

Il paradigma Orientato agli Oggetti

Metodi Avanzati di Programmazione

Laurea Triennale in Informatica

Università degli Studi di Bari Aldo Moro

Docente: Pierpaolo Basile

Credits

- Prof.ssa Annalisa Appice
- Prof. Michelangelo Ceci
- Prof. Donato Malerba

Introduzione...

Un difetto fondamentale della programmazione imperativa è che le variabili globali sono potenzialmente accessibili da ogni parte del programma.

I grandi programmi che permettono l'accesso alle variabili globali tendono ad essere ingestibili. La ragione è che nessun modulo che accede ad una variabile globale può essere sviluppato e compreso indipendentemente da altri moduli che pure accedono alla medesima variabile.

Introduzione...

Agli inizi degli anni 70 questo problema fu riconosciuto da **David Parnas** che patrocinò la disciplina dell'*information hiding* come rimedio. La sua idea era quella di incapsulare in un modulo ogni variabile globale insieme a un gruppo di operazioni autorizzate ad accedervi. Gli altri moduli possono accedere alla variabile solo indirettamente, chiamando queste operazioni.

L'idea di Parnas era proprio quella di definire degli *oggetti*.

...Introduzione.

Si deve osservare che nella programmazione imperativa è possibile definire degli oggetti, tuttavia

1. Il loro utilizzo è legato all'auto disciplina dei programmatori e non viene in alcun modo forzato
2. Gli oggetti ***non sono cittadini di prima classe***

OO: Evoluzione o Rivoluzione?

Confrontando il paradigma orientato a oggetti rispetto a quello imperativo si può dire che esso costituisce:

- Una **evoluzione**, in quanto permette agli oggetti di essere cittadini di prima classe
- Una **rivoluzione**, in quanto gli oggetti assumono un ruolo fondamentale nella progettazione e nella programmazione. **Information hiding** + **incapsulamento** sono principi cardine nel paradigma orientato a oggetti

La classificazione di Wegner (1987)

I linguaggi di programmazione si classificano in:

- **Object-based**: supportano la nozione di oggetto (Modula-2)
- **Class-based**: supportano la nozione di oggetto e classe (Ada-83)
- **Object-oriented**: supportano la nozione di oggetto, classe, ereditarietà (Smalltalk, C++, Java, ...)

object-oriented = objects + classes + inheritance

Gli oggetti

Gli oggetti incapsulano uno **stato** e un **comportamento**.

- Lo stato è identificato dal contenuto di una certa area di memoria
- Il comportamento è definito da una collezione di procedure e funzioni (chiamate **metodi**) che possono operare sulla rappresentazione dell'area di memoria associata all'oggetto

Da una prospettiva di progetto, gli oggetti modellano le **entità** presenti nel dominio dell'applicazione.

Gli oggetti

Esempio: in un gioco elettronico che usa una palla, si può pensare alla palla come un oggetto dotato di uno

- **Stato**: dimensione, posizione in uno spazio di riferimento, etc.
- **Comportamento**: una palla può ‘apparire’ o ‘scompare’ dallo schermo, può muoversi, può rimbalzare, etc.

Le variabili e i metodi definiti nell’oggetto ‘palla’ stabiliscono lo stato e il comportamento che sono rilevanti all’uso della palla nel gioco elettronico.

Identificatore di oggetto

- Un oggetto ha la sua **identità**, cioè è riconoscibile indipendentemente dal suo stato corrente. Per questo ogni oggetto ha un **identificatore di oggetto** (*object identifier, OID*) che lo identifica univocamente. In alcuni contesti gli OID sono anche detti **riferimenti** (*references*)
- Un identificatore di oggetto è **immutabile**, cioè non può essere modificato da una qualche opzione di programmazione. Cambiare l'OID di un oggetto equivale alla cancellazione dell'oggetto e alla creazione di un altro oggetto con lo stesso stato

Identificatore di oggetto

- Normalmente gli OID sono assegnati in modo automatico agli oggetti, sicché non hanno un significato nel mondo reale
- In molti ambienti di programmazione object-oriented, l'OID corrisponde all'indirizzo dell'area di memoria che conserva lo stato dell'oggetto
- Quasi mai il programmatore utilizza esplicitamente i riferimenti. Generalmente questi vengono legati a delle variabili e si fa riferimento agli oggetti mediante gli identificatori di variabile
- Variabili distinte possono riferirsi al medesimo oggetto. In questo caso si hanno degli **alias**

Identificatore di oggetto

- N.B.: La presenza di alias non significa che un oggetto non è identificato univocamente, ma semplicemente che **diversi identificatori di variabile** sono stati legati al medesimo riferimento di oggetto
- Lo stato di un oggetto può anche contenere il riferimento ad un altro oggetto. Si dice che un oggetto **punta** ad un altro. Il puntamento è *asimmetrico*

UML

UML™ (*Unified Modeling Language*), un linguaggio visuale utilizzato per

- definire
- progettare
- realizzare
- documentare



sistemi (software) mediante un approccio object oriented

- È un linguaggio di **rappresentazione dei sistemi**
- È **universale**: può rappresentare sistemi eterogenei per architettura, tecnologie, tipologia applicativa (gestionale, real-time, ...)
- UML è di supporto
 - alla progettazione di un nuovo sistema
 - documentazione di un sistema esistente

UML

- Può essere utilizzato per la progettazione dei sistemi più disparati, dai sistemi Web a quelli più tradizionali
- Può essere utilizzato in molte fasi del ciclo di vita del SW
 - nella relazione cliente-fornitore
 - ingegnere-ingegnere
- senza perdersi nei dettagli di un linguaggio di programmazione

Modellazione Object-oriented

- Nella modellazione object-oriented le componenti elementari sono l'oggetto e la classe
 - Un oggetto è qualcosa tratta generalmente dal vocabolario dello spazio del problema o dello spazio della soluzione
 - Una classe è una descrizione di un insieme di oggetti omogenei
 - Ogni oggetto ha una identità, uno stato, e un comportamento
- Visualizzare, specificare, costruire, e documentare sistemi object-oriented è esattamente lo scopo dello UML

Cosa non è UML

- Non è una “metodologia” di sviluppo del software
 - Non indica alcuna decomposizione dell’attività di sviluppo del software in sottoattività
 - Né fornisce indicazioni sul proprio utilizzo in una metodologia
 - Ciò consente l’utilizzo di UML in differenti processi di sviluppo, quali sistemi a cascata, incrementale, a spirale, ecc...
 - L’utilizzo di UML va quindi accompagnato a una metodologia che ne specifichi le modalità di utilizzo
- Non è un “linguaggio di programmazione”
- Non è un linguaggio proprietario
 - non è legato né a uno specifico linguaggio di programmazione, né a uno specifico strumento
- Notazione, sintassi e semantica sono standard

Chi lo ha definito e promosso

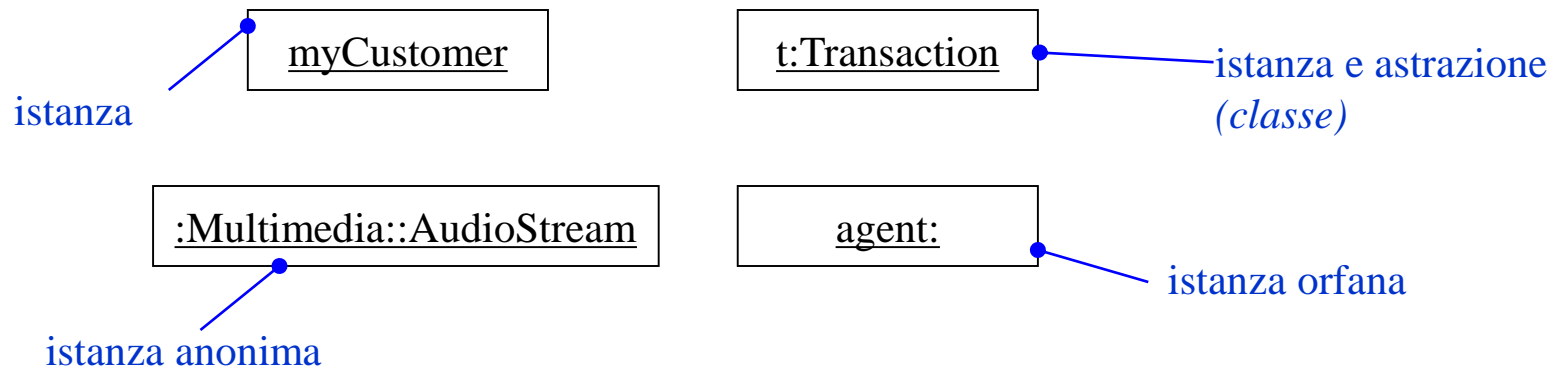
- Standard OMG (Object Management Group), dal novembre 1997
- Originatori:
 - Grady Booch
 - Ivar Jacobson
 - James Rumbaugh
- Riferimenti:
 - G.Booch, J. Rumbaugh, I. Jacobson
 - The Unified Modeling Language user guide*. Addison Wesley, 1999
 - documenti ufficiali: www.omg.org
 - altre risorse UML: www.uml.org

Notazione UML

- Adotteremo UML come **notazione universale per illustrare alcuni concetti del paradigma orientato a oggetti**, senza fare così riferimento a un particolare linguaggio di programmazione
- **Attenzione:** UML è ben più di un insieme di simboli grafici. Ogni simbolo in UML ha una semantica ben definita che ne permette l'univoca interpretazione fra diversi interlocutori e strumenti software

Oggetti (Istanze) in UML

In UML un oggetto (o **istanza**) è graficamente rappresentato in questo modo:

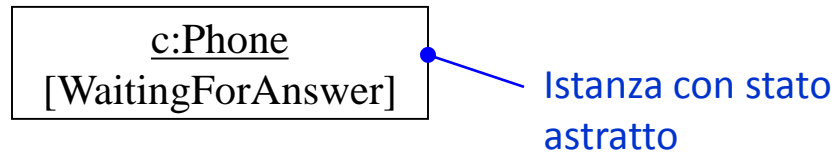


È possibile indicare:

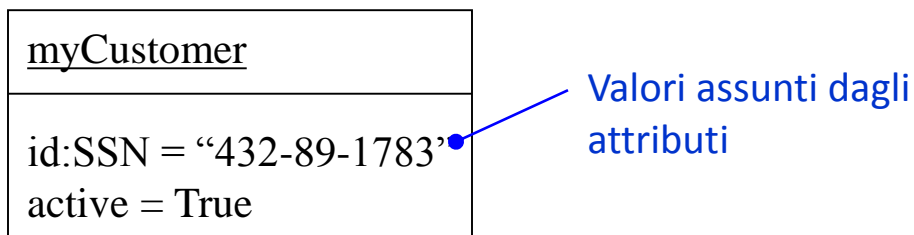
- solo il nome (identificatore) dell'istanza
- nome dell'istanza e della astrazione (la classe) cui appartiene
- solo il nome dell'astrazione, qualora non si conosca a priori il nome dell'istanza
- un'istanza orfana, se non si conosce a priori la sua astrazione

Oggetti in UML

Lo stato di un oggetto può essere rappresentato in modo astratto:



oppure indicando esplicitamente i valori assunti dagli attributi dell'oggetto



Le classi

Una **classe** è la descrizione di una famiglia di oggetti che condividono la stessa struttura (gli attributi) e il medesimo comportamento (operazioni)

Nella programmazione OO **ogni oggetto è un'istanza di una classe**, cioè *un oggetto non può essere ottenuto se non si definisce la sua classe di appartenenza*. Analogamente nella modellazione OO *le istanze esistono in quanto ci sono le loro astrazioni*

I dettagli della realizzazione di una classe sono normalmente nascosti.

Ogni classe ha una doppia componente:

Le classi

1. **Una componente statica**, i dati, costituita da *campi* o *attributi* dotati di nome, che contengono un valore. I campi caratterizzano lo stato degli oggetti durante l'esecuzione del programma.

Gli attributi si distinguono in base al loro ambito d'azione (**scope**):

- **Attributi d'istanza**: sono associati ad una istanza e hanno un tempo di vita pari a quello dell'istanza alla quale sono associati.
- **Attributi di classe**: sono associati alle classi e condivisi da tutte le istanze della classe. Il loro tempo di vita è lo stesso della classe.

Gli attributi di istanza contribuiscono a caratterizzare lo stato di ogni singolo oggetto, mentre gli attributi di classe contribuiscono a definire il fattore comune allo stato di tutti gli oggetti di una classe.

Le classi

2. **Una componente dinamica**, i metodi (o operazioni), che rappresentano il comportamento comune degli oggetti appartenenti alla classe, cioè i servizi che possono essere richiesti a un oggetto di una classe. I metodi manipolano gli attributi.

I metodi possono essere classificati:

1. **Metodi costruttori**: sono invocati per creare (istanziare) gli oggetti e inizializzarli
2. **Metodi di accesso**: restituiscono astrazioni significative dello stato di un oggetto
3. **Metodi di trasformazione**: modificano lo stato di un oggetto
4. **Metodi distruttori**: sono invocati quando si rimuovono gli oggetti dalla memoria

I metodi di accesso e trasformazione possono essere distinti in:

Le classi

1. **Metodi di istanza:** operano su almeno un attributo di istanza, pertanto possono essere invocati solo specificando l'istanza
2. **Metodi di classe:** operano esclusivamente su attributi di classe, pertanto possono essere invocati specificando la classe. Si possono invocare metodi di classe anche quando non è stato istanziato alcun oggetto per quella classe

In linea di principio, l'invocazione di un metodo di classe può avvenire anche specificando un oggetto (e non la classe), tuttavia ciò è sconsigliato perché non evidenzia il fatto che si manipolano solo attributi di classe

Non è possibile invocare un metodo di istanza sulla classe

Le classi: notazione UML

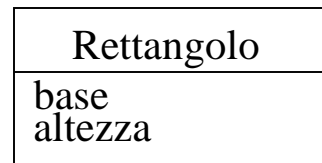
In UML una classe è resa graficamente mediante un rettangolo



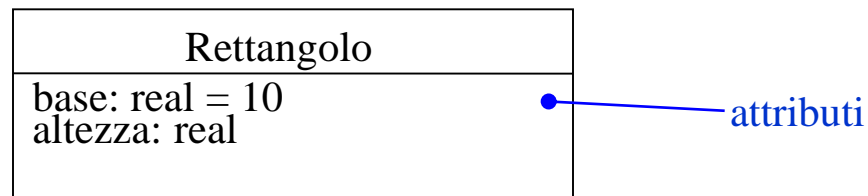
Ogni classe deve avere un nome che la contraddistingue dalle altre
Questo può essere semplice o indicare un percorso.



Graficamente gli attributi sono indicati sotto il nome della classe.

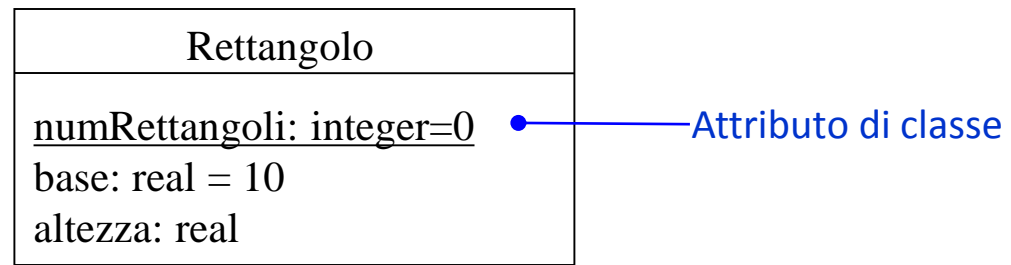


Di un attributo si può specificare l'insieme dei valori assunti (oggetti di una classe) e una eventuale inizializzazione.



Le classi: attributi statici

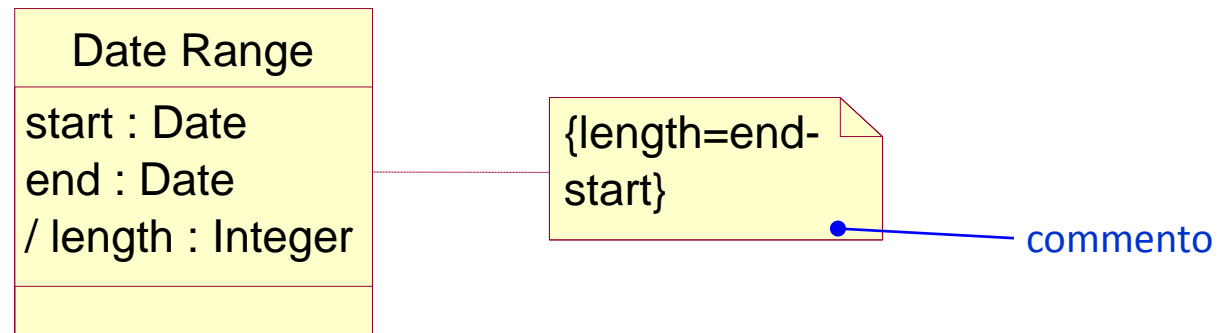
Un attributo di classe (detto **statico** in UML) è indicato come sottolineato



L'attributo numRettangoli indica il numero di oggetti della classe Rettangoli che sono stati istanziati. È un attributo statico (o di classe) in quanto condiviso da tutte le istanze della classe Rettangoli.

Le classi: attributi derivati

Gli attributi **derivati** sono quelli che possono essere calcolati partendo da altri attributi. UML prevede una rappresentazione specifica mediante una **'/'**.

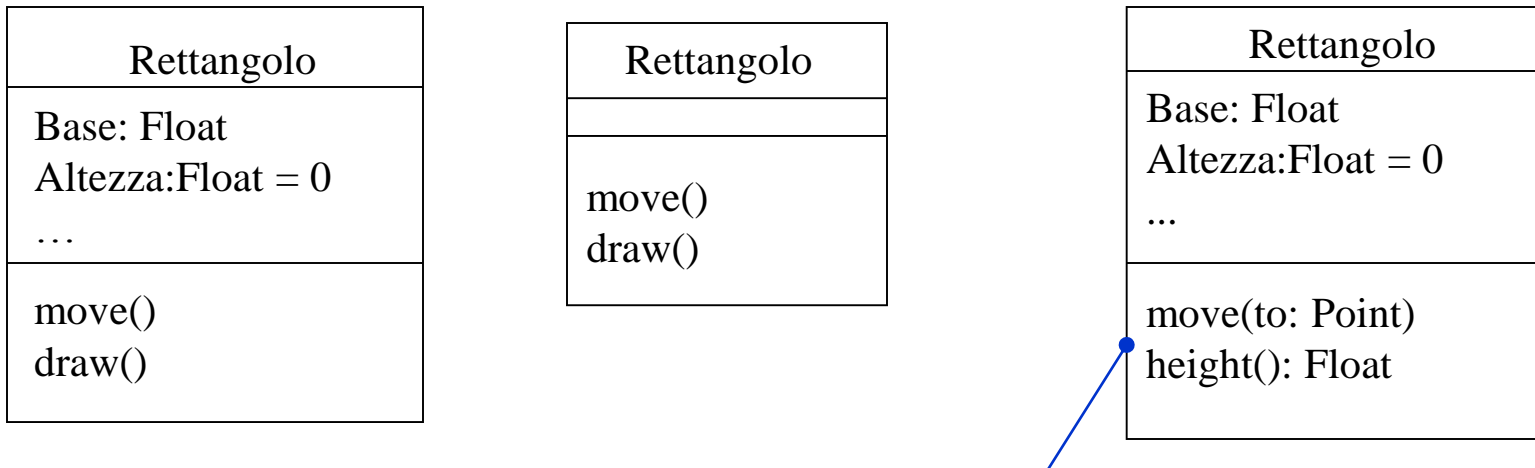


Il commento sulla destra fa parte della notazione standard UML e in questo caso è utilizzato per indicare come si calcola l'attributo derivato *length*.

I commenti possono essere aggiunti a qualunque elemento della notazione UML.

Le classi: i metodi

In UML le operazioni sono indicate graficamente in una sezione al di sotto degli attributi:



Di una operazione si può specificare la **segnatura**

Le classi

Una classe può essere rappresentata a diversi livelli di dettaglio:

- specificandone solo il nome:

Cliente

- specificandone gli attributi e i metodi:

Rettangolo
base altezza ...
muovi() vuoto() ...

Rettangolo
base:int=10 altezza:int ...
muovi(xoff:int; yOffset:int) vuoto():boolean ...

Rettangolo
<<costruttori>> rettangolo() rettangolo(x:int;y:int;h:int;b:int) <<accesso>> vuoto():boolean ...

I diversi livelli di dettaglio consentono a chi progetta/modella di attribuire maggiore o minore importanza a determinati fattori a seconda della vista del sistema che si sta considerando

Gli stereotipi

Nel precedente esempio si è fatto uso della notazione «**costruttori**» e «**accesso**». Questi sono due tipici esempi di ***stereotipi***

Gli stereotipi sono dei tipici ***meccanismi di estendibilità*** di UML. Infatti essi estendono il vocabolario di UML, permettendo di creare nuovi blocchi per la costruzione dei modelli, derivandoli da blocchi già esistenti ma rendendoli specifici per il particolare dominio

Nel precedente esempio è stato esteso il blocco “operazioni” in modo da poter distinguere le diverse tipologie di operazioni

Gli stereotipi sono identificabili perché racchiusi da « »

Sono già previsti diversi stereotipi in UML

Visibilità di attributi e metodi

Gli attributi e i metodi di una classe possono avere diversi livelli di **visibilità**. Un elemento (attributo o metodo) ha visibilità **pubblica** quando può essere visto (utilizzato, invocato) da altre classi. Ha visibilità **privata** quando può essere visto solo dalla classe di appartenenza.

