti
  - variazioni di comportamento
- **POLIMORFISMO**
  - a fronte dello stesso codice, il comportamento che si ottiene, in istanti diversi, è diverso
- **CONCORRENZA E DISTRIBUZIONE**
  - oggetti come unità di esecuzione e di allocazione

CONCETTI DI BASE DEL PARADIGMA A OGGETTI

❋ Un **OGGETTO** rappresenta
l'entità indivisibile di strutturazione e
ha una identità precisa e permanente

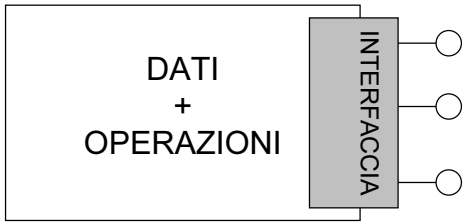
❋ Un oggetto raggruppa al proprio interno
- le informazioni (lo **STATO**), che gestisce mediante
- **OPERAZIONI** opportunamente definite,
le uniche in grado di agire sullo stato

NOTA: In questo senso, quindi il ***concetto di oggetto*** può essere assimilato a quello di ***astrazione di dato***

❋ Lo stato è ***incapsulato*** e ***protetto***,
poiché solo una **INTERFACCIA**, costituita dai nomi delle operazioni invocabili (il protocollo di comunicazione), è visibile dall'esterno

❋ Un oggetto è l'unico ***gestore*** autorizzato a operare sulle informazioni che esso contiene

☺ Il paradigma a oggetti propone una nuova **METODOLOGIA DI SVILUPPO DEL SOFTWARE**
⇨ DATI + OPERAZIONI incapsulati in un'unica entità



❋ In generale, nei modelli a oggetti, un oggetto non ha un ***nome***, ma uno o più PUNTATORI che lo referenziano
⇨ **SEMANTICA PER RIFERIMENTO**

Semantica per riferimento applicata alle variabili:
una variabile non contiene un oggetto, ma solo un riferimento a esso

Tale riferimento potrà essere usato per richiedere le operazioni su tale oggetto o per passare la conoscenza di tale oggetto ad altri oggetti

LO STATO

❋ Le variabili che compongono lo stato di un oggetto costituiscono la base di conoscenza dell'oggetto
Queste variabili contengono:

RIFERIMENTI ad altri oggetti
VALORI primitivi (ad esempio, numeri e caratteri)

Esempio: un OGGETTO O_1 ha

- STATO ⇨ due variabili
x che referenzia un altro oggetto O_2
y che contiene il valore primitivo 642

- OPERAZIONI ⇨ due operazioni
M_1 e M_2 che usano x e y

VARIAZIONE DEL PUNTO DI VISTA

Linguaggi procedurali (come C o Pascal)

⇨ punto di vista funzionale
operazione (oggetto, altri parametri)

ESEMPIO:
Enqueue(lista1, ch1);

Linguaggi a oggetti (come Smalltalk o C++)

⇨ punto di vista a oggetti
oggetto operazione(altri parametri)

L'oggetto riceve la richiesta di eseguire l'operazione (che fa parte della sua interfaccia): l'operazione fa uso della conoscenza dello STATO dell'oggetto e degli eventuali parametri

- Gli oggetti sono i *soggetti* cui vengono richieste le operazioni

ESEMPIO:
lista1 Enqueue (ch1)

- Inoltre, gli oggetti sono entità **FIRST-CLASS** cioè possono essere passati come parametri (di ingresso e di uscita) nelle operazioni

ESEMPIO:
lista2 Enqueue (lista1)