Ad esempio, un metodo pubblico può essere invocato da qualunque punto del codice (purché la classe sia 'importata' in qualche modo), mentre un metodo privato può essere invocato solo da altri metodi della stessa classe.

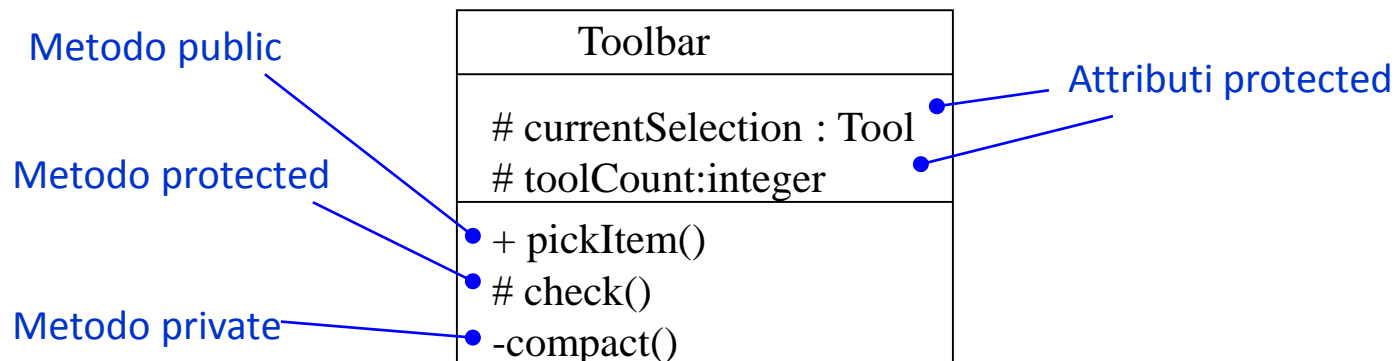
Altri livelli di visibilità sono:

- **protetta**: l'elemento è visibile all'interno del package e all'esterno solo ai discendenti della classe di appartenenza
- **package**: l'elemento è visibile solo agli elementi del package che contiene la classe in cui l'elemento è definito

Visibilità di elementi

UML consente di specificare i livelli di visibilità di attributi e metodi utilizzando la seguente notazione:

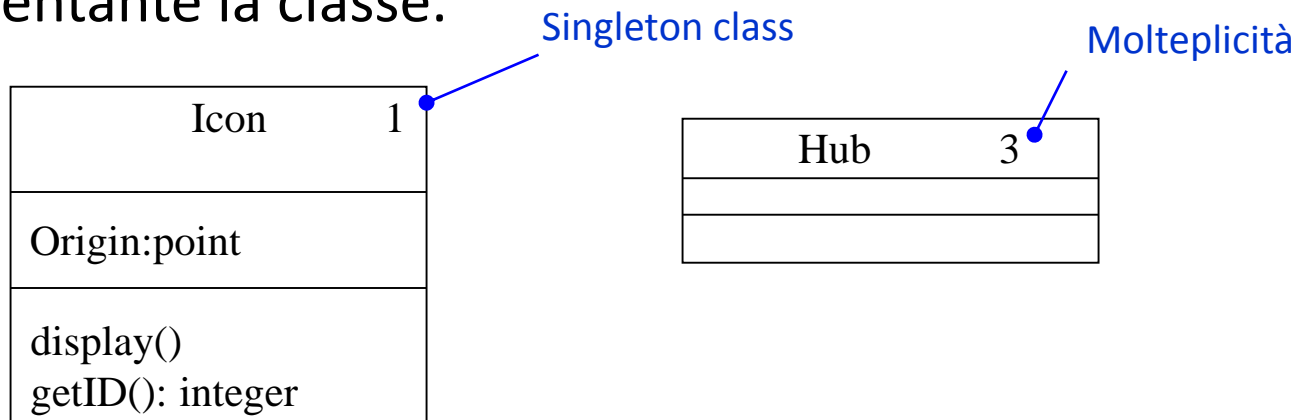
- **public**: l'elemento è preceduto da un +
- **protected**: l'elemento è preceduto da un #
- **private**: l'elemento è preceduto da un -
- **package**: l'elemento è preceduto da un ~



Molteplicità di classe

Con il termine **molteplicità di classe** si intende il numero di istanze che essa può avere. Generalmente non si pone un limite, tuttavia in alcuni casi è necessario indicare che la classe può avere una sola istanza (**classe singoletto**, *singleton class*) o comunque un numero ben definito di istanze.

UML impone che tale valore sia indicato in alto a destra nell'icona rappresentante la classe.



Esercizio

Come si può realizzare una classe singoletto?

Icon	1
<u>+istanzaUnica: Icon = new Icon()</u>	
<<costruttori>> - Icon() <u>+ getInstance() : Icon</u>	

Rendendo privati i costruttori della classe e dotando la classe di un attributo statico che è inizializzato all'unico oggetto di quella classe. Ovviamente, nessun metodo della classe singoletto invocherà il costruttore, né potrà farlo un utente della classe. **Inoltre può essere utile disporre di un metodo di classe che restituisce l'unica istanza memorizzata nella stessa classe singoletto.**

Schema per la definizione di un attributo

Si è visto come è possibile specificare in UML le varie caratteristiche degli attributi di una classe. A riepilogo, mostriamo lo schema generale per la definizione di un attributo:

[visibilità] nome [molteplicità] [:tipo]

[= valore iniziale] [{proprietà}]

Esempi di definizioni lecite di attributo:

- | | |
|---------------------------|------------------------------|
| • origine | solo il nome |
| • + origine | visibilità e nome |
| • origine : point | nome e tipo |
| • testa : *elemento | nome e tipo complesso |
| • nome [0..1] : String | nome, molteplicità e tipo |
| • origine : Point = (0,0) | nome, tipo e valore iniziale |
| • id : Integer {frozen} | nome, tipo e proprietà |

Proprietà per gli attributi

Focalizziamo l'attenzione sull'ultimo degli esempi:

- `id : Integer {frozen}` nome, tipo e proprietà

In UML esistono tre **proprietà** predefinite che possono essere utilizzate con gli attributi:

- **changeable**: non vi sono restrizioni sulla modificabilità dell'attributo
- **addOnly**: per gli attributi con molteplicità maggiore di uno, i valori possono essere aggiunti, ma una volta creati, non possono più essere rimossi o modificati
- **frozen**: il valore dell'attributo non può essere modificato dopo che l'oggetto è stato inizializzato

Nel caso in cui la proprietà non viene specificata si sottintende che assume valore *changeable*

Schema per la definizione di una operazione

Prima di procedere con la definizione dello schema per la definizione di un'operazione, è necessario sottolineare che UML distingue tra operazione e metodo:

- una **operazione** è un servizio che può essere richiesto alla classe
- un **metodo** è un'implementazione del servizio

Infatti, possono anche esistere diversi metodi per la stessa operazione nei diversi livelli della gerarchia delle classi.

Schema per la definizione di una operazione:

[visibilità] nome [(lista dei parametri)]

[: valore di ritorno] [{proprietà}]

Schema per la definizione di una operazione

Esempi di definizioni lecite di operazione :

- visualizza solo il nome
- + visualizza visibilità e nome
- set(n : Nome, s : String) nome e parametri
- getID() : Integer nome e tipo del dato restituito
- riparti() {guarded} nome e proprietà

Ogni parametro riportato nella segnatura prende la forma:

[direzione] nome : tipo [= valore iniziale]

La **direzione** può assumere uno dei seguenti valori:

- **in** parametro di input, non può essere modificato
- **out** parametro di output, può essere modificato per comunicare un'informazione al chiamante
- **inout** parametro di input che comunque può essere modificato

Schema per la definizione di una operazione

UML fornisce diverse proprietà predefinite per le operazioni:

- **isQuery**: l'esecuzione dell'operazione lascia lo stato del sistema immutato, un'operazione con tale proprietà è quindi priva di *side-effect*
- **leaf**: l'operazione non può essere più specializzata (overriding) nelle sottoclassi (vedi *final* in Java)
- **sequential**: i chiamanti (callers) di questo oggetto devono coordinarsi affinché solo uno alla volta richieda il servizio. Nel caso di sovrapposizione la semantica e l'integrità dell'oggetto **non sono garantite**
- **guarded**: simile al caso precedente, in ogni istante un solo chiamante può usufruire del servizio, tuttavia, in questo caso, la sequenzialità del servizio è gestita dalla classe proprietaria del servizio stesso (vedi *synchronized* in Java)
- **concurrent**: la semantica e l'integrità dell'oggetto è garantita anche in caso di chiamate multiple

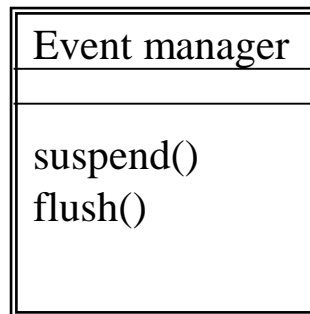
Classi attive

Le ultime tre proprietà sono rilevanti solo in classi **attive**, cioè in classi i cui **oggetti** sono **attivi**.

Un oggetto è attivo se esso ha un thread e può far partire un thread concorrente.

Una classe attiva è simile ad una classe con l'eccezione che le sue istanze rappresentano elementi il cui comportamento è concorrente con gli altri.

Essa è mostrata con bordi raddoppiati.



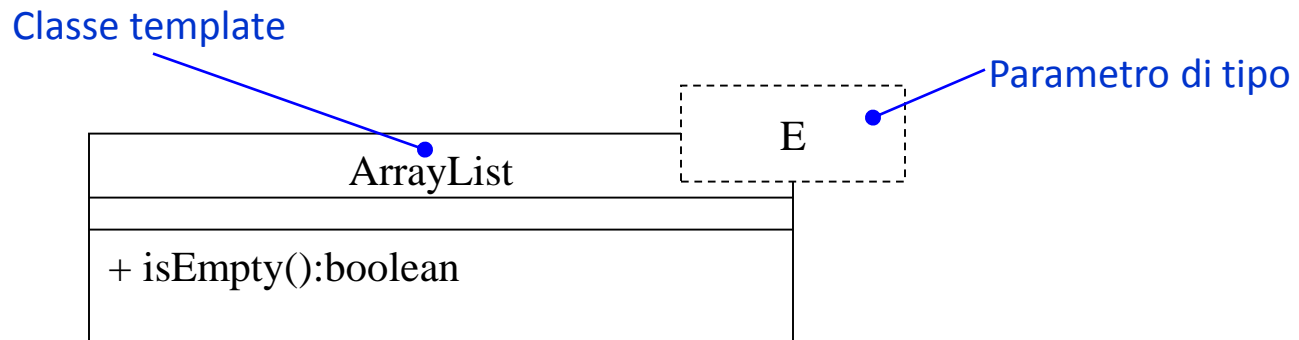
Classi template

Una *classe template* definisce una famiglia di classi parametrizzate (con parametro di tipo). Non è possibile usare direttamente una classe template. È necessario prima specificare il tipo (operazione di istanziiazione). In Java, una classe template corrisponde a una classe generica.

Esempio:

```
public class ArrayList<E>
```

Notazione UML



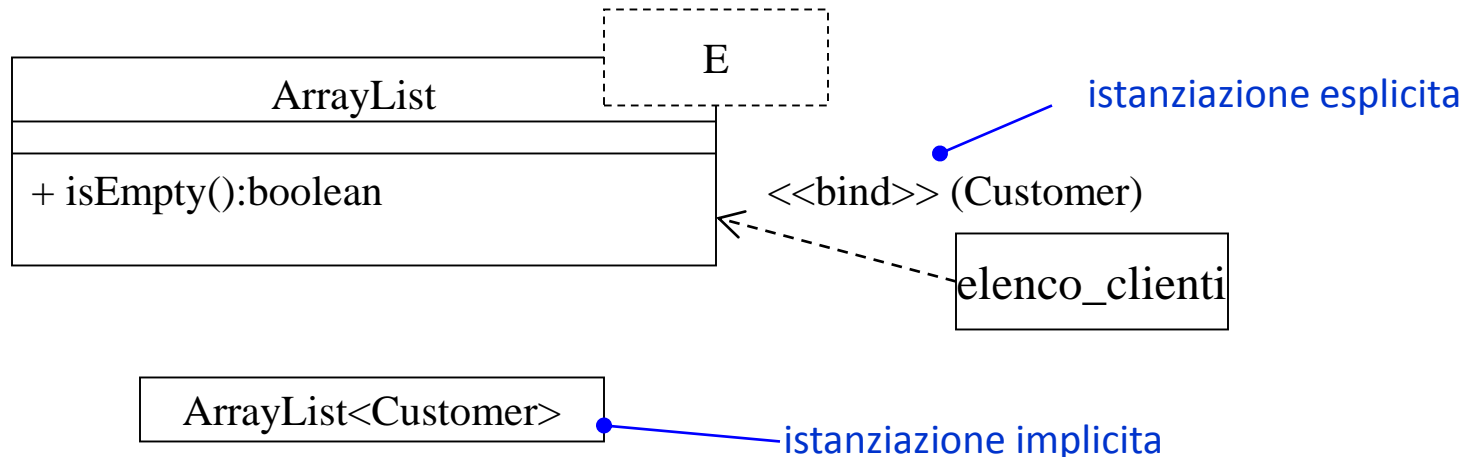
Classi template

Si potrebbe voler istanziare questo template per creare una lista di clienti.

```
ArrayList<Customer> elenco_clienti;
```

L'istanziatura di una classe template può essere effettuata in due modi:

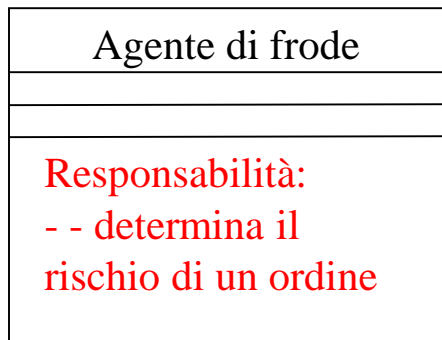
1. Implicitamente, dichiarando una classe il cui nome esplicita i parametri
2. Esplicitamente, mediante una dipendenza stereotipata <<bind>>



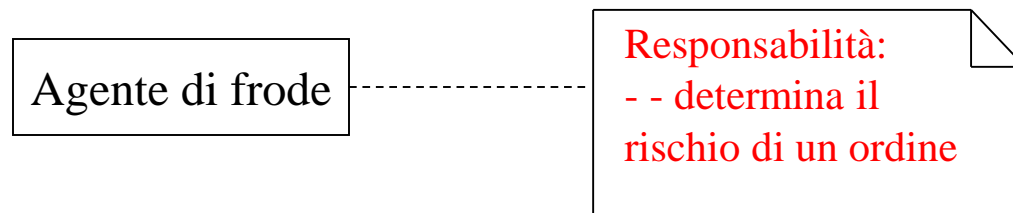
Le classi: individuazione delle responsabilità

In una buona modellazione OO di un sistema software è necessario stabilire le responsabilità da attribuire a ciascuna classe individuata. UML consente di modellare le responsabilità in due modi:

- specificandole all'interno della classe:



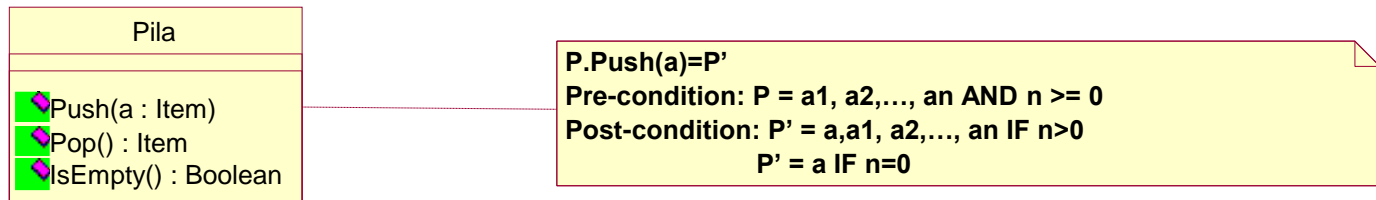
- utilizzando delle note:



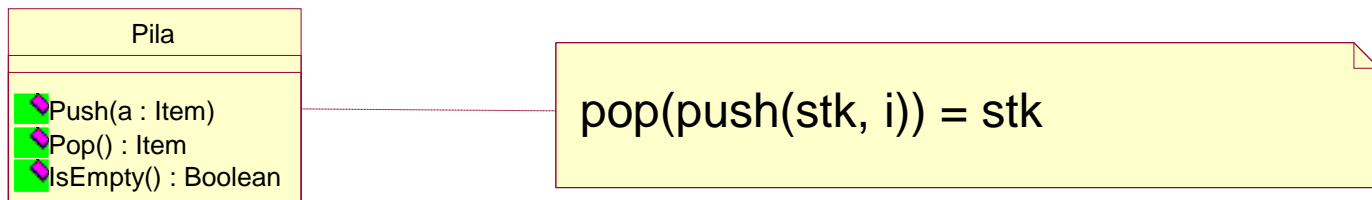
Le classi: individuazione delle responsabilità

La specifica della responsabilità di una classe serve a definire cosa fa una classe (la semantica della classe)

In alternativa si possono indicare pre- e post-condizioni delle operazioni (specifica assiomatica)



o un insieme di equazioni (specifica algebrica)



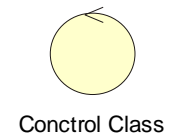
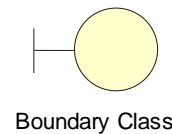
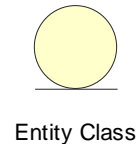
Classi (identificazione)

- Al fine di rendere il sistema altamente riutilizzabile, il lavoro di individuazione delle classi deve essere condotto in modo preciso
- È necessario a tal fine seguire una metodologia ben definita:
 - identificare gli elementi che gli utenti usano per descrivere il problema
 - per ogni astrazione individuata è necessario identificare un insieme di responsabilità, sincerandosi che vi sia un buon bilanciamento di responsabilità tra le classi
 - fornire ad ogni classe gli attributi e le operazioni di cui ha bisogno per eseguire tali responsabilità
- L'individuazione delle classi dipende dunque sia dalla realtà che si vuole modellare, sia dalla necessità di individuare il giusto bilanciamento dei compiti da assegnare, al fine di rendere il sistema software altamente mantenibile e riutilizzabile

Stereotipi di classi

Alcune metodologie suggeriscono l'individuazione e la classificazione delle classi secondo i seguenti gruppi:

- classi entità
- classi di confine
- classi di controllo



Tale suddivisione, modellata mediante l'utilizzo di stereotipi, permette il partizionamento del sistema in tre componenti differenti:

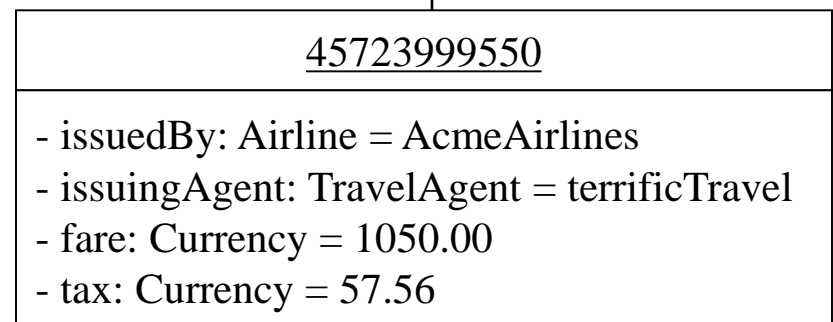
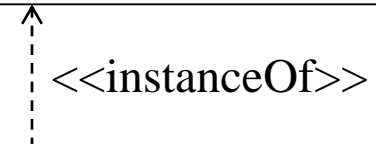
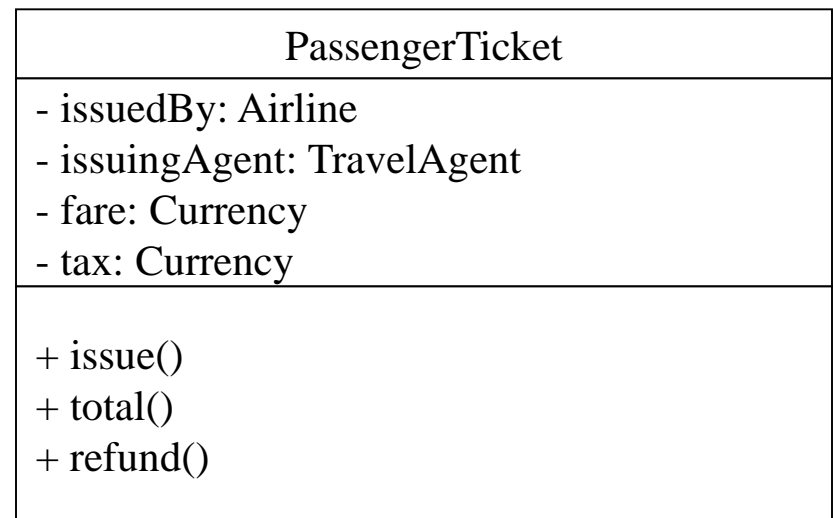
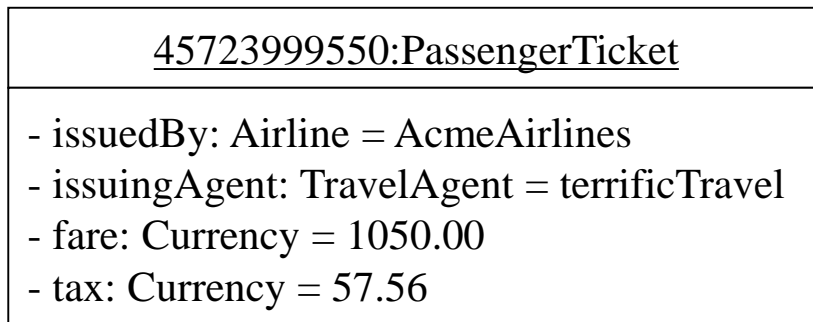
- dominio
- vista
- controllo

La relazione Instance-of

Fra un oggetto e una classe sussiste una relazione 'instance of' che specifica che un oggetto è una istanza di una classe.

In UML questa relazione è resa con lo stereotipo <<instanceOf>>.

Le seguenti notazioni grafiche sono semanticamente equivalenti (ma cambia il livello di dettaglio):



Ereditarietà: una relazione fra classi

Nella progettazione e programmazione OO una relazione fondamentale è quella esistente fra le classi: **la relazione di ereditarietà** (*inheritance*).

Una classe è considerata come un *repertorio di conoscenze* a partire dal quale è possibile definire altre *classi più specifiche*, che completano le conoscenze della loro classe madre.

Una **sottoclasse** è dunque, una *specializzazione* della descrizione di una classe, detta la sua **superclasse**, della quale essa mutua (parte di) gli attributi e i metodi.

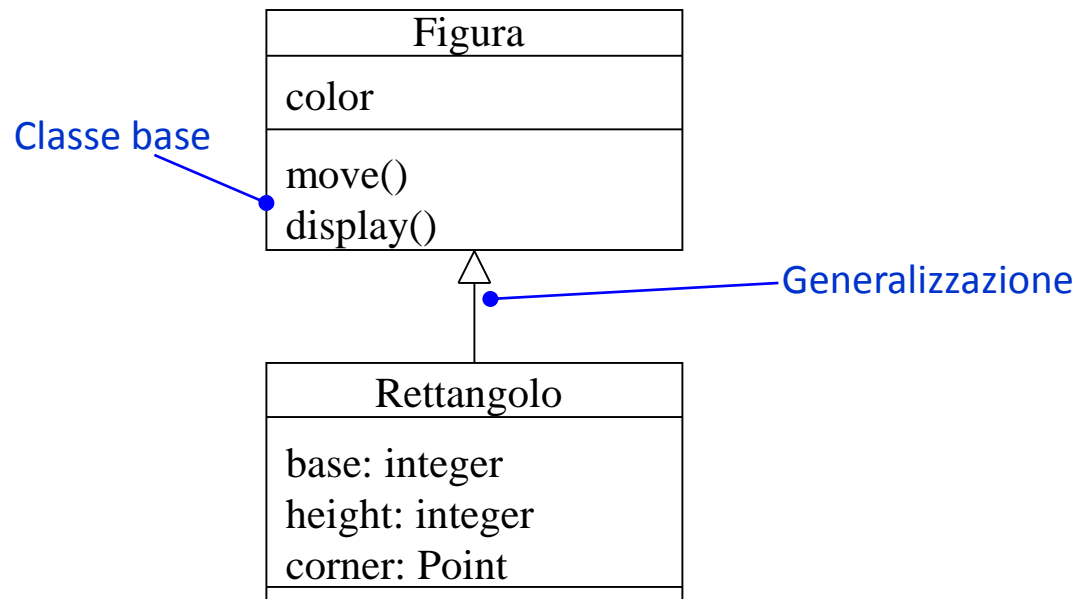
Ci sono diverse forme di ereditarietà. Ne riportiamo alcune.

Ereditarietà per estensione (*extension inheritance*)

Estensione

La sottoclasse introduce delle caratteristiche (attributi e metodi) non presenti nella superclasse e non applicabili a istanze della superclasse.

La visibilità (pubblica, protetta, privata, package) degli attributi e delle operazioni ereditate dalla superclasse non è modificata.



Ereditarietà per estensione

Esempio: gli oggetti di Rettangolo dispongono dell'attributo ereditato '*color*' e dei metodi ereditati *move* e *display*. Inoltre gli oggetti di Rettangolo di attributi specifici dei rettangoli: *base*, *height* e *corner*.

- La specializzazione per estensione permette di sviluppare del codice estendibile
- Allorquando si dovesse avvertire la necessità di aggiungere ulteriori funzioni, occorrerà individuare le classi interessate e derivare da queste nuove classi alle quali verranno aggiunti gli attributi e i metodi necessari per implementare le nuove funzioni

Ereditarietà per variazione funzionale (*functional variation inheritance*)

Variazione funzionale

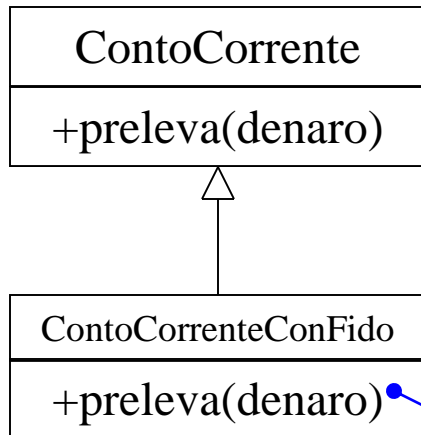
Si **ridefiniscono** alcune caratteristiche (metodi) della superclasse quando quelle ereditate si rivelano inadeguate per l'insieme di oggetti descritti dalla sottoclasse

La ridefinizione (***overriding***) del metodo ereditato **riguarda solo l'implementazione** e non la segnatura (nome, parametri formali e parametro di ritorno)

Ogni richiesta di esecuzione del metodo ridefinito da parte di un oggetto della sottoclasse, farà riferimento alla nuova implementazione fornita nella sottoclasse

Ereditarietà per variazione funzionale

Esempio



Nella classe *ContoCorrente* il metodo *preleva* controlla che il conto non vada in rosso.

Il metodo *preleva* della classe *ContoCorrenteConFido* ridefinisce quello della superclasse per controllare che il prelievo non vada oltre il fido concesso.

overriding

Cosa succede se cambiano le regole al contesto e si richiede che il prelievo in contante non possa superare i 10.000 Euro al giorno, indipendentemente dalla disponibilità?

Occorre modificare il metodo *preleva* di *ContoCorrente*, ma la modifica non viene riportata automaticamente anche alla classe *ContoCorrenteConFido*.

Ereditarietà per variazione funzionale

La ridefinizione non è incrementale, quindi i cambiamenti nel metodo originale devono essere riportati anche nei metodi ridefiniti. Purtroppo non c'è alcuna garanzia che questo accada e si possono introdurre degli errori

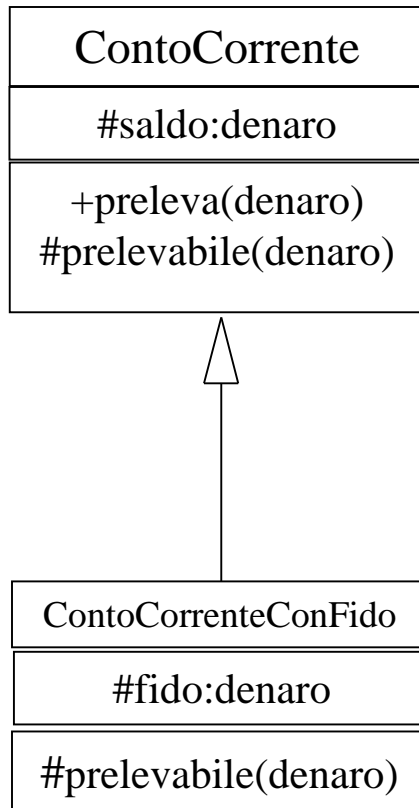
Molti esperti vedono nella ridefinizione una potenziale fonte di errori e ne sconsigliano l'uso. Secondo loro i metodi dovrebbero essere ereditati completamente senza sovrascrittura. Altrimenti non andrebbero specificati affatto (vedi metodi astratti)

Ereditarietà per variazione funzionale

Per mitigare gli effetti di questo problema, si può adottare qualche accorgimento nella realizzazione dei metodi per i quali si riconosce già in fase di progetto una incrementalità al cambiamento.

Es.: il metodo di una sottoclasse deve prevedere, nella sua realizzazione, l'invocazione del metodo della superclasse, e nel caso di risposta positiva, aggiungere ulteriori controlli

Ereditarietà per variazione funzionale



```
prelavabile(x:denaro)
{
    return (x<=saldo) ;
}
```

```
preleva(x:denaro){
    if(prelevabile(x) && x<=10000)
        saldo-=x;
}
```

```
prelavabile(x:denaro)
{
    return (super.prelevabile(x) || ( x<=saldo+fido));
}
```

Ereditarietà per variazione funzionale

Per combinare, in un overriding, il metodo della superclasse con il codice specifico della sottoclasse, i linguaggi di programmazione offrono diversi meccanismi.

In C++, si può esplicitare il nome della classe contenente il metodo che si intende invocare. Nell'esempio, il metodo prelevabile ridefinito in una sottoclasse potrà fare esplicitamente riferimento al metodo prelevabile della classe *ContoCorrente*:

`ContoCorrente::prelevabile()` ← Operatore di qualifica della visibilità

In Smalltalk e Java, un metodo definito in una (sotto-)classe può invocare una operazione della superclasse facendo riferimento a *super*

`super.prelevabile()`

Ereditarietà per variazione funzionale

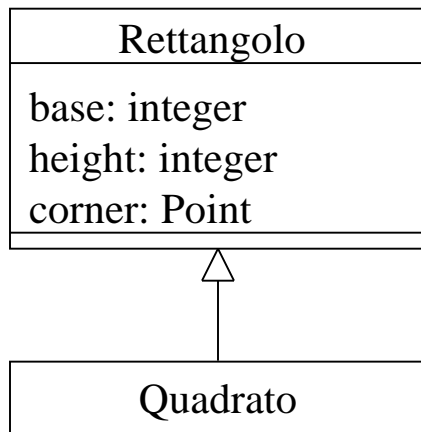
Osserviamo che anche nell'ereditarietà per variazione funzionale, la visibilità (pubblica, protetta, privata, package) degli attributi e delle operazioni ereditate dalla superclasse non è modificata.

Osserviamo inoltre che la variazione funzionale attiene solo le operazioni di accesso e trasformazione di una classe e non i costruttori degli oggetti.

Ereditarietà per restrizione (*restriction inheritance*)

Restrizione

Le istanze di una sottoclasse soddisfano vincoli che non sono necessariamente soddisfatti da istanze della superclasse.



Esempio

In matematica è possibile trovare molti esempi. Un quadrato è un particolare caso di rettangolo nel quale i due lati hanno medesima lunghezza.

L'ereditarietà per restrizione non chiede la modifica della visibilità degli attributi e delle signature delle operazioni ereditate dalla superclasse.

Principio di sostituibilità

Data una dichiarazione di una variabile o di un parametro il cui tipo è dichiarato come X, una qualunque istanza di una classe che è discendente di X può essere usato come valore effettivo senza violare la semantica della dichiarazione e il suo uso.

In altri termini, **l'istanza di un discendente può essere sostituita all'istanza di un ascendente**

Questo importante principio è attribuito al prof. *Barbara Liskov* del MIT

Il principio di sostituibilità è legato al **polimorfismo di inclusione**, nella programmazione orientata a oggetti

Principio di sostituibilità

La conseguenza del principio di sostituibilità è che **una sottoclasse non può rimuovere o rinunciare a proprietà/metodi della superclasse**. Altrimenti una istanza della sottoclasse non sarà sostituibile in una situazione in cui si dichiara l'uso di istanze della superclasse.

In effetti, preservando la visibilità degli attributi e dei metodi ereditati, così come accade nelle tre forme di ereditarietà viste, si garantisce che gli oggetti della sottoclasse offrano quanto meno gli stessi servizi degli oggetti della superclasse (anche se i servizi potranno essere implementati diversamente, come accade nella variazione funzionale).

Pertanto il principio di sostituibilità (o polimorfismo di inclusione) è compatibile con l'ereditarietà per estensione, variazione funzionale e restrizione.

Ereditarietà e relazione “is_a”

Nell'ereditarietà per

- estensione,
- variazione funzionale e
- restrizione

la relazione di ereditarietà fra classi corrisponde a una relazione di **generalizzazione** (o “**is_a**”). Ciò perché **ogni istanza di una classe derivata da una classe base va considerata come (è anche) una istanza della classe base.**

Esempio: *Le istanze di Rettangolo sono istanze anche di Figura.*

Il simbolo  usato in UML denota una generalizzazione.

Ereditarietà di implementazione (*implementation inheritance*)

Ereditarietà di implementazione

La sottoclasse utilizza il codice della superclasse (definizioni di attributi e metodi) per implementare l'astrazione associata.

Esempio

Una classe *Vettore* mette a disposizione una rappresentazione di un vettore, un costruttore, un metodo di accesso (a un elemento sulla base di un indice) e uno di trasformazione (cambiamento di stato di un elemento di cui si conosce la posizione)

Una classe *Pila* può essere definita come sottoclasse di *Vettore*, in modo da potersi basare sulla rappresentazione del vettore per poter rappresentare una pila. Si dice che la realizzazione del dato astratto *Pila* è **delegata** a quella del *Vettore*. Se si cambia la rappresentazione del *Vettore* si cambia anche quella della *Pila*

Ereditarietà di implementazione

Esempio (cont):

Grazie all'ereditarietà per implementazione, la realizzazione degli operatori *push*, *pop*, *top* previsti per una pila può basarsi sugli operatori messi a disposizione dalla classe *Vettore*

Si noti bene che la *Pila* ha un diverso insieme di operatori rispetto a quelli del *Vettore*. L'ereditarietà riguarda solo la realizzazione del vettore e l'implementazione dei metodi, **non l'interfaccia** della classe (ciò che è visibile agli utilizzatori di *Vettore*)

Pertanto l'ereditarietà di implementazione comporta la modifica alla visibilità delle caratteristiche ereditate

Ereditarietà di implementazione

L'**ereditarietà a base privata** supportata dal C++ e da Ada-95 permette di rendere privati gli attributi e gli operatori pubblici ereditati.

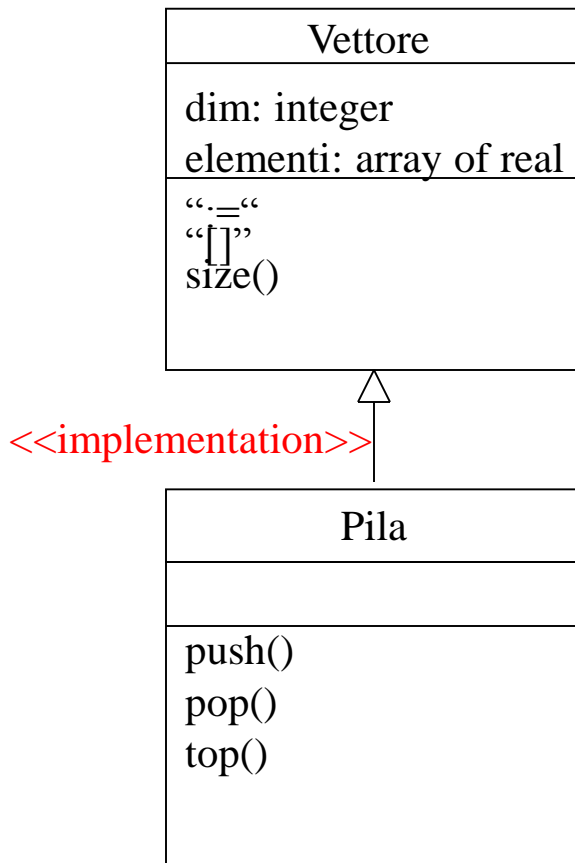
Visibilità con **ereditarietà a base pubblica**:

Base\Derivata	Visibile ai clienti	Visibile alla sottoclasse
Public Spec.	✓	✓
Private Spec.	✗	✗

Visibilità con **ereditarietà a base privata**:

Base\Derivata	Visibile ai clienti	Visibile alla sottoclasse
Public Spec.	✗	✓
Private Spec.	✗	✗

Ereditarietà di implementazione



In UML l'ereditarietà di implementazione è indicata utilizzando lo stesso simbolo della generalizzazione, ma specificando a fianco lo stereotipo `<<implementation>>`.

Ereditarietà di implementazione

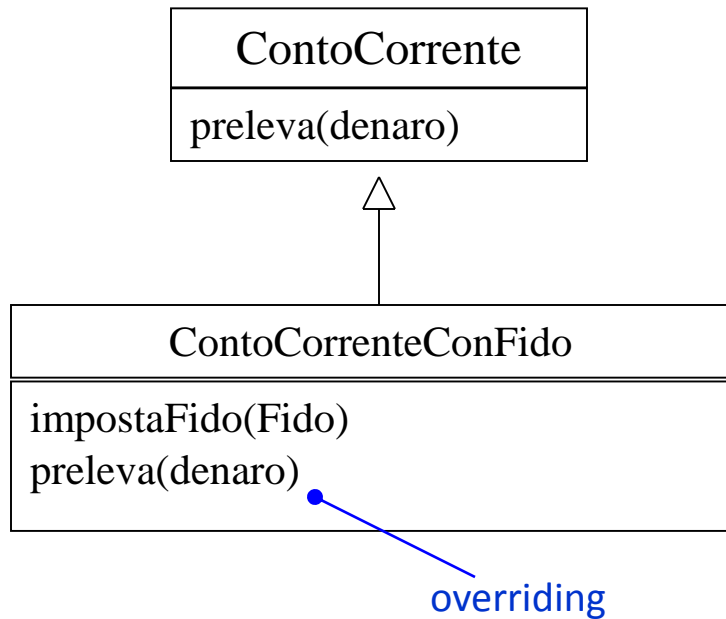
- L'ereditarietà di implementazione non è compatibile con il principio di sostituibilità
- Pertanto se la classe Y eredita solo l'implementazione della classe X, non si può riutilizzare su istanze di Y tutto il codice in cui si dichiarano e utilizzano dati di classe X, in quanto non vale il polimorfismo di inclusione
- L'ereditarietà di implementazione permette un **riuso parziale** del codice, pertanto secondo alcuni andrebbe bandita. Tuttavia Bertrand Meyer sostiene che questa forma di ereditarietà, insieme a molte altre incompatibili con il principio di sostituibilità, sono comunque teoricamente legittime e praticamente indispensabili

Per ulteriori forme di ereditarietà si veda l'articolo:

Bertrand Meyer, "The many faces of inheritance: A taxonomy of taxonomy," Computer, vol. 29, no. 5, pp. 105-108, May, 1996

Ereditarietà

Nella progettazione di una classe si possono combinare diverse forme di ereditarietà.



In questo esempio si ha ereditarietà per estensione (si aggiunge il metodo *impostaFido*) e per variazione funzionale (si ridefinisce il metodo *preleva*).

Proprietà della relazione di generalizzazione

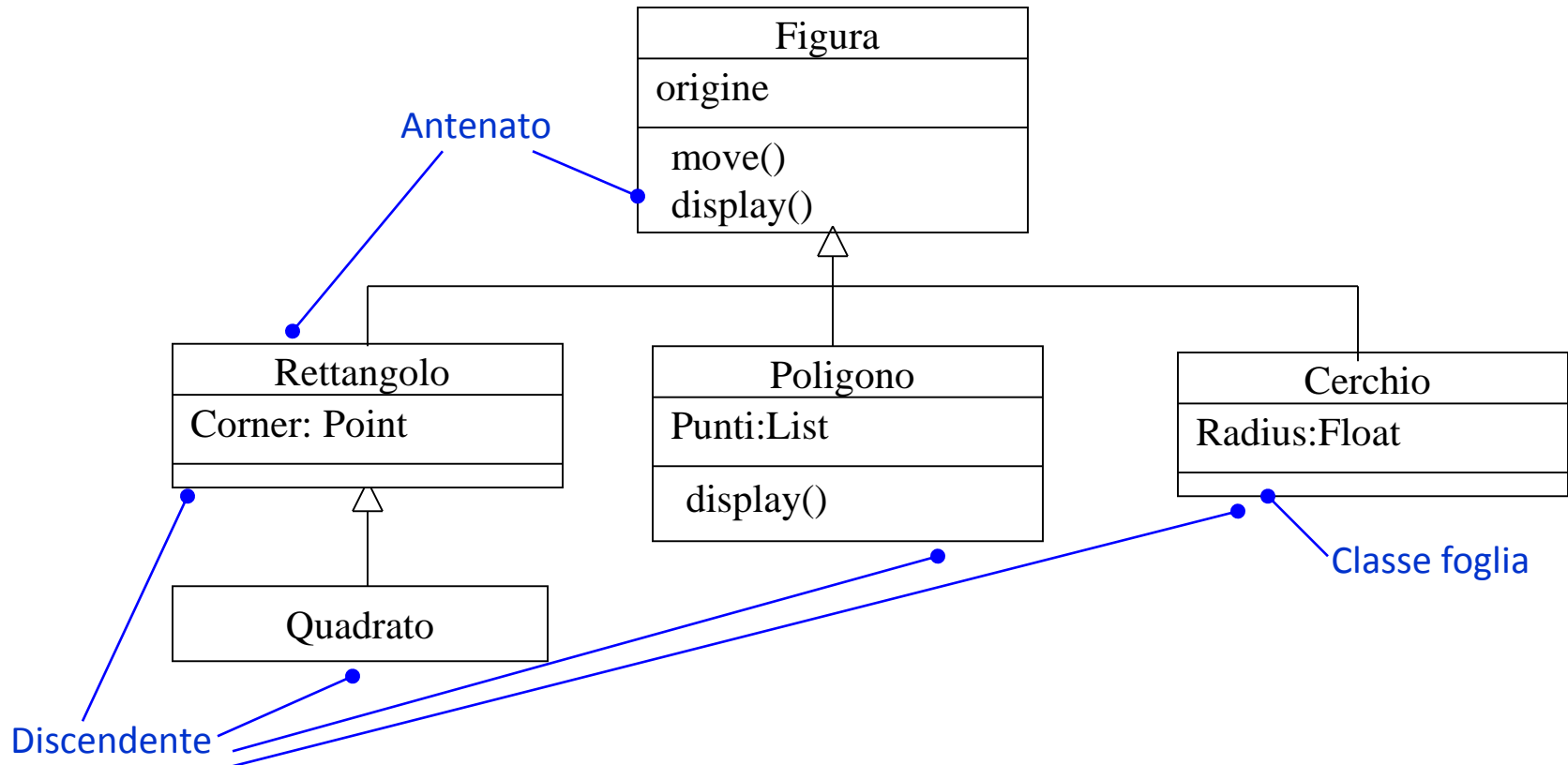
- La rappresentazione della relazione di generalizzazione fra un insieme di classi definisce un **grafo di ereditarietà** che è un **grafo orientato aciclico** (*directed acyclic graph*, dag).
- La relazione di generalizzazione è transitiva e antisimmetrica.