LA COMUNICAZIONE FRA GLI OGGETTI

IN UN MODELLO A OGGETTI, LA COMPUTAZIONE È IL RISULTATO DELL'INSIEME DI ATTIVITÀ CHE SI STABILISCONO NEGLI OGGETTI

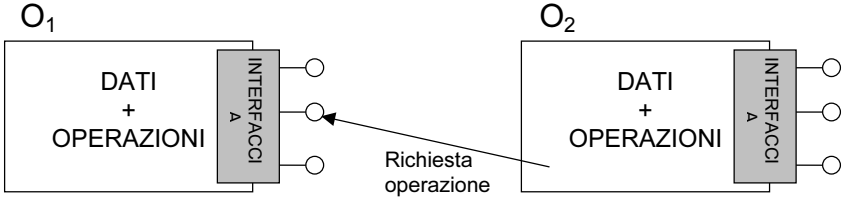
Un oggetto O_1 che abbia bisogno delle informazioni contenute in un altro oggetto O_2 deve invocare una delle operazioni che sono accessibili dall'esterno (e che quindi fanno parte dell'*interfaccia* dell'oggetto)

- L'interazione tra oggetti implica una richiesta di operazione di un oggetto da parte di un altro oggetto
  ⇨ Il meccanismo di interazione è disciplinato
- Date le caratteristiche di località insite nel concetto di oggetto, il meccanismo su cui si basa la INTERAZIONE è la **COMUNICAZIONE** mediante **SCAMBIO DI MESSAGGI**

UNA COMUNICAZIONE TRA OGGETTI IMPLICA LO STABILIRSI DI UNA RELAZIONE MITTENTE/RICEVENTE - CLIENTE/SERVITORE

- Il mittente (cliente) specifica l'identità del ricevente (servitore), usando una variabile, e l'operazione da eseguire
  Ad esempio: *ricevente operazione (lista-parametri)*

- Il ricevente decide quando e come servire la richiesta; al completamento della corrispondente operazione invia una risposta (risultato)



- La relazione cliente/servitore è una relazione ***dinamica***: in un istante, O_1 è il cliente del servitore O_2, in un altro istante, O_2 può diventare il cliente del servitore O_1

- ***L'identità*** dell'oggetto che riceve la richiesta di una operazione viene determinata a ***tempo di esecuzione***

  ⌦ Infatti, in generale, il riferimento al ricevente di un messaggio è il valore di una variabile

  ⇨ altro aspetto di DINAMICITÀ

DYNAMIC BINDING: l'oggetto cui si richiede una certa operazione viene calcolato dinamicamente
⇨ dipende dal valore attuale della variabile usata per effettuare la richiesta

CASO PARTICOLARE DI SCAMBIO DI MESSAGGI

Un oggetto può all'interno di una operazione richiedere una sua operazione (anche la stessa ➔ ricorsione)
⇨ per uniformità, l'oggetto deve mandare un messaggio a se stesso
⌦ **pseudo-variabile SELF**

RIUTILIZZO DELLE INFORMAZIONI

Insiemi di oggetti possono condividere gli stessi comportamenti
⇨ è possibile non dovere rispecificare il comportamento comune grazie ai concetti di **CLASSIFICAZIONE** ed **EREDITARIETÀ**

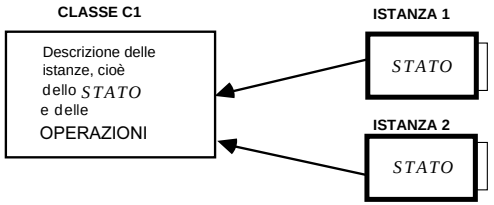
CONCETTO DI CLASSE

- **IL PRINCIPIO DI CLASSIFICAZIONE**
  si definisce una entità descrittiva, la CLASSE, che raggruppa tutte le definizioni (dello stato e delle operazioni) che caratterizzano un insieme di oggetti

NOTA: In questo senso, il ***concetto di classe*** può essere assimilato a quello di ***tipo di dato astratto*** poiché descrive le istanze da essa generate: nel caso di linguaggi a oggetti con variabili dotate di tipo, il nome di una classe diventa un **tipo** utilizzabile per definire variabili

- Ogni oggetto (**ISTANZA**) creato da una classe C_1 avrà uno stato proprio la cui rappresentazione e le cui operazioni sono descritte nella classe C_1