La **transitività** comporta che le caratteristiche delle classi superiori sono ereditate dalle classi inferiori.

L'**antisimmetria** definisce una direzione di attraversamento del grafo di ereditarietà che porta dalla sottoclasse alla superclasse.

Partendo da una classe C e seguendo la direzione che porta al genitore, si trovano tutte le classi **antenate** (*ancestor*), che per estensioni sono chiamate **superclassi** di C. Seguendo la direzione opposta si trovano tutte le classi **discendenti** (*descendant*) di C.

Grafo di ereditarietà



Ogni classe C ha il proprio **grafo di ereditarietà $G(C)$** che è una restrizione del grafo di ereditarietà completo alle sole superclassi della classe in esame.

Ereditarietà singola

In questo caso specifico, il grafo è in realtà un albero: ogni classe ha una sola superclasse diretta.

Si dice che l'ereditarietà è **semplice** (*simple*) o **singola** (*single inheritance*) .

In questo caso il **grafo di ereditarietà $G(C)$** di una classe C è una catena di antenati. Gli elementi della catena sono ordinati secondo una relazione d'ordine totale.

Poiché un metodo può essere ridefinito in più classi si pone il seguente

Problema: Sia dato un metodo m , eventualmente ereditato, della classe C_1 . Da quale classe $C_2 \in G(C_1)$ si eredita m ?

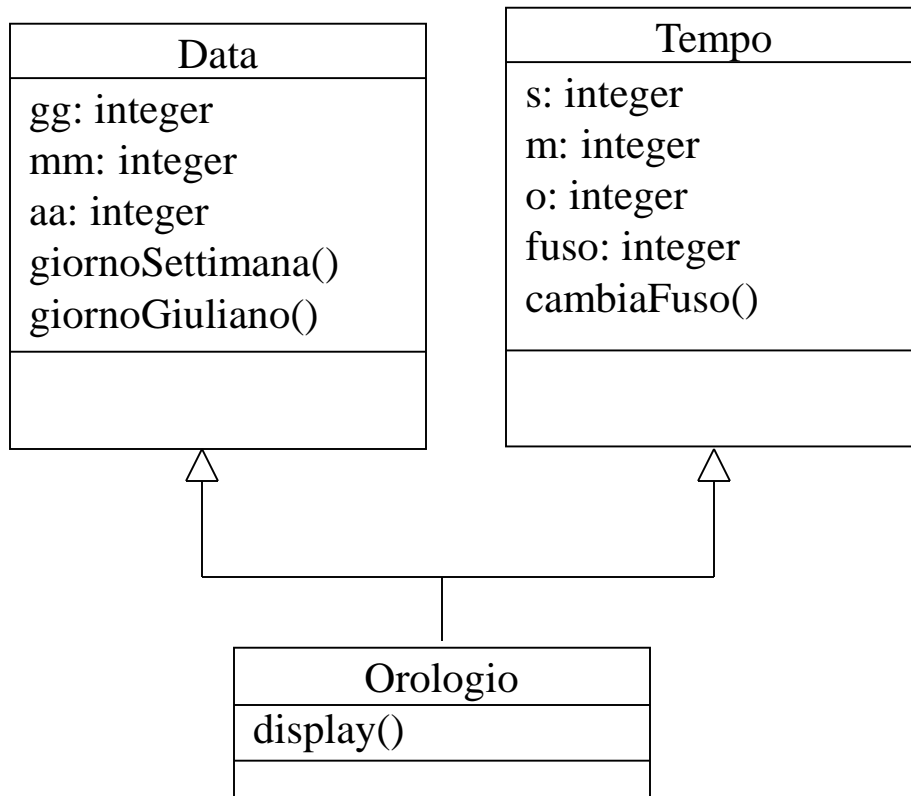
Soluzione

1° passo: si determina catena di antenati di C_1

2° passo: si ricerca la **prima occorrenza** della (ri-)definizione di m a partire dall'estremità C_1 della catena.

Ereditarietà multipla

Una classe può avere più superclassi. In questo caso si parla di ereditarietà **multipla** (*multiple inheritance*).



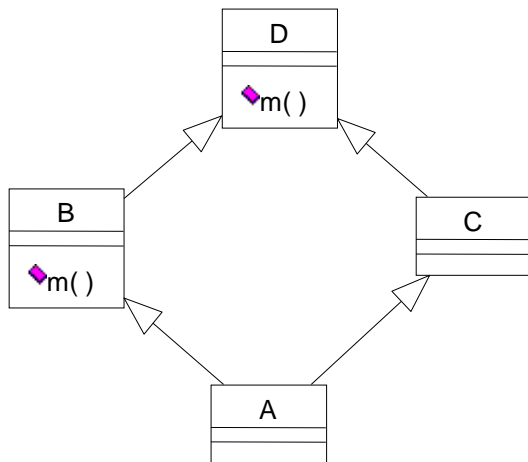
Esempio: la classe Orologio necessita dei servizi messi a disposizione sia da *Data* che da *Tempo*. In più ne implementa degli altri.

Il grafo di ereditarietà non è più un albero. Data una classe C , $G(C)$ non è più una catena ma un grafo aciclico orientato. L'ordine fra le classi in $G(C)$ è parziale.

Ereditarietà multipla

Anche nell'ereditarietà multipla un metodo può essere ridefinito in diverse classi e si pone il seguente

Problema: Sia dato un metodo **m**, eventualmente ereditato, della classe **C₁**. Da quale classe **C₂ ∈ G(C₁)** si eredita **m**?



Se $C_1=A$ c'è un conflitto fra le diverse definizioni di m in B e D , entrambe ereditabili. Il conflitto può essere facilmente risolto in questo caso. Si considera una qualunque *linearizzazione* del grafo $G(C_1)$. Si possono avere due casi:

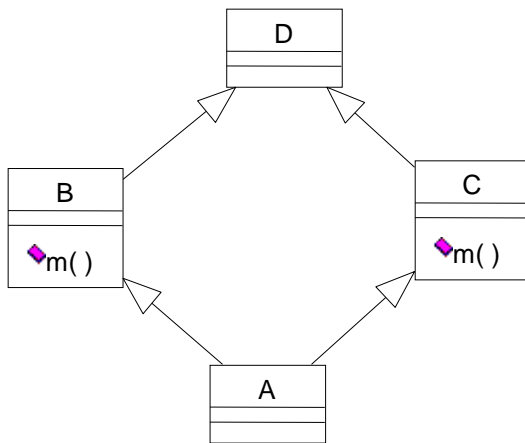
$A \rightarrow B \rightarrow C \rightarrow D$

$A \rightarrow C \rightarrow B \rightarrow D$

In entrambi i casi, B precede D , il che significa che **il metodo m è ereditato da B** . Il metodo m di B è quello più specifico e maschera quello omonimo di D .

Ereditarietà multipla

Problema: Sia dato un metodo **m**, eventualmente ereditato, della classe **C₁**. Da quale classe **C₂ ∈ G(C₁)** si eredita **m**?



Se $C_1 = A$ c'è un conflitto fra le diverse definizioni di **m** in B e C, entrambe ereditabili.

Le due linearizzazioni del grafo $G(C_1)$:

$A \rightarrow B \rightarrow C \rightarrow D$

$A \rightarrow C \rightarrow B \rightarrow D$

non aiutano a scegliere.

Esistono dei criteri euristici per gestire queste situazioni conflittuali:

1. La molteplicità dell'ereditarietà
2. La modularità

Ereditarietà multipla

1. La molteplicità dell'ereditarietà

Nel definire che A deriva dalle due superclassi occorrerà elencarle in un qualche ordine:

$A \longrightarrow B, C$ piuttosto che $A \longrightarrow C, B$

L'ordine delle classi è considerato per risolvere il conflitto. Questo significa che l'ordine delle classi è utilizzato per preferire una delle due linearizzazioni. Tuttavia questo principio **può essere in contraddizione con quello che indica di preferire l'eredità da classi più specifiche.**

Esempio: $A \longrightarrow B, C$ e $C \longrightarrow B, D$

Se m è definito sia in B e sia in C, si dovrebbe preferire C in quanto C precede sempre B nelle linearizzazioni:

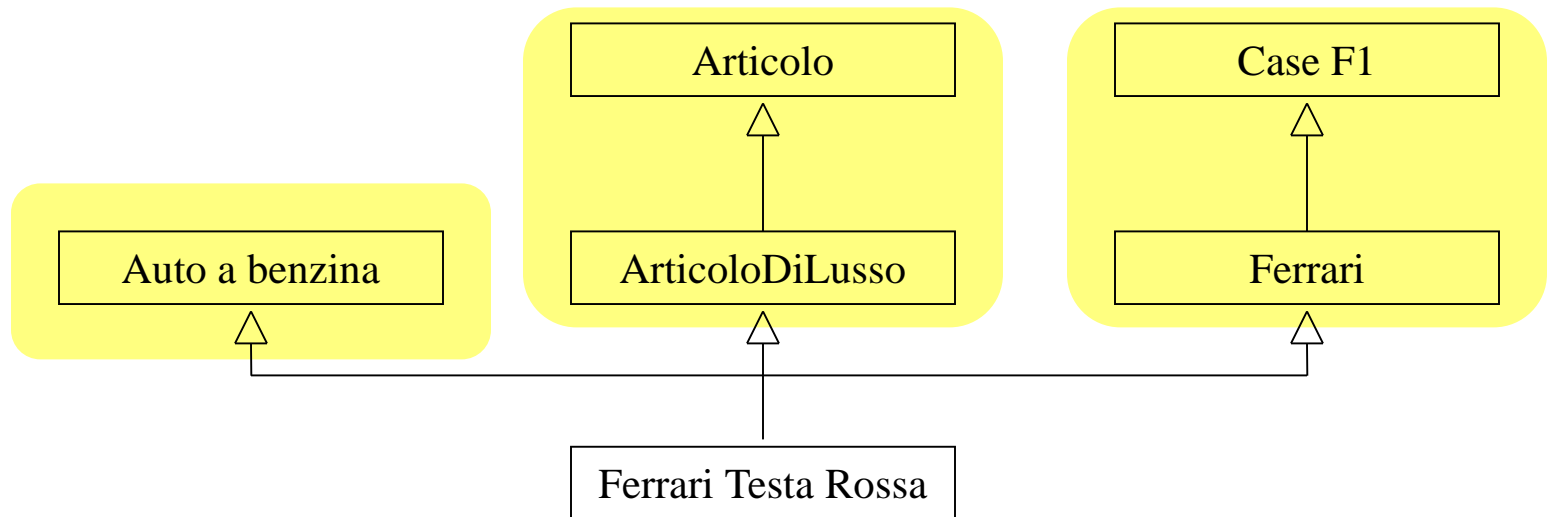
$A \rightarrow C \rightarrow B \rightarrow D$, però nella definizione di A abbiamo preferito B

Ereditarietà multipla

2. La modularità

Si può scomporre un grafo di ereditarietà in moduli che corrispondono ai diversi punti di vista sull'oggetto.

Esempio: i diversi moduli corrispondono ai diversi punti di vista (tipo alimentazione, tipo articolo, casa costruttrice).



Le linearizzazioni non devono mescolare i diversi sottografi associati ai moduli (non ci sono relazioni di ereditarietà tra classi appartenenti a moduli diversi). Anche in questo caso potrebbero esserci contraddizioni (da risolvere specificando l'ordinamento dei moduli).

Ereditarietà multipla

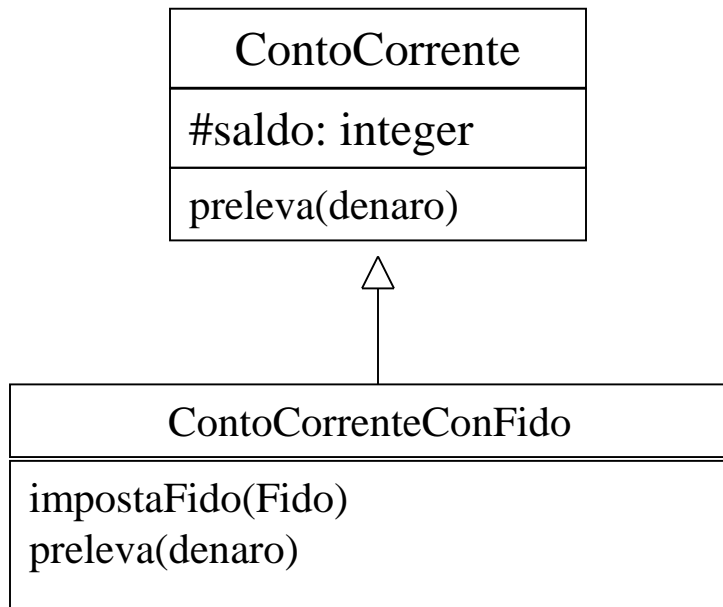
Queste contraddizioni mettono in evidenza le difficoltà che si incontrano a dare una semantica a un metodo quando si permette l'ereditarietà multipla.

I diversi principi per gestire le situazioni conflittuali non sono universalmente accettati. La risoluzione dei conflitti, in realtà, non può essere efficace se non prende in considerazione le conoscenze specifiche legate all'applicazione.

Per questo molti sconsigliano l'ereditarietà multipla, in quanto i benefici sono pochi mentre i problemi legati a una semantica ben definita sono molti.

Ereditarietà: la visibilità protetta

La relazione di ereditarietà introduce un ulteriore livello di visibilità: quella **protetta** (*protected*). Una caratteristica (attributo o metodo) con visibilità protetta può essere vista solo da una classe e da tutte le classi del package e dalle classi discendenti (anche in altri package).



In questo esempio la classe *ContoCorrenteConFido* potrà accedere al dato saldo, mentre ciò non sarà permesso ad altre classi dislocate in altri package che non ereditano da *ContoCorrente*.

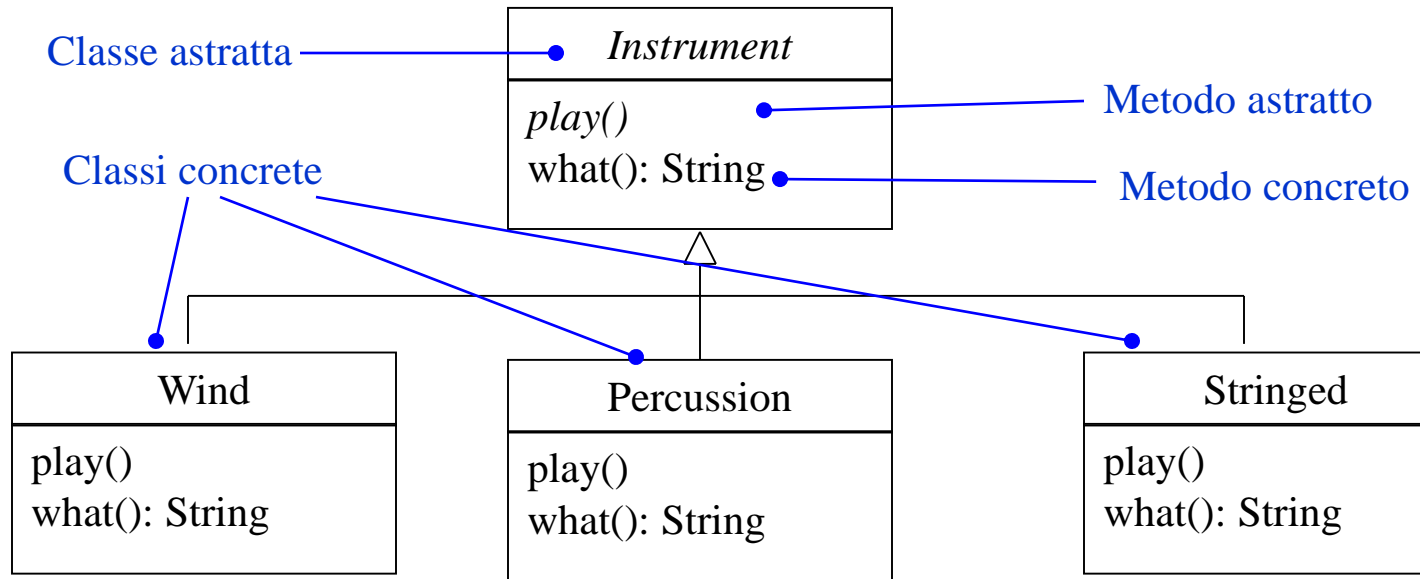
Classi astratte

- Se in programmazione object-oriented non può esistere un oggetto senza che sia stata creata la classe di appartenenza, è invece possibile che **esistano classi per le quali non è possibile generare delle istanze** (classi *astratte*)
- Una classe astratta può essere una classe non completamente specificata. In particolare, **non è definito il metodo corrispondente a una operazione** (il **metodo è astratto**)

Classi astratte (cont.)

- Una classe astratta non può essere istanziata perché il comportamento dei suoi oggetti non è completamente definito
- La relazione di ereditarietà riguarda anche le classi astratte

Classi astratte: notazione



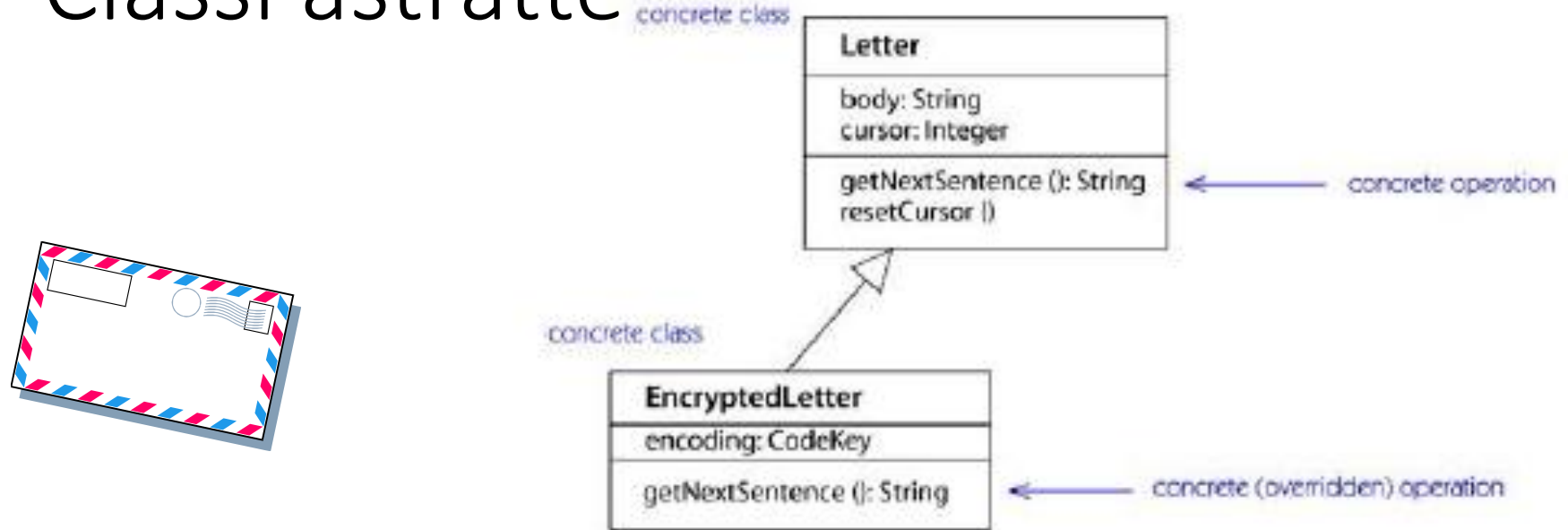
Esempio: la classe degli strumenti musicali è astratta perché non possiamo definire il metodo *play* (non sappiamo come si suoni uno strumento in generale). Il metodo *play* è implementato nelle sottoclassi degli strumenti a fiato, degli strumenti a percussione, e dei cordati. Il metodo *what* è invece **concreto** (è implementato) e restituisce il nome della classe. Esso è ridefinito in ogni classe.

Classi astratte

Le classi astratte sono strumenti per fattorizzare proprietà comuni tra classi simili e poterle organizzare in una gerarchia di ereditarietà. Non potremo mai creare oggetti a partire da una classe astratta, ma possiamo servircene **per dare una radice comune a un insieme di classi che condividono le stesse proprietà** e poter quindi sfruttare il polimorfismo di inclusione e il binding dinamico (di cui si parlerà in seguito)

Le classi astratte fungono da **serbatoi di ereditarietà**

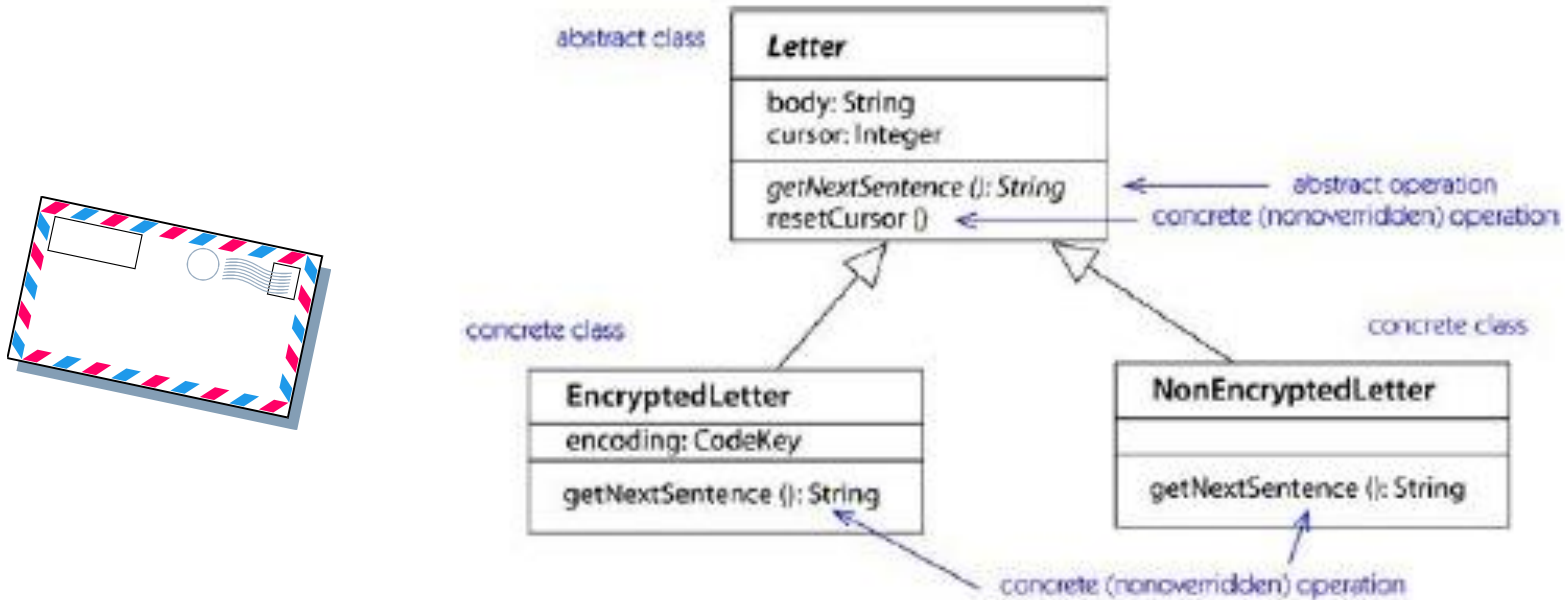
Classi astratte



La classe *Letter* ha un metodo che restituisce la frase successiva da leggere e l'operazione *resetCursor* che riporta all'inizio del testo. La sottoclasse *EncryptedLetter* rappresenta una lettera crittografata. Il metodo *getNextSentence* è stato sovrascritto perché il testo dev'essere decrittato prima di essere restituito. L'implementazione dell'operazione *getNextSentence* è completamente diversa da quella ereditata.

PROBLEMA: Il progettista di *EncryptedLetter* non ha alcuna informazione per comprendere quale metodo di *Letter* dev'essere sovrascritto.

Classi astratte



Se il metodo *getNextSentence* di *Letter* fosse astratto (e dunque fosse astratta anche la classe *Letter*) il progettista verrebbe informato della necessità di ridefinire il metodo. La classe astratta *Letter* fattorizza i metodi concreti comuni alle sue sottoclassi, come *resetCursor*.

Classi finali

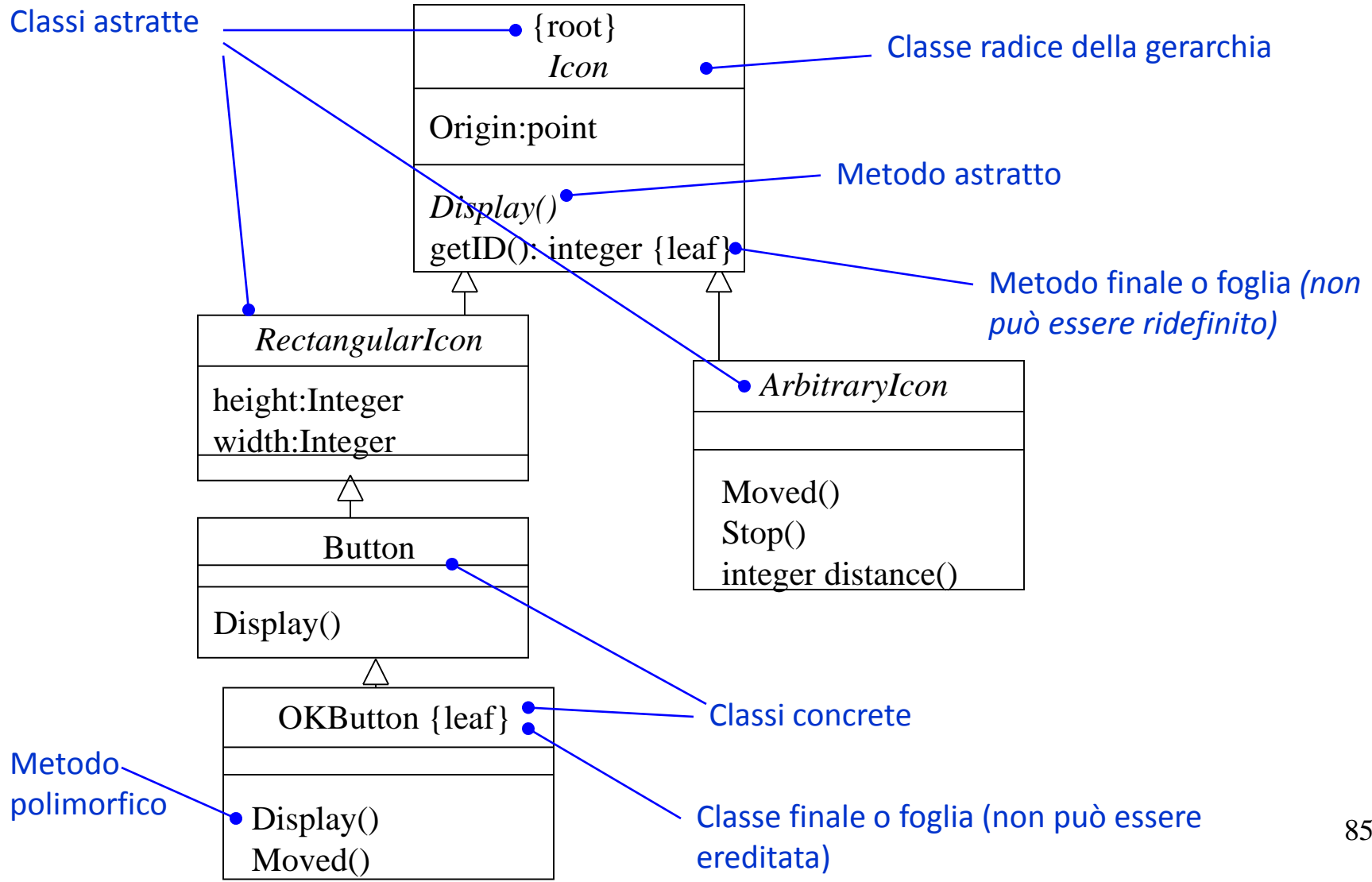
Una classe è detta finale (*final*) o foglia (*leaf*), quando non può essere ulteriormente specializzata, e quindi non può essere modificata

Si definisce una classe foglia quando il comportamento della classe dev'essere ben stabilito per ragioni di **affidabilità**

La dichiarazione di una classe foglia permette anche la generazione di codice ottimizzato in quanto facilita l'espansione in linea del codice (impossibile nel caso di metodi sovrascrivibili nelle sottoclassi).

Riflessione: ha senso stabilire una classe astratta foglia?

Class diagram in dettaglio (abstract, root, leaf, polymorphic)



Interfacce

Una interfaccia è la descrizione del comportamento degli oggetti senza specificarne una implementazione. Essa è una collezione di **operazioni**, cioè di servizi che possono essere richiesti, priva di informazioni sulle implementazioni dei servizi (i **metodi**).

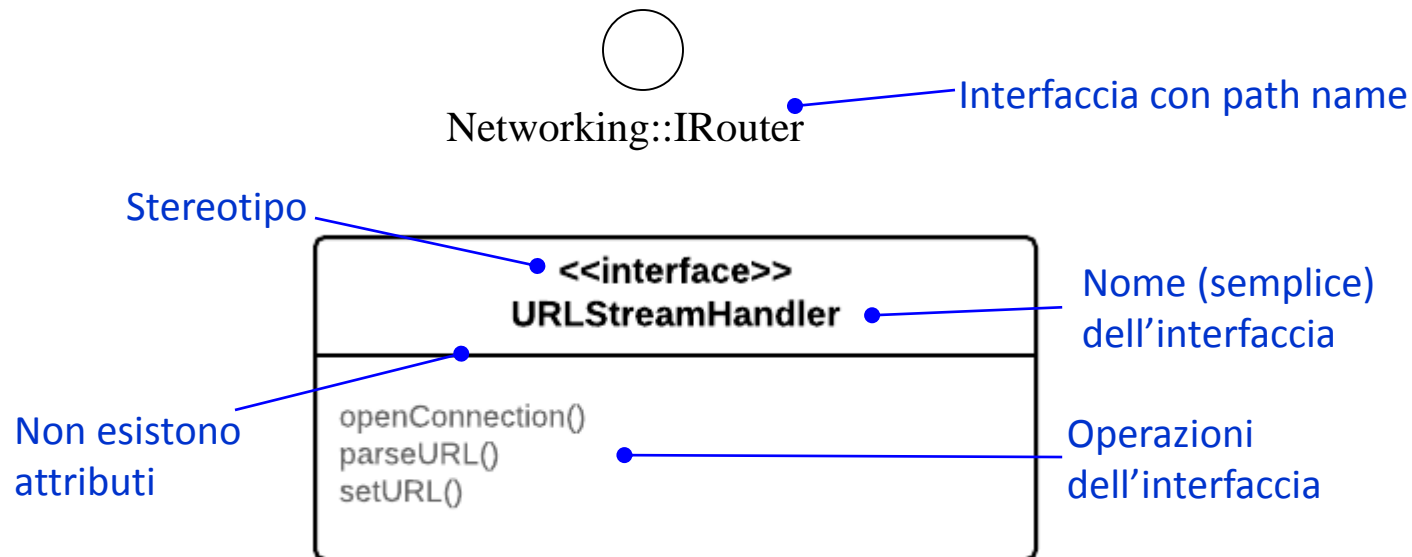
Similmente ad una classe, un'interfaccia può avere un qualsiasi numero di operazioni

Diversamente da una classe, un'interfaccia **non specifica una struttura** (non saranno inclusi attributi, se non statici) e **non fornisce un'implementazione** (non saranno quindi specificati i metodi che implementano le operazioni).

Una interfaccia è **simile** a una classe astratta i cui metodi sono tutti astratti e non dispone di attributi.

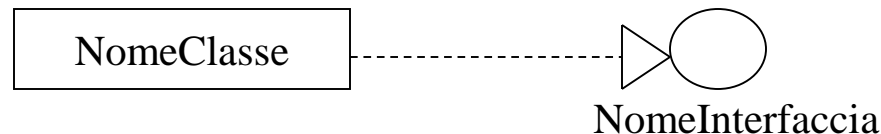
Interfacce

Nella sua rappresentazione più generale, un'interfaccia è rappresentata mediante l'utilizzo di un cerchio, sopprimendo la visualizzazione delle operazioni. Tuttavia, se lo si ritiene importante per la comprensione del modello, è possibile modellare un'interfaccia come una classe stereotipata



Relazione di realizzazione

Una o più classi possono realizzare/implementare le operazioni indicate in una interfaccia. La relazione che si stabilisce fra una interfaccia e una classe che la implementa è detta **relazione di realizzazione**. In UML è indicata con una freccia tratteggiata.



La relazione di realizzazione si presenta come un valido strumento per scindere **la specifica di un “contratto”** (cosa una classe deve implementare) e la sua **implementazione** (come si rendono i servizi del contratto).

La relazione di realizzazione unisce il “cosa” (indicato nell’interfaccia) al “come” (indicato nella classe).

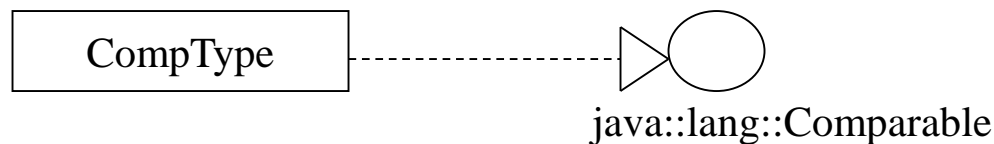
Relazione di realizzazione

Esempio: l'interfaccia *java.lang.Comparable* specifica una sola operazione.

```
public int compareTo(Object o)
```

che prende un oggetto generico *Object* come argomento e restituisce un valore negativo se l'argomento è più piccolo dell'oggetto corrente, zero se è uguale e un valore positivo se è maggiore.

Una qualunque classe che implementa questa interfaccia deve fornire una **implementazione** (un metodo) per *compareTo*.



Interfacce

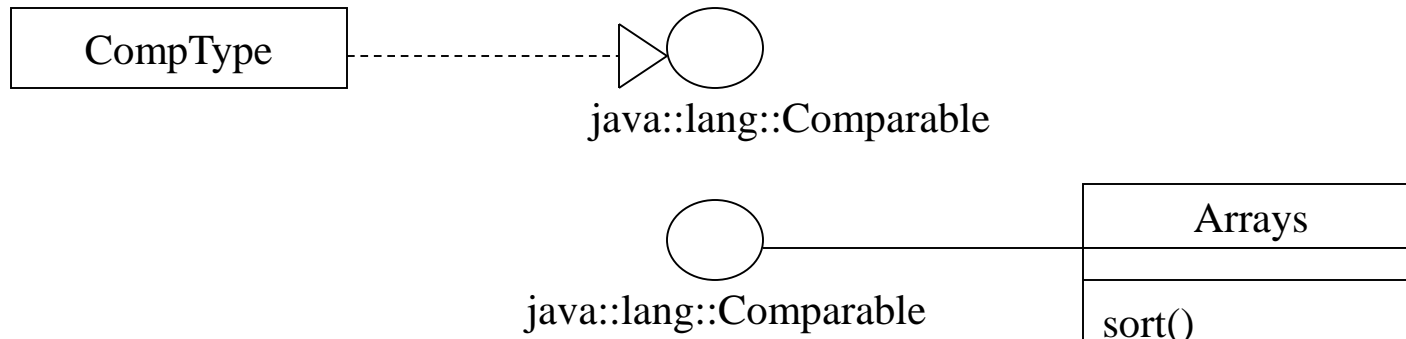
Le interfacce servono a **disaccoppiare la definizione delle operazioni dalla loro implementazione**. Per poter usare un certo oggetto è sufficiente conoscere la sua interfaccia: non serve conoscere l'implementazione.

Ad esempio, il metodo *sort* della classe *Arrays*, che ordina un array di oggetti, necessita solo di sapere che gli elementi dell'array offrono il servizio *compareTo*, in modo da poterli confrontare. Non serve sapere “come” si realizza il servizio, cioè come sono effettivamente confrontati gli oggetti. Si crea così una **dipendenza** di implementazione fra la classe *Arrays* e l'interfaccia *Comparable*.

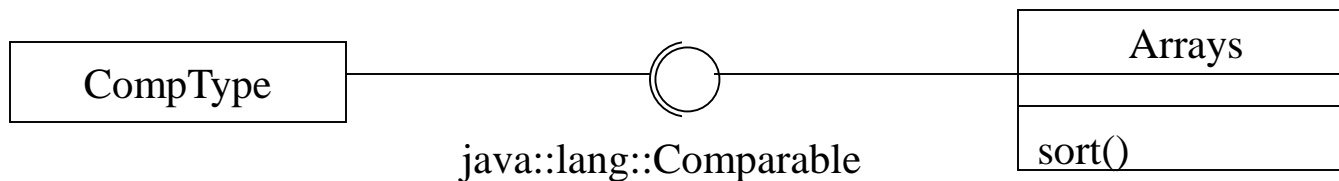


Interfacce

UML permette di rappresentare in modo compatto le seguenti relazioni:

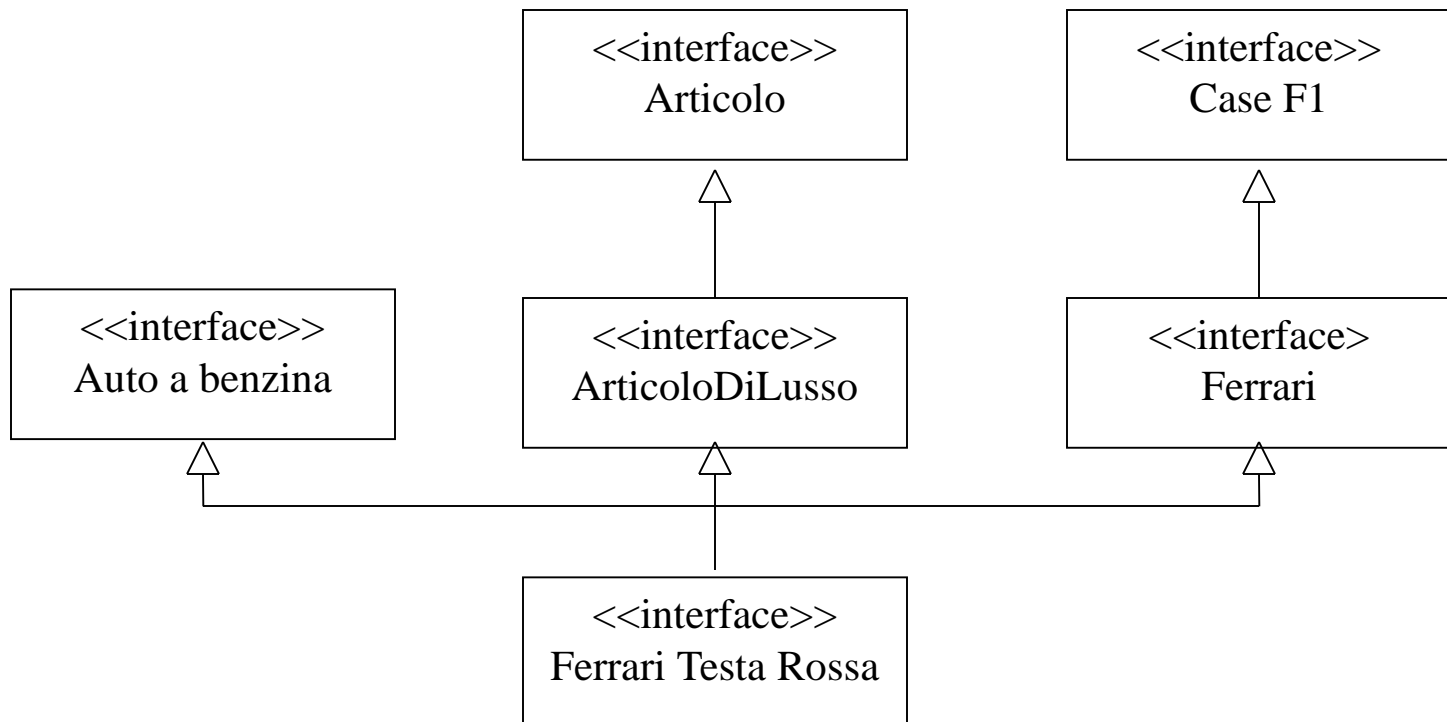


attraverso la notazione:



Interfacce: ereditarietà

Anche **le interfacce possono ereditare da altre interfacce**. Poiché non ci sono implementazioni, la relazione di ereditarietà è naturalmente una relazione di generalizzazione “*is_a*”.