- La relazione classe-istanza perdura per tutto il tempo di vita della istanza



DINAMICITÀ AMBIENTE

La presenza a tempo di esecuzione delle entità descrittive cioè delle classi, rappresenta la chiave della DINAMICITÀ degli ambienti a oggetti

Intesa:
A) sia come crescita incrementale, cioè creazione di ***nuovi oggetti*** (comprese ***nuove classi***, specializzazione di classi preesistenti);
B) sia come ***variazioni***, cioè ***modifiche dei comportamenti*** descritti in una classe e quindi delle istanze

PROBLEMA:

LE MODIFICHE DI UNA CLASSE POSSONO PRODURRE INCONSISTENZE:

Un cambiamento in una classe può ripercuotersi sulla struttura delle istanze: ad esempio se vengono aggiunte/cancellate nella descrizione delle variabili di stato
Le istanze create prima della variazione risultano non più congruenti con la nuova descrizione

Possibili soluzioni:
1) modificare tutte le vecchie istanze ristabilendo il corretto legame con la classe variata
2) lasciare le vecchie istanze collegate alla vecchia struttura della classe e creare una nuova classe con la descrizione modificata (metodo delle versioni)
3) è possibile anche dichiarare obsolete e "pericolose" le vecchie istanze
4) se sono già state generate istanze da una classe (anche solo una) non consentire la modifica

DIFFERENZE FRA CONCETTO DI CLASSE E CONCETTO DI TIPO TRADIZIONALE

LINGUAGGI TIPATI

- Il TIPO rappresenta l'insieme dei valori che possono essere attribuiti a una variabile

Ad esempio:

TIPI: **char** ⇨ insieme dei caratteri
int ⇨ insieme dei valori interi che una variabile può assumere

VARIABILI: **char** c;
int x;

- È possibile determinare staticamente il tipo di una variabile ed eseguire dei ***controlli statici di correttezza***

  ⇨ Ogni operazione può essere applicata solo su una variabile di tipo corretto: il dato contenuto nella variabile subisce passivamente l'operazione

MONDO a OGGETTI

- L'oggetto è il solo responsabile della propria rappresentazione e delle operazioni

  - l'oggetto è l'unico a determinare il proprio stato
    e
  - ad accettare le operazioni

DIFFERENZE fra CLASSE e TIPO:

a) Ogni istanza ha la classe che la descrive come sua proprietà intrinseca e tale proprietà viene rilevata a tempo di esecuzione

b) L'informazione di natura descrittiva contenuta in una classe permane a tempo di esecuzione garantendo la dinamicità, mentre nei linguaggi tradizionali l'informazione di tipo viene persa nelle prime fasi dello sviluppo (tipicamente a tempo di compilazione)

Quindi, come abbiamo detto si ha una variazione del punto di vista cioè si ha un NUOVO MODELLO DI COMPUTAZIONE

⇨ **LE OPERAZIONI NON SONO FORZATE**
***MA* SONO ACCETTATE DAI DATI (OGGETTI)**

Inoltre, i primi linguaggi a oggetti (ad esempio, Smalltalk) tendono a **non** evidenziare l'aspetto di **controllo statico di tipo**

in favore di una

maggiore **dinamicità** del sistema, che NON associa un tipo alle variabili, ma lo verifica dinamicamente sulla base della classe dell'oggetto cui si fa riferimento

⇨ **POLIMORFISMO**
Si parla di polimorfismo se, a fronte dello stesso codice, il comportamento che si ottiene, in istanti diversi, è diverso

POLIMORFISMO ORIZZONTALE (per linguaggi a oggetti NON TIPATI)

Questo tipo di POLIMORFISMO è presente solo nei sistemi che non introducono la definizione di tipo per le variabili
⇨ ***LINGUAGGI NON TIPATI***

In questo caso, LE VARIABILI POSSONO CONTENERE RIFERIMENTI A OGGETTI APPARTENENTI A OGNI CLASSE PRESENTE NELL'AMBIENTE

Quindi,
UNA OPERAZIONE CON LO STESSO NOME PUÒ ESSERE DIRETTA CORRETTAMENTE SU OGGETTI DIVERSI

Ad esempio:

```
v1 print
```

L'operazione `print` può essere diversa a seconda della classe dell'oggetto riferito da `v1`, dato che ***in tempi diversi, v1 può riferire oggetti diversi***

a) se `v1` riferisce una istanza della classe CERCHIO in risposta a questo messaggio verrà eseguita l'operazione specificata in tale classe

b) se `v1` riferisce una istanza della classe AUTOMOBILE in risposta a questo messaggio verrà eseguita l'operazione specificata in tale classe, che sarà quindi sicuramente diversa da quella eseguita nel caso a)

EREDITARIETÀ

- L'ereditarietà è un ulteriore meccanismo base per il ***riutilizzo*** del software, ortogonale alla suddivisione in livelli descrittivi (oggetto-classe-etc.)

IL PRINCIPIO DI FATTORIZZAZIONE:

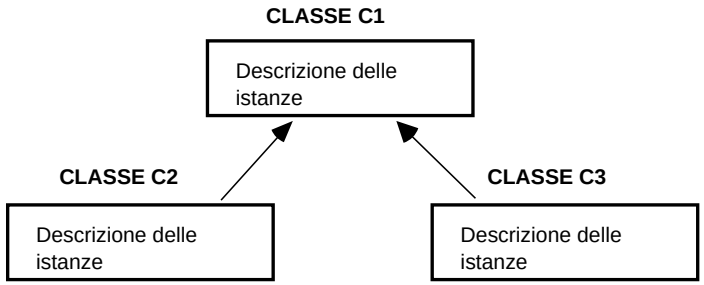
Nello specificare una nuova entità descrittiva, si può derivare da un'altra entità descrittiva parte della sua specifica

⌦ si applica quindi a ogni livello descrittivo (classe, etc.)

- In genere, il livello cui si fa riferimento quando si parla di ereditarietà è quello delle classi:
  una classe deriva da altre classi (superclassi), già definite, parte dei comportamenti delle sue istanze

L'ereditarietà introduce una relazione che correla le classi una all'altra

Esempio di ereditarietà:
la classe STUDENTI e la classe LAVORATORI ereditano dalla classe PERSONE



NOTE:
1) La relazione di ereditarietà è specifica degli ambienti a oggetti e non è presente nelle metodologie basate sul concetto di astrazione
2) L'ereditarietà fra classi introduce un'ulteriore differenza fra TIPO e CLASSE

L'ereditarietà interessa tutto ciò che una classe specifica per le proprie istanze: in particolare, sia la descrizione dello stato che le operazioni

⇨ grazie alla ereditarietà, un'istanza creata da una classe C ha tutte le variabili descritte nelle superclassi di C e può eseguire ogni operazione definita nelle superclassi di C

- Questo è vero a meno di cancellazioni dovute a **NON MONOTONICITÀ**

  ESEMPIO di non monotonicità: classe PINGUINO eredita da classe UCCELLO, ma elimina *volare*

Una classe C1 oltre che ereditare dalle superclassi variabili e operazioni, in genere, specifica ulteriori variabili e operazioni specifiche

- In particolare, un'operazione ereditata può essere modificata mediante sovrascrittura (**OVERRIDING**)
  - ⇨ in genere, questo serve per adattare la definizione di un'operazione alla classe specifica
  - ⇨ ***in genere, si può sempre riutilizzare anche l'operazione sovrascritta***

La relazione di ereditarietà deve stabilire la **strategia di ricerca** delle definizioni

Questa strategia viene applicata sia al momento della creazione di un'istanza per sapere il contenuto dello STATO, sia al momento della determinazione del codice associato a un'operazione

⌦ tale strategia dipende, in primo luogo, dal ***tipo*** di ereditarietà

❋ L'ereditarietà può essere:

1. **SINGOLA** ⇨ una classe deriva da una sola SUPERCLASSE
2. **MULTIPLA** ⇨ una classe deriva da più SUPERCLASSI (ad esempio, la classe MULO può derivare dalla classe CAVALLO e dalla classe ASINO grazie all'ereditarietà multipla)