Interfacce: ereditarietà multipla

Poiché non si considerano le implementazioni delle operazioni, l'ereditarietà multipla su interfacce non pone problemi di conflitto di realizzazione

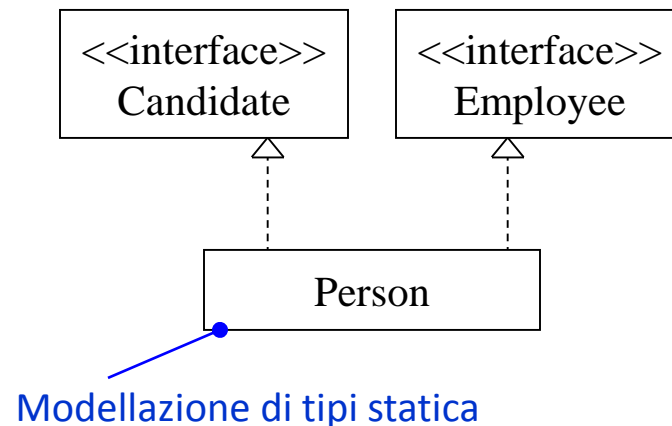
Per questo alcuni linguaggi di programmazione, come Java, permettono l'**ereditarietà singola sulle classi** (in quanto specificano anche le implementazioni) mentre permettono **l'ereditarietà multipla sulle interfacce**

Questo distingue le classi astratte (per le quali l'ereditarietà multipla può porre dei problemi riguardo ai metodi implementati) dalle interfacce

Realizzazione di più interfacce

Poiché non possono sorgere problemi di conflitto di realizzazione, **è permesso a una classe di realizzare più interfacce**, per di più non correlate da una relazione di generalizzazione.

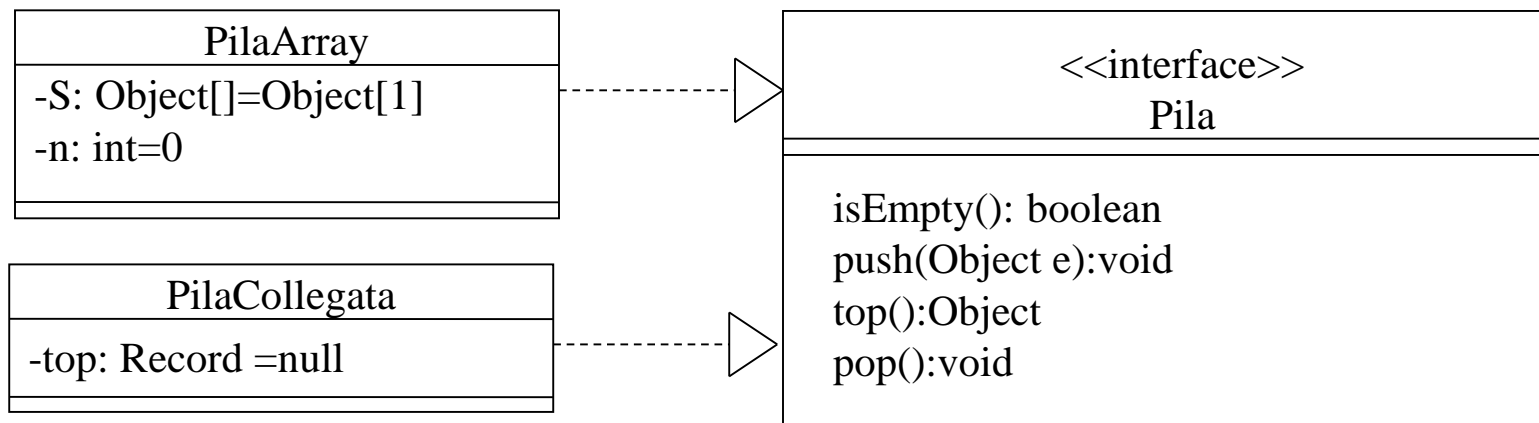
Ad esempio, una classe *Person* può implementare due interfacce:



In questo modo tutto il codice che utilizza *Candidate* e *Employee* può essere utilizzato su oggetti di classe *Person*.

Realizzazioni multiple di una interfaccia

Infine, **più classi possono implementare la stessa interfaccia**. È questo il caso di una interfaccia definita per dati astratti molto utilizzati nello sviluppo del software, come pile, code, liste, alberi e grafi, per i quali sono possibili molteplici realizzazioni



Il codice applicativo che intende fare uso di una pila **dichiarerà delle variabili di tipo *Pila***, svincolandosi dalla specifica realizzazione. Questa sarà specificata solo al **momento della inizializzazione** della variabile. Si garantisce così una forte invarianza ai cambiamenti delle realizzazioni di una pila.

Le metaclassi

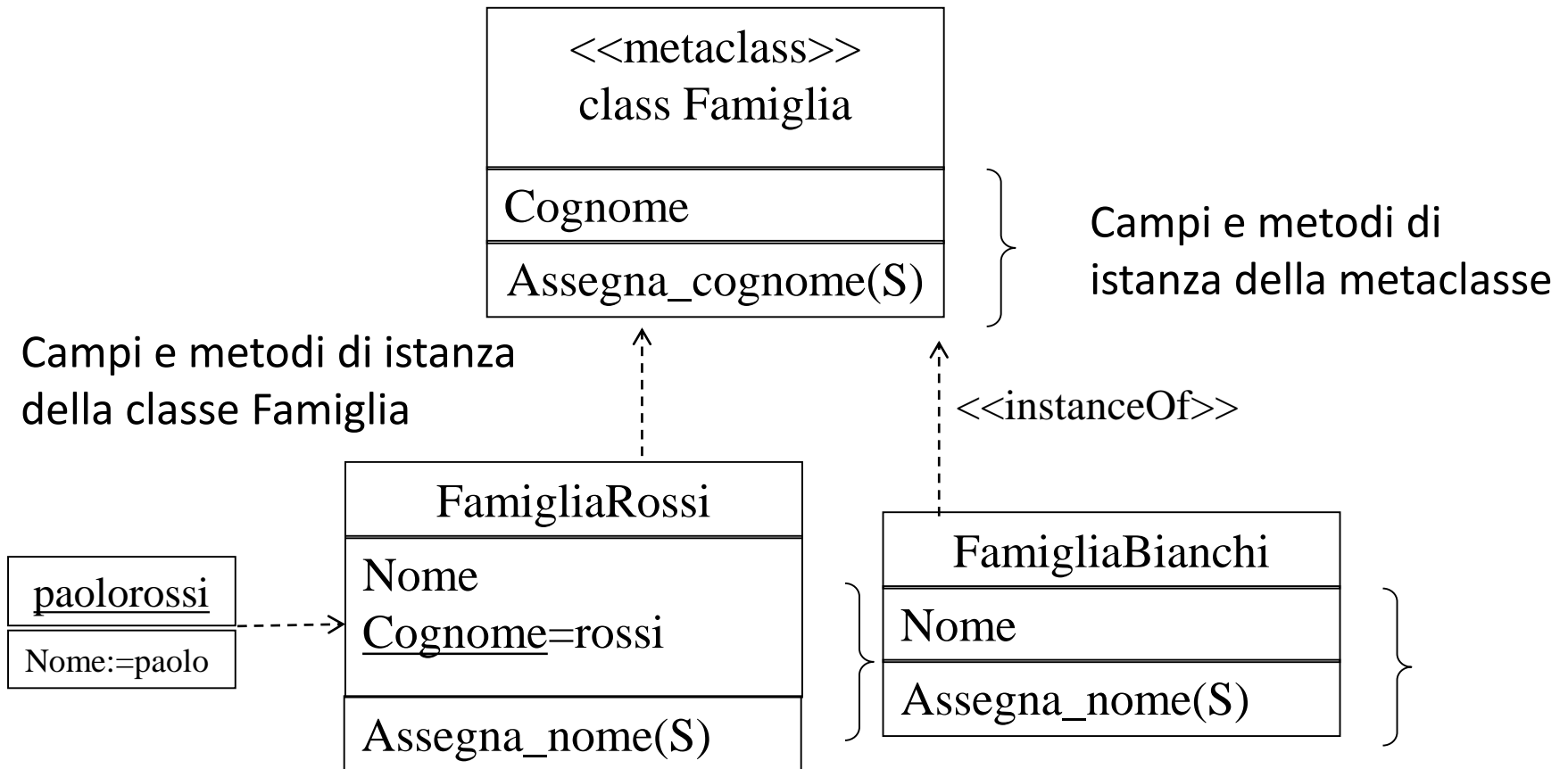
In alcuni modelli object-oriented le classi sono a loro volta degli oggetti ottenuti per istanziazione di una *metaclasses*. Le metaclassi sono classi che definiscono la struttura (dati e operazioni) di altre classi

Esempio

In Smalltalk-80 ogni classe *C* è istanza (unica) di una corrispondente metaclasses di nome ***class C***.

I metodi e gli attributi di classe per una classe *C* sono quindi metodi e attributi di istanza della corrispondente metaclasses ***class C***.

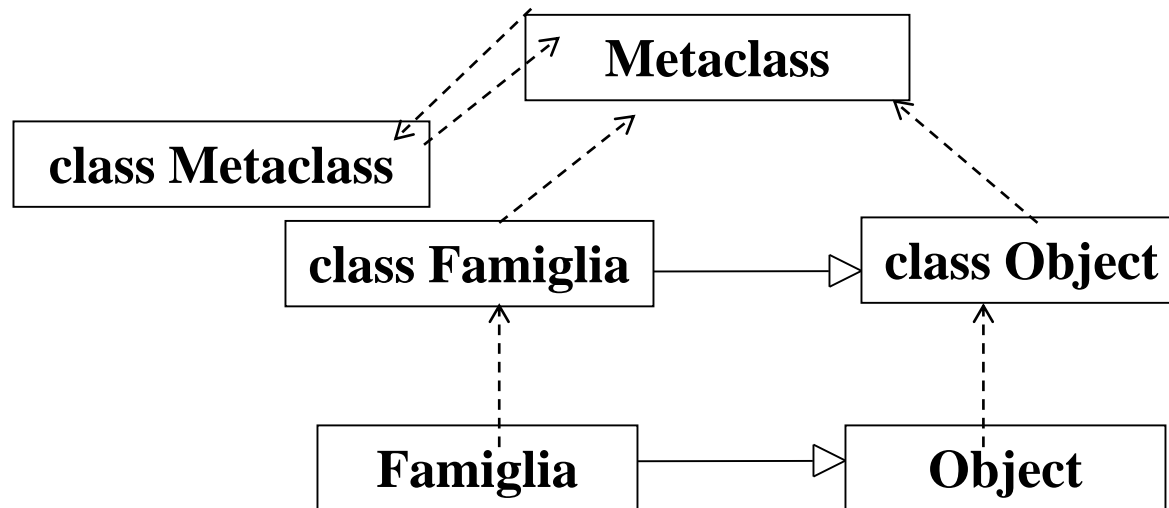
Le metaclassi



I campi e metodi di istanza della metaclassa sarebbero campi e metodi di classe delle classi *FamigliaRossi* e *FamigliaBianchi*.

Le metaclassi

Anche le metaclassi ***class C*** sono degli oggetti in quanto classi. Di quale classe? Di un'unica classe chiamata *Metaclass*. Questa, essendo una (meta-)classe è a sua volta istanza della metaclass *class Metaclass*. Inoltre l'oggetto *class Metaclass* è istanza di *Metaclass*.



Le metaclassi

In Java il modello è differente. Tutte le classi ereditano da *Object*. Inoltre *Object* dispone di un metodo *getClass()* che permette di restituire per ogni oggetto una istanza della classe ***Class*** che descrive la classe di appartenenza dell'oggetto. *Class* è detta **metaclasses**, per analogia con Smalltalk. Tuttavia non è una metaclasses in senso stretto. **I suoi oggetti non sono classi, ma descrivono delle classi.**

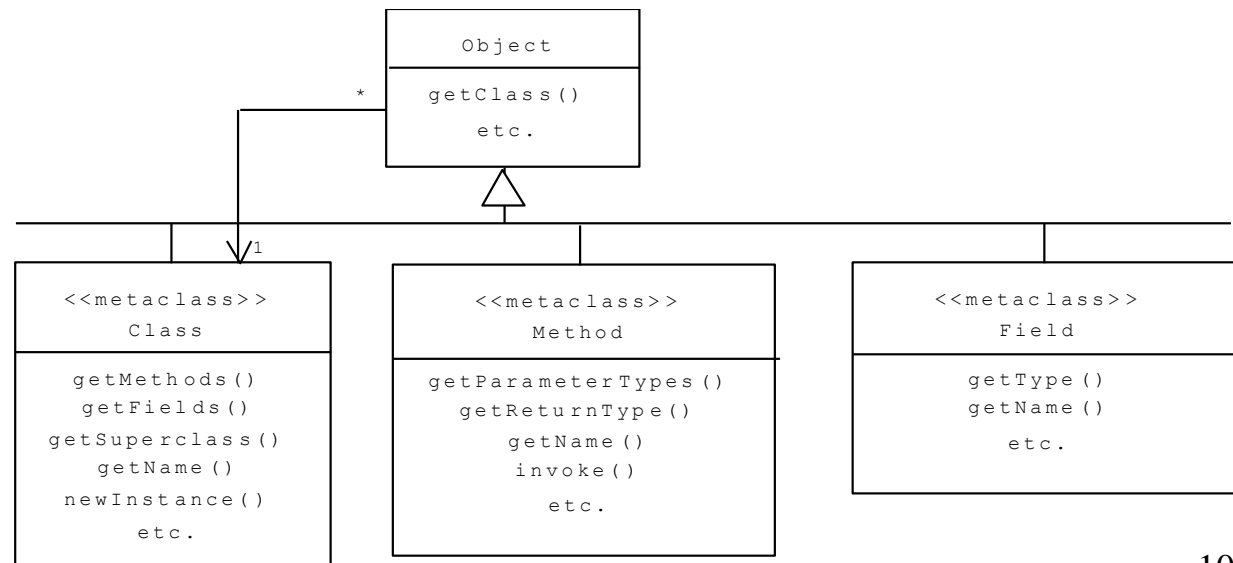
Questo modello serve per realizzare il meccanismo di **riflessione** (*reflection*) che permette a un oggetto di stabilire al **run-time** a quale classe esso stesso appartiene.

Le metaclassi

Infatti, la classe *Class* include dei metodi per scoprire il nome della classe (*getName*), la superclasse (*getSuperClass*), i metodi (*getMethods*), e gli attributi (*getFields*).

Naturalmente questo significa che ci sono anche oggetti per rappresentare i metodi e i campi. Questi appartengono rispettivamente alle classi *Method* e *Field*. *Class*, *Method*, e *Field* sono sottoclassi di *Object* come tutte le classi.

<<metaclass>> è lo stereotipo usato in UML per indicare una metaclass.



Aggregazione di oggetti

L'ereditarietà offre molti vantaggi, ma non tutti gli oggetti si ottengono bene **derivandoli** da altri oggetti. Spesso un oggetto è ottenuto **aggregando** altri oggetti.

Esempio: Ricorrendo all'ereditarietà multipla è allettante definire una classe *Automobile* partendo dalle classi *Carrozzeria*, *Sedile*, *Ruota* e *Motore*. Si tratta di un errore concettuale poiché l'ereditarietà multipla permette di definire due oggetti per **fusione** e non per **composizione**. Un'automobile è composta da una carrozzeria, da un motore, da sedili e da ruote; **ma il suo comportamento non è in alcun caso l'unione dei comportamenti** di queste differenti parti.

Aggregazione di oggetti

Nell'esempio specifico, sembrerebbe che il ricorso alla ereditarietà di implementazione sia la soluzione.

L'ereditarietà di implementazione permette il riuso della sola implementazione (attributi e metodi) ma non dei comportamenti.

Ma neanche questa forma di ereditarietà è adatta allo scopo. Infatti ereditando dalla classe *Ruota* si ha la possibilità di rappresentare e manipolare una sola ruota, mentre un'automobile ha quattro ruote.

Aggregazione di oggetti

Una **composizione di oggetti** può essere rappresentata **permettendo alle variabili di istanza di una classe di puntare a oggetti di altre classi**. Infatti si era detto che:

“lo stato di un oggetto può anche contenere il riferimento ad un altro oggetto. Si dice che un oggetto punta ad un altro. Il puntamento è asimmetrico.”

Esempio: una classe che descrive strutturalmente un'automobile è provvista di variabili d'istanza che prendono per valore i riferimenti alle istanze di *Ruota*, *Motore*, *Sedile* e *Carrozzeria* corrispondenti.

Si possono stabilire legami con più istanze di una classe che descrive un componente (per esempio, più ruote).

Aggregazione di oggetti

La relazione che si stabilisce in questo modo fra le classi è detta di **aggregazione** o **composizione** (o relazione “*has_a*”).*

Essa è quindi un'altra possibile relazione fra le classi, diversa dall'ereditarietà (e dalla generalizzazione “*is_a*”).

Una classe A si dice essere in relazione di aggregazione con una classe B quando alcune istanze di B contribuiscono a formare una parte delle istanze di A.

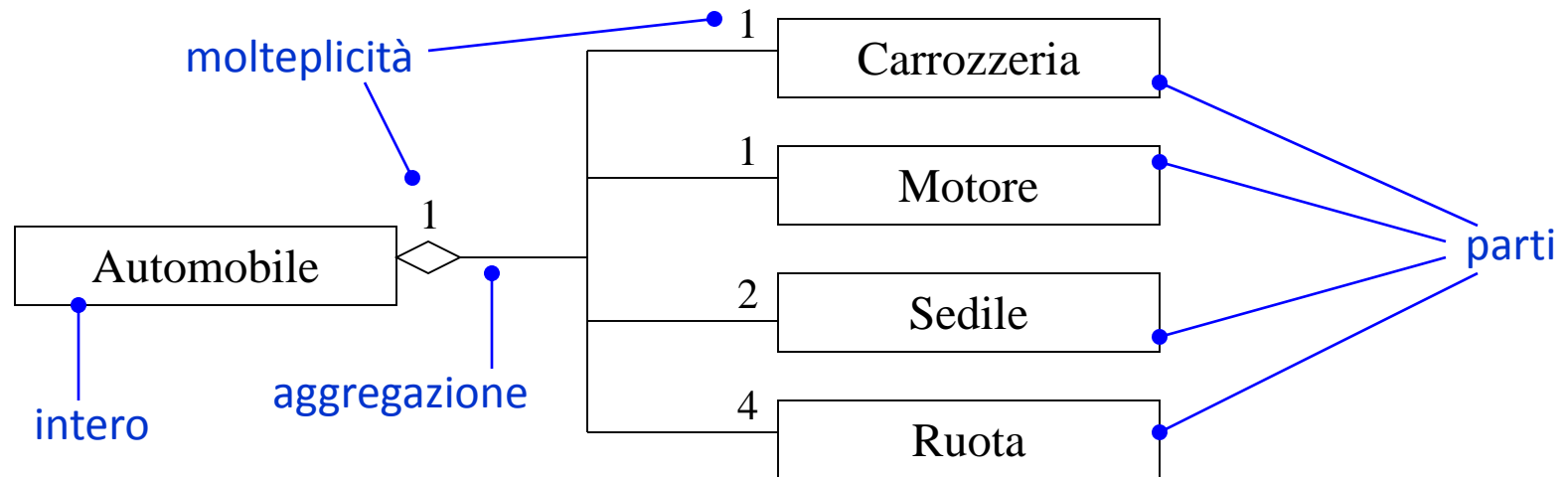
L'aggregazione è, come l'ereditarietà, una relazione asimmetrica.

* La differenza fra aggregazione e composizione sarà chiarita in seguito.

Aggregazione: notazione

In UML l'aggregazione è resa mediante una linea piena e un piccolo rombo dalla parte del contenitore.

Esempio:



La ***molteplicità*** consente di indicare quanti oggetti possono essere aggregati.

Aggregazione: uso


Grady Booch suggerisce l'uso dell'aggregazione nelle seguenti situazioni:

1. **Contenimento fisico**: la pagina di un libro.
2. **Appartenenza** (*membership*): il giocatore di una squadra di calcio
3. **Composizione funzionale**: le ruote di un'automobile.

Si deve osservare che **l'aggregazione non implica una dipendenza esistenziale**: un'automobile può essere distrutta (rottamata) ma alcune sue parti possono essere riutilizzate. Una squadra di calcio può fallire, ma i suoi giocatori non vengono “soppressi”.

La composizione

Le aggregazioni sono associazioni **deboli** fra parti e intero. Questo significa che le parti possono esistere senza l'intero.