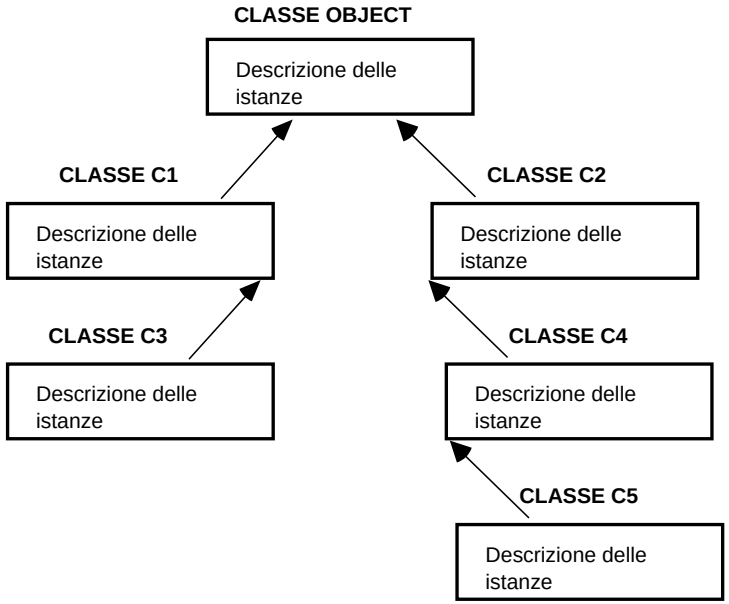
❋ In genere, un ambiente a oggetti definisce una **classe TERMINALE** che raggruppa al suo interno le definizioni comuni a ogni oggetto e dalla quale hanno origine tutte le classi ⇨ **RADICE** DELL'ALBERO o DEL GRAFO DELLE CLASSI

Ad esempio, in Java la radice dell'albero delle classi è la classe OBJECT

EREDITARIETÀ a SINGOLO GENITORE

ALBERO DI CLASSI

La ricerca dei comportamenti, nel caso di ereditarietà singola, procede nella catena di superclassi fino alla classe terminale (se esiste)



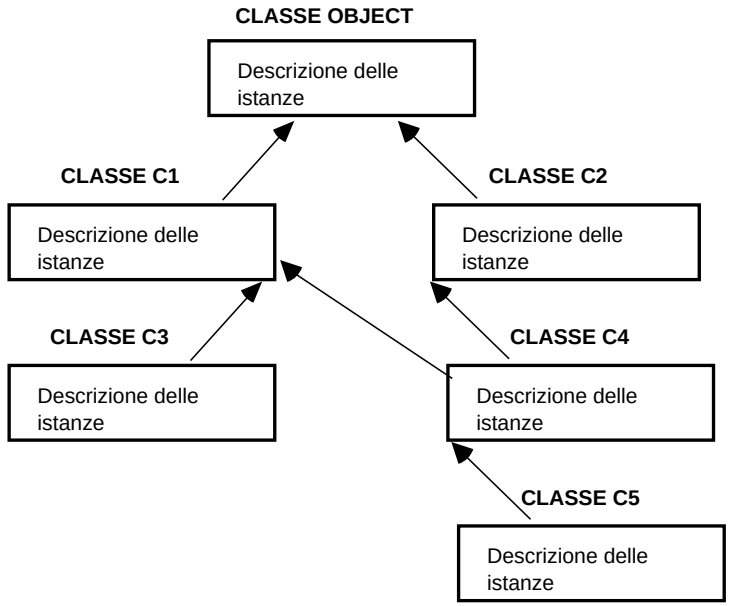
Per la classe C5, la ricerca è in C5, C4, C2, OBJECT

Quindi, le istanze della classe C5:

- possiedono lo stato descritto in C5, quello descritto in C4, quello descritto in C2, e quello descritto in OBJECT;
- e ad esse possono essere richieste tutte le operazioni descritte in C5, quelle descritte in C4, quelle descritte in C2, e quelle descritte in OBJECT (a meno di overriding)