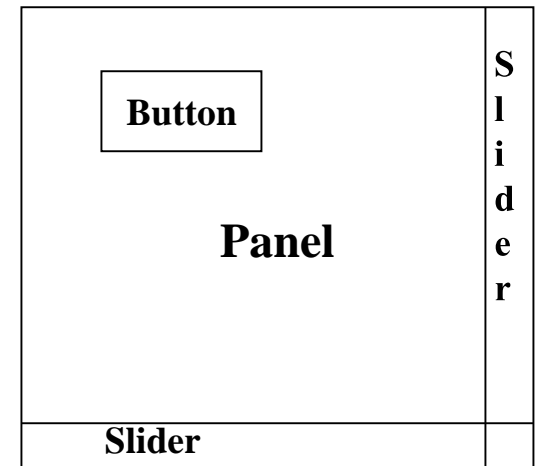
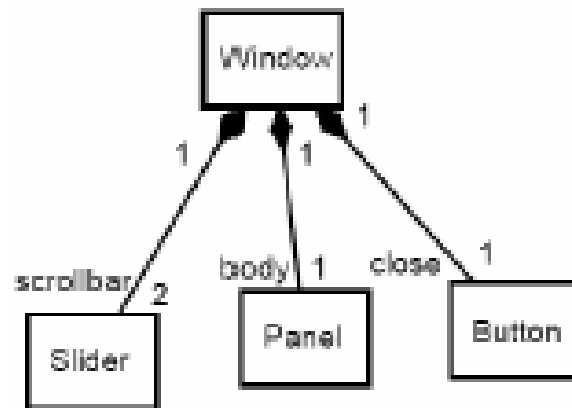
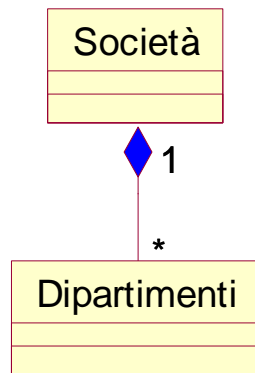
Un'associazione **forte** fra parti e intero è detta **composizione** e si rappresenta in UML mediante un rombo pieno (). La composizione comporta una **dipendenza esistenziale**, in quanto le parti non esistono senza il contenitore. Ciò presuppone che:

1. Creazione e distruzione delle parti avvengano nel contenitore
2. I componenti non siano parti di altri oggetti

La composizione

In questi esempi, la vita delle “parti” che compongono l’“intero” dipende dalla vita dell’intero stesso:

1. i dipartimenti non esisterebbero se non esistesse la società;
2. il tempo di vita dei diversi oggetti grafici d’interazione (pulsante, scrollbar, pannello) dipende da quello della finestra che li contiene.



La composizione

Come interpretare questo diagramma?



Un'istanza di *Punto* può essere parte di un poligono oppure il centro di un cerchio, ma non entrambe le cose.

Regola di “non condivisione”: benché una classe possa essere componente di molte altre classi, ogni sua istanza può essere componente di un solo oggetto.

Un diagramma delle classi può mostrare più classi di potenziali possessori di oggetti componenti, ma ogni istanza di componente deve appartenere a un solo oggetto possessore.

Questa regola è caratterizzante della composizione.

La composizione



La composizione fra classi stabilita in fase di progettazione offre delle informazioni importanti al programmatore che andrà a implementare le classi.

Ad esempio, se il programmatore utilizzerà il C++ farà sì che gli oggetti componenti finiscano nello stack e non nello heap (**non userà puntatori/new**).

Se utilizza il Java, che pone tutto nello heap, si preoccuperà di allocare i componenti nella classe contenitore e di non passare mai all'esterno della classe il riferimento ai componenti, in modo da impedirne la condivisione con oggetti di altre classi (**il componente sarà privato e non ci sarà alcun metodo che restituisce il suo riferimento**).

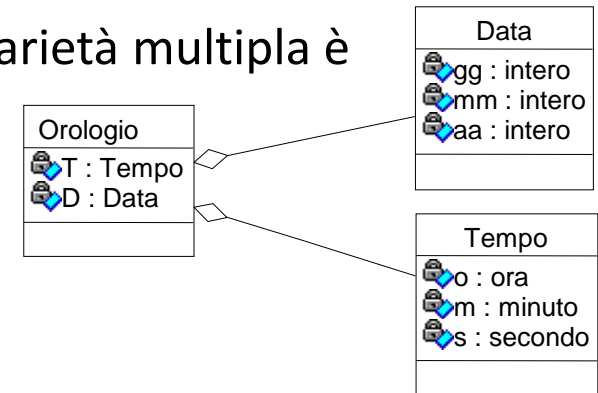
Ereditarietà vs. aggregazione

In molti casi è possibile associare due classi mediante ereditarietà o aggregazione/composizione e la scelta non è immediata come negli esempi visti precedentemente.

Esempio

Si supponga di disporre di una classe *Data* e di una classe *Tempo*, che rappresentano e manipolano, rispettivamente, date e istanti di tempo. Volendo definire una nuova classe *Orologio* per rappresentare sia le date che gli istanti di tempo, si può:

1. derivare *Orologio* da *Data* e da *Tempo*, se l'ereditarietà multipla è permessa.
2. Comporre *Orologio* con le due classi.



Sono anche possibili soluzioni ibride (ereditarietà da una classe + composizione con l'altra).

Ereditarietà vs. aggregazione

Differenze:

- Nell'ereditarietà, ogni oggetto *o* della classe *Orologio* contiene tutti i campi definiti nelle classi *Data* e *Tempo* e può accedervi direttamente (se non sono privati).
- Nell'aggregazione/composizione, ogni oggetto *o* della classe *Orologio* contiene due campi di tipo *Data* e *Tempo* rispettivamente. Per accedere all'informazione sul giorno occorrerà invocare un metodo (per esempio *day()*) sull'oggetto *D*.
- Se l'**ereditarietà** corrisponde a una relazione di generalizzazione (is_a) **vale il polimorfismo di inclusione**, cioè ogni oggetto di *Orologio* è utilizzabile come una istanza di *Data* e di *Tempo*, **mentre questo non è vero nel caso della composizione.**

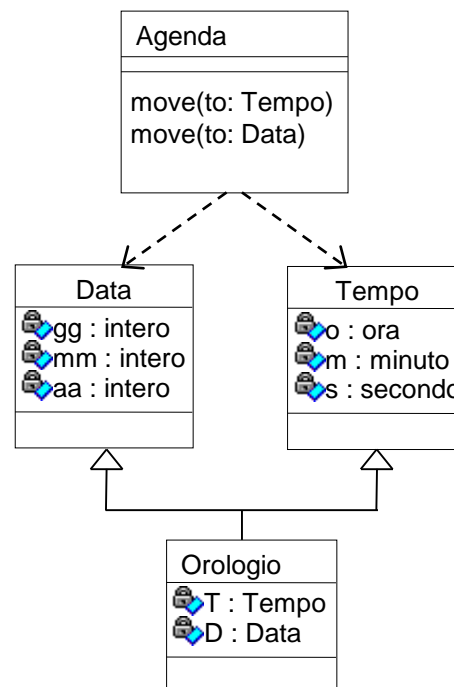
Ereditarietà vs. aggregazione

- Le scelte di progetto sono totalmente diverse.
- L'ereditarietà permette di poter “dimenticare” il fatto che **gg** è definito nella classe **Data** e di utilizzarlo direttamente come campo di **Orologio** (sempre che sia definito come `protected` e non come `private`)
 - questa caratteristica della derivazione è particolarmente utile se si vogliono utilizzare più derivazioni successive e creare vere e proprie gerarchie di classi. In questo caso non è necessario conoscere a quale livello della gerarchia è definito un dato campo, ma è possibile utilizzarlo direttamente come se fosse un elemento della classe stessa.

Ereditarietà vs. aggregazione

L'ereditarietà permette di riutilizzare tutti i metodi delle varie classi che operano su oggetti delle classi *Data* e *Tempo*.

Esempio: *Agenda* ha due metodi che possono operare anche su istanze di *Orologio*. Il codice dei metodi polimorfi *move* è riutilizzabile per istanze di *Orologio*.



Ereditarietà vs. aggregazione

Il **meccanismo di aggregazione/composizione** è generalmente usato quando si vogliono utilizzare i servizi di una classe predefinita ma **non la sua interfaccia**.

L'ereditarietà di implementazione, qualora non dovesse essere permessa da un linguaggio di programmazione, potrebbe essere resa da una relazione di aggregazione/composizione.

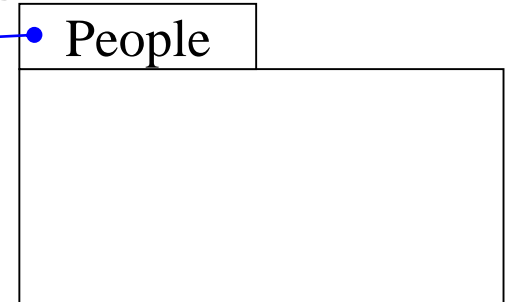
Raggruppare classi

La mole di classi aumenta in modo considerevole all'aumentare della complessità del sistema da modellare. È importante, quindi, organizzare tali classi in **gruppi** separati al fine di rendere più facilmente individuabili, e quindi accessibili, le singole parti che compongono il nostro sistema.

I **package** sono un meccanismo generale per organizzare le classi in gruppi.

Nome del package

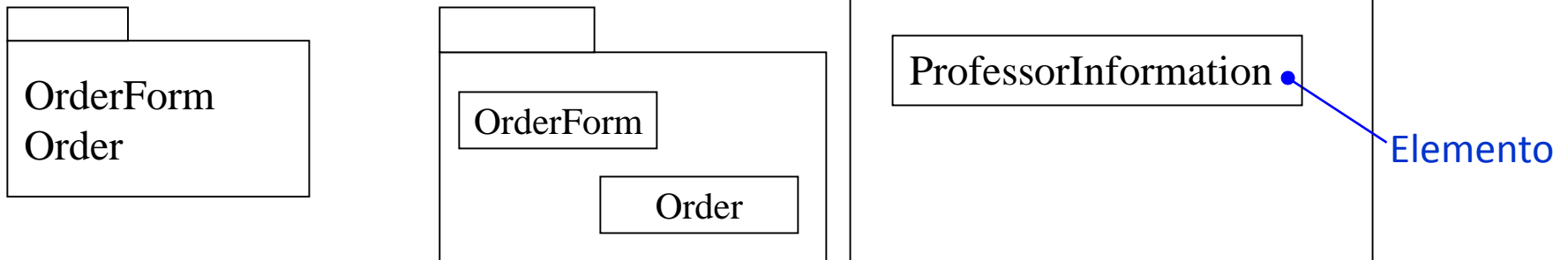
In UML un package è rappresentato come



In genere i package si usano per riunire classi, ma nella notazione UML essi possono includere qualsiasi costrutto UML e possono essere persino eterogenei (ad esempio, possono contenere classi e interfacce).

Raggruppare classi

É consentito mostrare il contenuto di un package sia testualmente
sia graficamente:

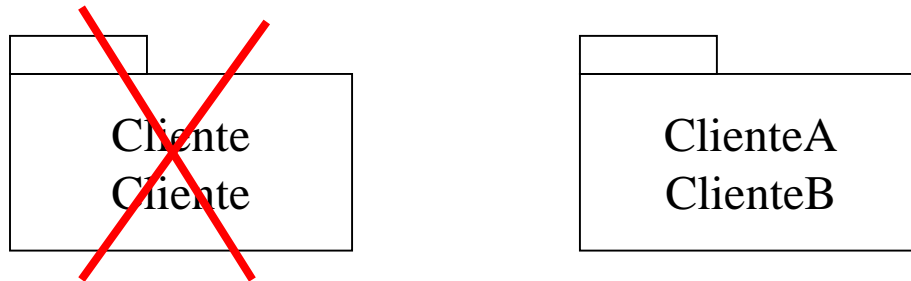


Un package definisce un *namespace* (spazio degli identificatori) per i suoi elementi. Questo significa che ogni classe di un package di classi deve avere un nome distinto all'interno del package che la racchiude.

Il nome completo (o *qualificato*) della classe sarà ottenuto indicando prima il nome del package che la contiene mediante una notazione `::` (ad esempio, *People::ProfessorInformation*).

Raggruppare classi

Elementi della stessa specie, inseriti all'interno di uno stesso package, devono avere necessariamente nomi differenti



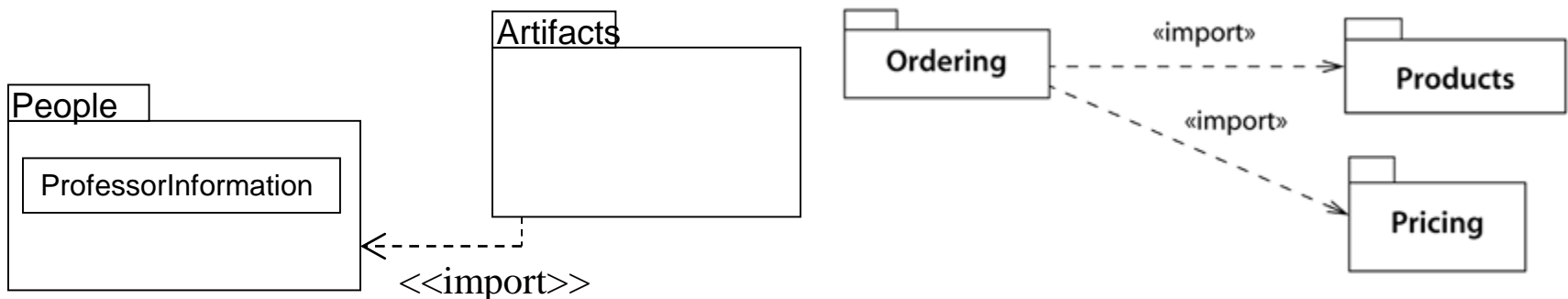
Al contrario é possibile avere nomi uguali per elementi di tipo differente. Ad esempio è possibile avere una interfaccia *Cliente* e una classe *Cliente*.

É possibile avere nomi uguali per elementi della stessa specie se sono inseriti in package differenti.

Raggruppare classi

Per evitare la necessità di utilizzare nomi qualificati, un package può **importare** gli elementi o il contenuto di un altro package nel proprio *namespace*. Un elemento nel package che importa può quindi riferirsi a un elemento importato come se esso fosse definito direttamente nel package (localmente).

L'import di un package viene ottenuto stereotipando la relazione di dipendenza.



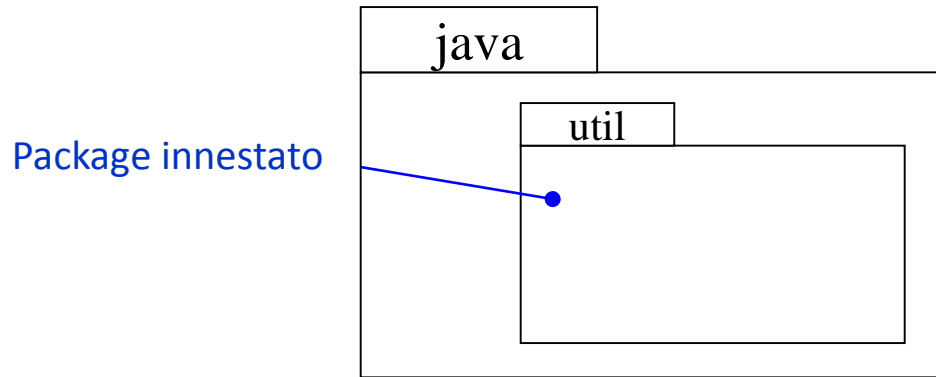
Raggruppare classi

Nota bene:

- La relazione di importazione non è transitiva. Se A importa B e B importa C, allora A può usare (senza qualificarli) gli elementi di B e B può usare (senza qualificarli) gli elementi di C, ma non è detto che A possa usare (senza qualificarli) gli elementi di C (a meno che A non importi anche C).
- Se c'è conflitto di nomi in due elementi importati, nessuno dei due elementi è aggiunto al namespace.
- Se il nome di un elemento importato è in conflitto con il nome di un elemento definito localmente (internamente) a un package, il **nome dell'elemento interno ha precedenza** sul nome importato che non viene aggiunto al namespace (sarà sempre raggiungibile mediante qualificazione).

Raggruppare classi

I package possono essere **innestati** senza alcun limite di profondità



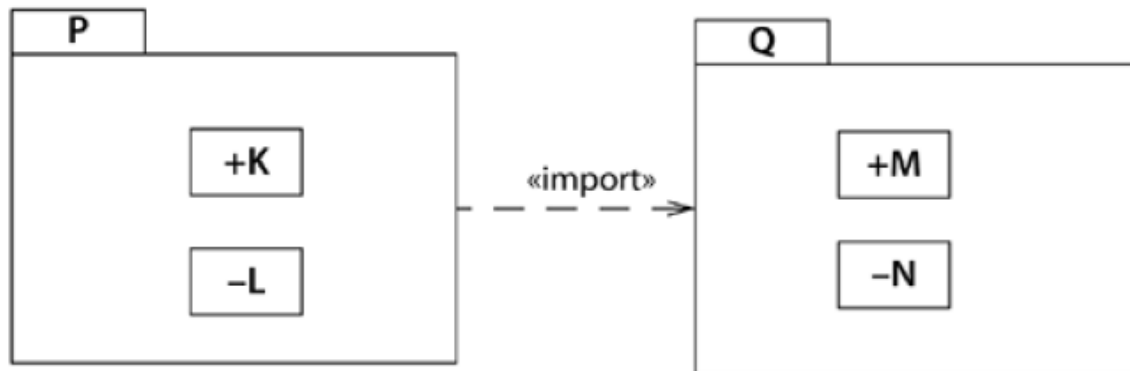
Un package innestato ha accesso a tutti gli elementi contenuti direttamente nei package esterni (a qualunque livello di annidamento), senza necessità di importazione.

Raggruppare classi

Si può anche specificare la **visibilità** degli elementi di un package:

- **Public** (+) sono visibili ad altri elementi del package stesso, a uno dei package innestati o a package che li importano.
- **Private** (-) non sono visibili all'esterno del package.

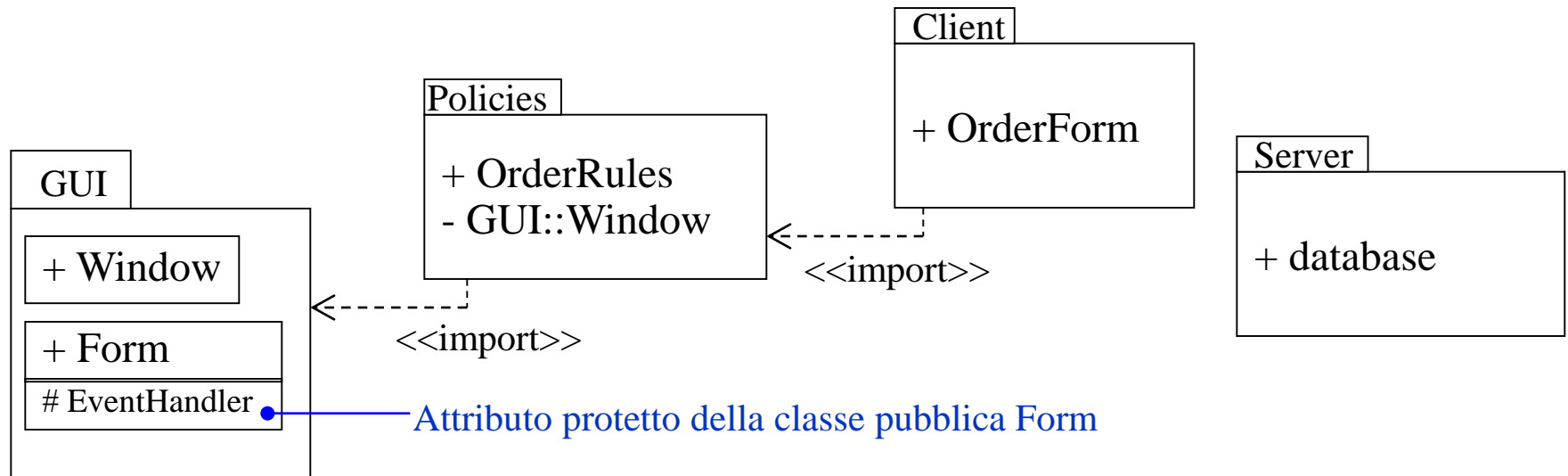
Esempio



Il package P importa il package Q. Le classi K e L nel package P possono usare M come nome di classe senza qualificarlo (::) poiché M è pubblica, ma non possono vedere la classe N, in quanto privata. Le classi M e N nel package Q possono usare il nome qualificato P::K per referenziare la classe pubblica K nel package P, ma non possono vedere in alcun modo la classe L.

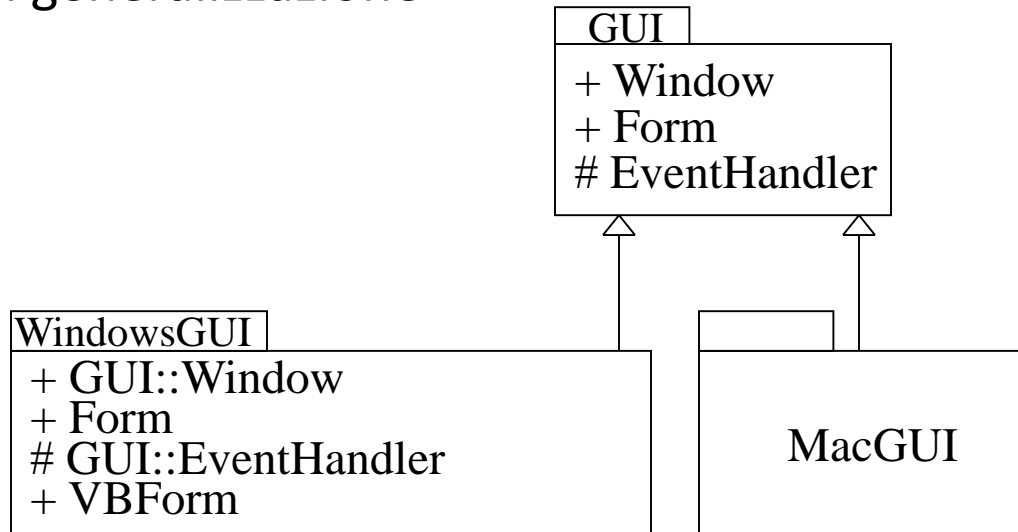
Raggruppare classi

La relazione di importazione può specificare la visibilità (pubblica o privata) di ciò che è importato. Se è pubblica, l'elemento importato è visibile a un qualunque altro elemento che può vedere il package che importa. Se è privata, l'elemento importato non è visibile al di fuori del package che importa



Raggruppare classi

Oltre alla relazione di import i package possono essere legati da una relazione di generalizzazione



Sia *WindowsGUI* che *MacGUI* ereditano sia gli elementi public che protected di *GUI*. Inoltre, *WindowsGUI* rimpiazza *Form* e include un nuovo elemento *VBForm*.