EREDITARIETÀ MULTIPLA

GRAFO DI CLASSI



La CLASSE C4 eredita in modo multiplo dalle classi C1 e C2 ⇒ occorre definire una politica di ricerca

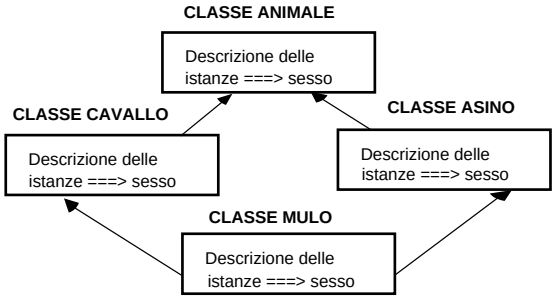
Per ogni istanza si determina un *albero* di classi che ha come radice la classe di appartenenza

Ad esempio: per una istanza della CLASSE C5, abbiamo un albero con nodi C5, C4, C1, C2 e OBJECT

La politica di ricerca più usata è quella **depth-first** *con ordine dei rami fissato*, lasciando per ultima la classe terminale

ESEMPIO: da C4, C1 prima e poi C2

ESEMPIO 1: supponiamo di avere una classe ANIMALE che definisce una variabile di stato `sesso`; da questa ereditiamo due classi, CAVALLO e ASINO. Quindi, tramite ereditarietà multipla, definiamo una classe MULO: in questo caso, si deve avere solo una copia della variabile `sesso` definita in ANIMALE anche se la classe ANIMALE viene incontrata ("visitata") esplorando sia la catena che parte da CAVALLO, sia quella che parte da ASINO



Ma è veramente sempre un problema oppure in alcuni casi va bene che le informazioni siano ripetute?

ESEMPIO 2: supponiamo di avere una classe BARCA che definisce una variabile di stato `patente`; da questa ereditiamo due classi, BARCA_VELA e BARCA_MOTORE. Quindi, tramite ereditarietà multipla, definiamo una classe MOTOVELIERO: in questo caso, si devono avere due copie separate della variabile `patente` poiché per guidare un motoveliero è necessario avere sia la patente per le barche a vela che quella per le barche a motore

OPERATION DISPATCHING

Nel momento in cui l'oggetto servitore è stato identificato, bisogna selezionare l'operazione richiesta (***operation dispatching***)

Quando si richiede ad un oggetto O_1 di svolgere una operazione, il codice da eseguire è stato specificato nella classe di O_1 ⇨ meccanismo di interazione automatico grazie alla relazione classe-istanza

LINGUAGGI NON TIPATI

L'associazione fra l'oggetto servitore e l'operazione da eseguire viene risolta a tempo di esecuzione effettuando una ricerca del codice da mettere in esecuzione nella classe e, nel caso, nelle sue superclassi

SVANTAGGIO: tale ricerca può anche fallire se non si trova nessuna operazione che risponda alla specifica richiesta ⇨ la ricerca si conclude quando si arriva alla classe terminale

LINGUAGGI TIPATI

CONFORMITÀ ⇨ si verifica staticamente che l'operazione richiesta esista: essa deve appartenere alla interfaccia dell'oggetto servitore

Su una variabile di tipo-classe C possono essere richieste solo le operazioni definite in C e nelle sue superclassi ⇨ l'operazione dipenderà dal tipo statico della variabile tramite cui si identifica l'oggetto servitore

Collegamento statico (cioè a tempo di sviluppo):
quali problemi?
Vedi il caso di operazioni sovrascritte (overriding)

POLIMORFISMO VERTICALE (per linguaggi ad oggetti TIPATI)

Tale collegamento statico, però, non può essere effettuato sempre: infatti, in presenza di **overriding** di operazioni, il codice da mettere in esecuzione deve essere l'ultimo definito nella gerarchia di ereditarietà e non quello selezionato staticamente

Ad esempio:

```
v1: POLIGONO
...
v1 perimetro
```

⌦ richiediamo all'oggetto riferito attraverso `v1`, l'operazione `perimetro`

L'operazione `perimetro` può essere diversa a seconda della classe dinamica dell'oggetto riferito da `v1`, dato che in tempi diversi, `v1` può riferire oggetti conformi diversi

a) se `v1` riferisce una istanza della classe POLIGONO in risposta a questo messaggio verrà eseguita l'operazione specificata in tale classe

b) se `v1` riferisce una istanza della classe RETTANGOLO in risposta a questo messaggio verrà eseguita l'operazione specificata in tale classe, che è diversa da quella eseguita nel caso a) nel caso di overriding

Questo polimorfismo viene detto ***verticale*** (per distinguerlo da quello orizzontale) poiché deriva dalla relazione di ereditarietà

GARBAGE COLLECTOR

Nei sistemi ad oggetti "puri", la creazione di un oggetto è sempre una operazione richiesta ad una classe

⇨ ***Tale richiesta è fatta dinamicamente e in modo esplicito***

Per quanto riguarda invece la distruzione degli oggetti i sistemi ad oggetti "puri" non prevedono nessuna operazione esplicita

⇨ la ragione di ciò è data dal volere evitare problemi di **dangling reference**

NOTA: in questo caso, la richiesta di distruzione dovrebbe essere una operazione richiesta all'oggetto/istanza che si vuole distruggere

ESEMPIO: Supponiamo che sia l'oggetto O_1 che l'oggetto O_2 riferiscano l'oggetto O_3
Se l'oggetto O_1 decidesse ad un certo istante di distruggere l'oggetto O_3, l'oggetto O_2 si troverebbe con un riferimento per un oggetto non più esistente

Per evitare questo tipo di problemi, si adotta una soluzione che fa uso di un **GARBAGE COLLECTOR** a ***livello di sistema*** (vedi Java)

Infatti, il GARBAGE COLLECTOR è uno strumento che si occupa di distruggere un oggetto ***se e solo se*** non esistono più riferimenti a quell'oggetto nel sistema
In questo modo, si è sicuri di non produrre mai problemi di dangling reference

CAPACITÀ COMPUTAZIONALI

I MODELLI AD OGGETTI POSSONO ESSERE:

PASSIVI

La capacità computazionale è fornita da processi (flussi di esecuzione) e gli oggetti sono entità passive

- Possibilità di avere un **solo processo** oppure di avere **più processi** che eseguono (multiprocessing)
- Gli oggetti PASSIVI sono solo capaci di rispondere a **sollecitazioni esterne**
- Gli oggetti passivi sono **"attraversati" dai processi**, che determinano la esecuzione al loro interno

ATTIVI

La capacità computazionale è associata ad ogni oggetto

- Gli oggetti ATTIVI sono dotati di capacità di **esecuzione autonoma**
- Gli oggetti attivi possono avere la capacità di **discriminare** fra le richieste pervenute e **decidere** quale servire in base al loro stato computazionale corrente
- A ogni oggetto deve essere associata una **coda dei messaggi** arrivati e non ancora serviti
- Possibilità di avere un solo flusso di esecuzione o **più flussi di esecuzione** associati allo stesso oggetto

DIVERSE MODALITÀ DI COMUNICAZIONE FRA OGGETTI (eventualmente distribuiti)

A) **Scambio di messaggi di tipo SINCRONO**
   ⇨ attesa da parte del mittente (cliente) della risposta

B) **Scambio di messaggi di tipo ASINCRONO**
   ⇨ il mittente (cliente) non aspetta la risposta che viene scartata

C) **Scambio di messaggi di tipo ASINCRONO CON RISPOSTA DIFFERITA**
   ⇨ come l'asincrono, ma il mittente (cliente) può, in un istante successivo, ritrovare la risposta

Quest'ultima modalità viene realizzata, solitamente, introducendo un'entità intermedia in cui viene memorizzata la risposta ritornata dal ricevente (servitore)