Si noti che l'ereditarietà per i package è del tutto simile all'ereditarietà per le classi

Classi interne

Una classe interna (*inner class*) è una classe la cui dichiarazione si trova all'interno di un'altra classe ospite (*top level class*).

Le classi interne sono identificate da un nome al pari delle classi top level.

Una classe interna può essere privata. In tal caso non è visibile all'esterno della classe ospite. (*)

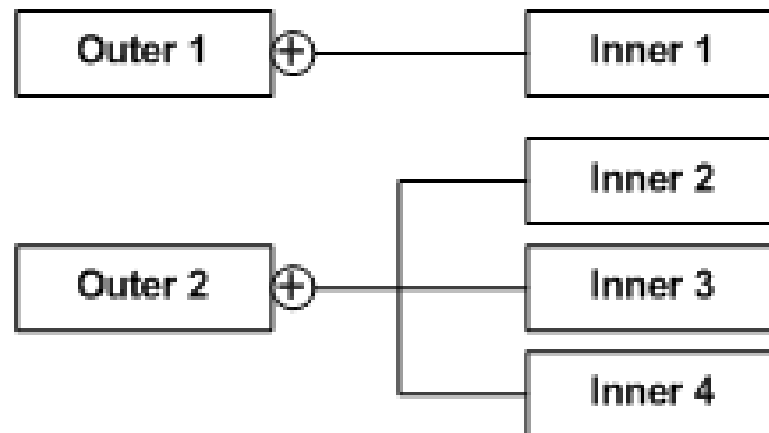
La inner class può accedere a tutti i metodi e i campi della classe ospitante, mentre la classe ospitante può vedere solo la parte pubblica della inner class.

Un oggetto di classe inner non può esistere se non esiste un oggetto della classe ospitante. Una classe interna non può avere campi statici.

(*) In Java, una classe *top level* non può mai essere privata.

Classi interne

UML non ha una notazione standard per le classi interne.
Allen Holub suggerisce la seguente notazione:



<http://www.holub.com/goodies/uml/index.html>

Il polimorfismo

Con questo termine si intende la possibilità di **associare a una operazione diverse realizzazioni** (o *morfismi*).

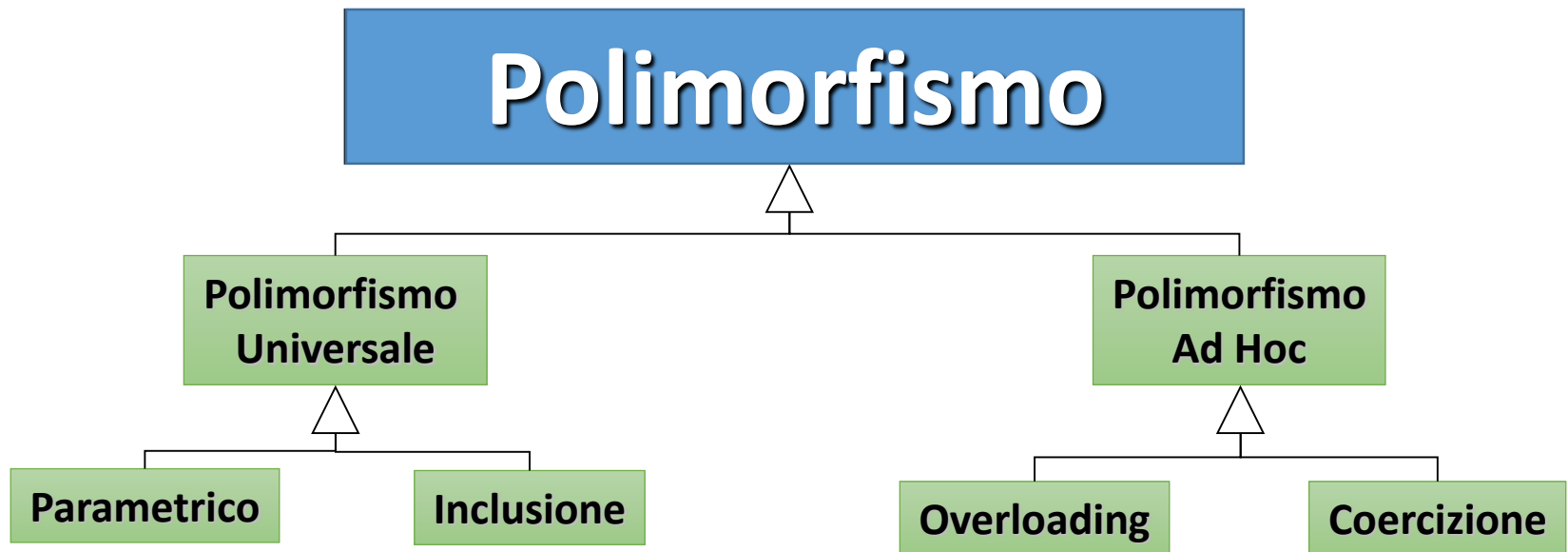
Il concetto di polimorfismo è stato definito in vari modi.

Strachey (1967) ha distinto informalmente due tipi principali di polimorfismo:

- **polimorfismo parametrico**: è ottenuto quando una funzione lavora uniformemente su una gamma di tipi. Questi tipi normalmente esibiscono una qualche struttura comune.
- **polimorfismo ad hoc**: è ottenuto quando una funzione lavora, o sembra lavorare, su tipi differenti (che potrebbero non esibire una struttura comune) e potrebbe comportarsi in modo totalmente differente per ciascuno di essi.

Il polimorfismo

Una classificazione più raffinata dei diversi meccanismi di polimorfismo è data da *Cardelli e Wegner* alla metà degli anni '80.



L. Cardelli, P. Wegner: On understanding types, data abstraction, and polymorphism. ACM Computing Surveys, vol. 17, no. 4, pp. 471-522, December 1985

Il polimorfismo

La classificazione di Cardelli e Wegner introduce una nuova forma di polimorfismo, quello *per inclusione*, al fine di modellare i concetti di sottotipo e di ereditarietà.

Il polimorfismo per inclusione e parametrico sono classificati come due sottocategorie del *polimorfismo universale*, che è posto in contrasto al polimorfismo non universale o ad hoc.

L'idea di polimorfismo universale è quella di poter operare su un numero infinito di tipi, a patto che essi rispettino alcuni vincoli.

Il polimorfismo

Universale:

- Il polimorfismo è su un numero potenzialmente illimitato di tipi
- I diversi morfismi sono generati automaticamente
- C'è una base unificante, comune a tutti i diversi morfismi che può assumere l'entità polimorfa

Ad hoc:

- Il polimorfismo è su un numero finito di tipi, spesso pochissimi
- I diversi morfismi sono generati in modo manuale o semi-manuale
- Non c'è una base comune a tutti i morfismi, al di là delle intenzioni del progettista.
 - L'uniformità è un caso, non la regola

La coercizione

La **coercizione** è il meccanismo di conversione implicita operata da un compilatore per applicare un operatore definito per oggetti di tipo **T1** anche a oggetti di tipo **T2**.

Esempio:

$$3.14 + 5$$

L'operatore $+$ è definito per valori reali, ma lo si può usare su un insieme di tipi più grande di quello per il quale è stato definito.

Senza coercizione, avremmo un errore di tipo.

La coercizione

Altri esempi di coercizione sono, in Java, l'**autoboxing** di un `int` in un `Integer` e l'**unboxing** di un `Integer` in un `int`.

```
int i;
```

```
Integer j;
```

```
i = 1;
```

```
j = 2;
```

```
i = j;
```

```
j = i;
```

La coercizione

Le **coercizioni** possono essere stabilite staticamente, inserendole automaticamente fra gli argomenti e le funzioni al momento della compilazione, oppure potrebbero essere determinate dinamicamente da test al run-time sugli argomenti.

La coercizione è la forma di polimorfismo più semplice. Essa opera a un livello **semantico**, cioè cambiando la rappresentazione del dato.

Pascal e Ada supportano questa modalità di polimorfismo. Tuttavia in questi linguaggi il polimorfismo è più un'eccezione che la regola. Pertanto sono considerati per lo più monomorfi.

L'overloading

Si ha polimorfismo per **overloading** quando si usa lo stesso identificatore per metodi differenti e si ricorre a informazione di contesto per decidere quale metodo è denotato da una particolare occorrenza dell'identificatore.

La disambiguazione necessaria per una corretta compilazione si basa sul tipo degli argomenti del metodo o sulla classe dell'oggetto a cui si richiede il servizio.

Possiamo immaginare che una pre-compilazione del programma potrebbe disambiguare ed eliminare l'overloading dando nomi differenti a metodi differenti. In questo senso, l'overloading è giusto una conveniente abbreviazione **sintattica**.

L'overloading

L'overloading è presente nella maggior parte dei linguaggi (anche imperativi), dove gli operatori aritmetici $+$, $*$, $-$ etc. sono applicabili a più di un tipo.

Esempio: In Pascal, l'operatore: $+$, se applicato a 2 interi genera una istruzione *add* fra interi, mentre se applicato a 2 numeri reali genera una istruzione *add* fra floating point.

In Ada, come in C++, è possibile ridefinire alcuni operatori (ad esempio, $+$) per operare su tipi/classi definite dall'utente. Ad esempio, l'operatore $+$ può essere ridefinito per numeri complessi, rappresentabili mediante un record.

L'overloading

// C++'s user-defined overloading of the + operator:

```
class Rational {  
public: Rational(double);  
const Rational& operator + (const Rational& other);  
...  
};
```

Overloading
di operatori

// C++'s user-defined overloading of the function name max:

```
double max(double d1, double d2);  
char max(char c1, char c2);  
char *max(char *s1, char *s2);  
const char *max(const char *s1, const char *s2);
```

Overloading
di funzioni

L'overloading

L'overloading può efficacemente interagire con la coercizione.

Ad esempio, l'operatore `+` può essere definito per due interi e per due reali, ragione per cui le espressioni:

`3 + 4`

`3.0 + 4.0`

non causano errore di tipo.

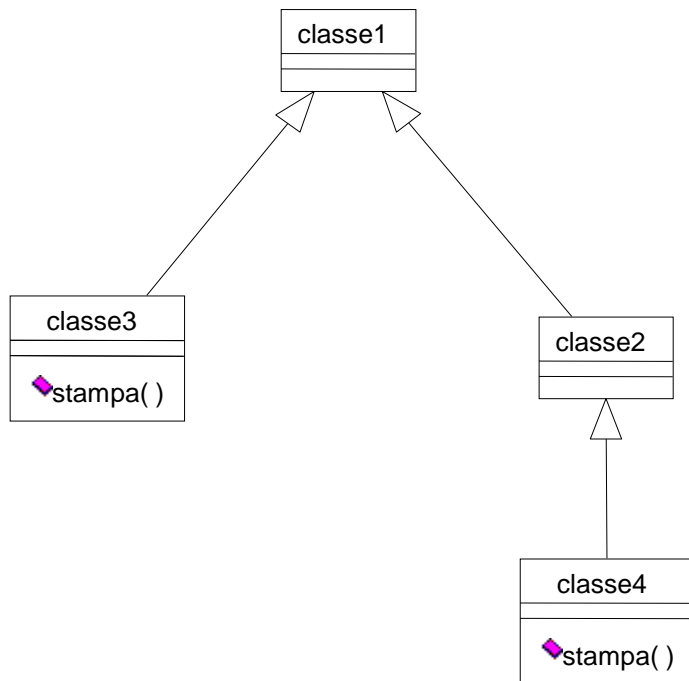
Combinando l'overloading di `+` con il meccanismo di coercizione, non avremo errore di tipo anche nei seguenti casi:

`3 + 4.0`

`3.0 + 4`

L'overloading

Nel paradigma a oggetti si ha overloading anche nel caso di funzioni con medesimo nome ma definite in classi non correlate gerarchicamente.



In questo esempio l'informazione di contesto utilizzata per disambiguare è la classe di appartenenza dell'istanza sulla quale si invoca il metodo stampa.

Questo polimorfismo è meno potente di quello universale, perché l'unicità del nome del metodo nasconde una molteplicità di implementazioni.

Il polimorfismo parametrico

Nel polimorfismo parametrico, una funzione polimorfa ha un parametro di tipo esplicito o implicito, che determina il tipo dell'argomento per ciascuna applicazione della funzione.

Le funzioni che esibiscono il polimorfismo parametrico sono anche dette *funzioni generiche*.

Una funzione generica può lavorare su argomenti di molti tipi, generalmente esibendo lo stesso comportamento indipendentemente dal tipo dell'argomento.

Esempio polimorfismo parametrico

```
/*funzione che calcola la lunghezza di una lista qualunque*/  
length(x)= if (x=nil) then 0 else (1+length(tail(x)))
```

La funzione *length* ha lo stesso comportamento, indipendentemente che si tratti di liste di interi, liste di reali, liste di liste, ecc.

Il polimorfismo parametrico

Le funzioni *generic* dell'Ada ma anche le funzioni *template* del C++ sono un esempio di funzioni generiche.

```
template<class T> void sort (vector<T>);  
{ ... def. di ordinamento generico su un vettore di dati di tipo T ... }
```

```
vector <complex> cv(100);  
vector <int> ci(200);  
void f(vector <complex> &cv, vector<int> &ci)  
{  
    sort(cv);  
    sort(ci);  
}
```

Il polimorfismo parametrico

Esempio di classe stack generica in C++:

```
template<class ITEM>
class Stack {
private: ITEM
    mem[MAXSIZE];
    int topp;
public:
    Stack(void) { topp=-1; }
    int empty(void) { return topp<0; }
    int full(void) { return topp>=MAXSIZE; }
    ITEM top(void) { if( empty()) fatal("stack empty"); return mem[topp]; }
    void pop(void) { if( empty()) fatal("stack empty"); topp=topp-1; }
    void push(ITEM c) { if( full()) fatal("stack full"); topp = topp+1;
    mem[topp] = c; }
};
```

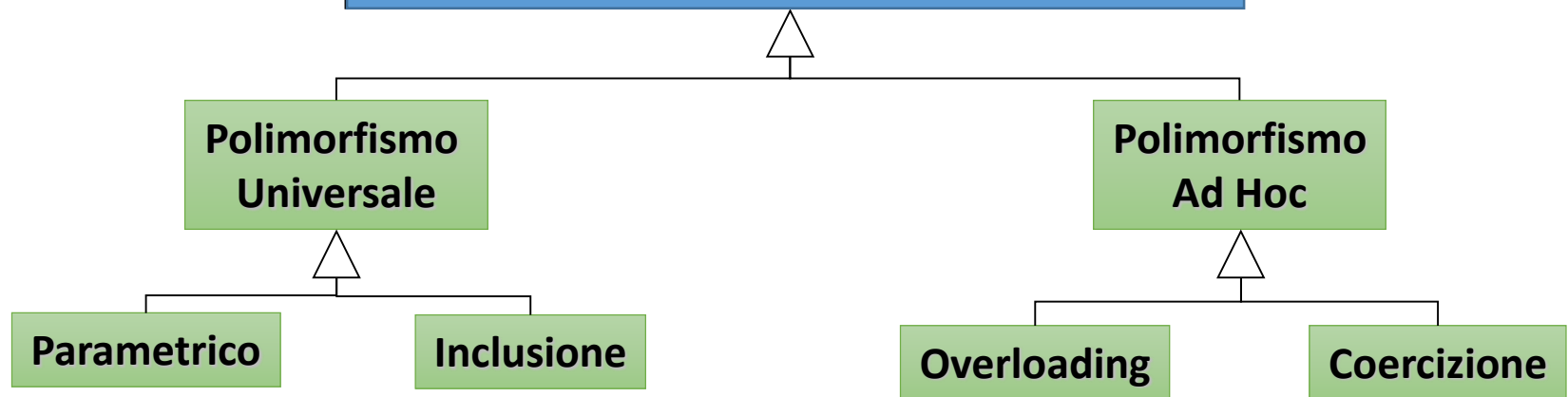
Il polimorfismo parametrico

Per usare la classe `stack` occorre istanziarla. In C++ ciò accade al momento della dichiarazione di un oggetto:

```
main() {  
    Stack<int> istk;  
    Stack<char *> sstk;  
    ...  
}
```

`int` e `char` sono parametri di tipo.

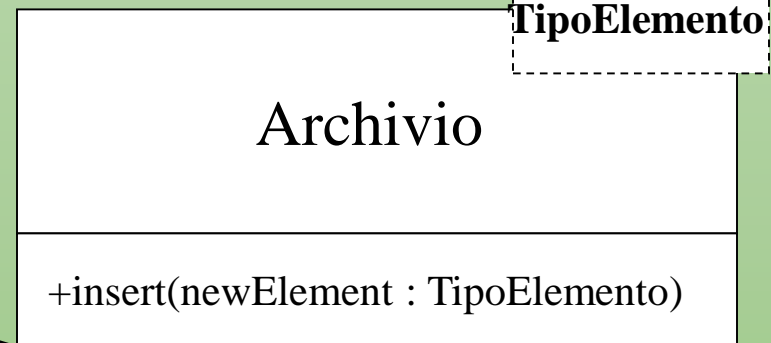
Polimorfismo



UML

Impiegato

Archivista



Il polimorfismo parametrico

Si deve osservare che il polimorfismo parametrico di Ada (ma anche di C++) è **sintattico**, dal momento che una istanziatura generica è effettuata al momento della compilazione sulla base dei tipi effettivi che devono essere determinabili (manifesti) al compile-time.

Pertanto, le procedure generiche di Ada (o le funzioni template di C++) **possono essere considerate delle abbreviazioni per insiemi di procedure monomorfe**. Rispetto al polimorfismo parametrico propriamente inteso, esse hanno il vantaggio che si può generare del codice ottimizzato per forme di input differenti.

Al contrario, nei veri sistemi polimorfici, il codice è generato **solo una volta** per ogni procedura generica (vedi Java).

Il polimorfismo di inclusione

Il polimorfismo di inclusione nasce da una relazione di inclusione fra insiemi di valori.

L'inclusione di insiemi di valori è *tipico* dei **sottotipi**.

Ad esempio in Pascal, ogni valore di un subrange di interi può essere usato nel contesto di un intero.

Tuttavia, il polimorfismo di inclusione non è *necessariamente* legato ai sottotipi.

Esempi di polimorfismo per inclusione built-in:

- *Pascal*: Il valore *Nil* appartiene a tutti i tipi puntatore.
- *C*: Il valore 0 è polimorfo. Appartiene a tutti i tipi puntatore.
- *C++*: Il tipo `void *` è un supertipo di tutti i tipi puntatore. Un qualunque puntatore può essere assegnato a un `void *`.

Il polimorfismo di inclusione

Nella programmazione orientata a oggetti, si ha polimorfismo per inclusione se un oggetto appartiene a una classe e a tutte le sue superclassi.

Esso si manifesta in almeno due modi:

- Si può assegnare un oggetto di una qualsiasi sottoclasse di una classe C a una variabile definita di classe C

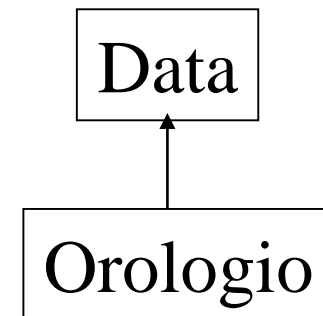
```
Articolo unArticolo = new Televisore()
```

- Una funzione che opera su un oggetto di classe C può essere applicata anche a oggetti di classe C', sottoclasse di C.

```
int giorno(Data d) {...}
```

```
Orologio o;
```

```
giorno(o)
```



Il polimorfismo di inclusione

Questi due utilizzi del polimorfismo di inclusione sono limitatamente interessanti.

Non a caso, essi non sono molto dissimili dall'utilizzo che permette il concetto di sottotipo anche nella programmazione imperativa.

L'utilizzo più interessante si ha quando le invocazioni dei metodi su oggetti di classi diverse (ma gerarchicamente correlate) produce un comportamento differente, anche se la definizione della funzione è unica.

Ciò dipende dal **tipo di legame statico/dinamico** fra identificatore di funzione e relativa realizzazione.

Il legame statico/dinamico

Nella maggior parte dei linguaggi di programmazione la visibilità degli identificatori e dei legami (*binding*) dei nomi alle dichiarazioni è determinata al momento della compilazione (*compile-time*). Si parla di **ambito d'azione statico** (*static scope*).

Nell'ambito d'azione **dinamico** (*dynamic scope*), il legame fra l'uso di un identificatore e la sua dichiarazione dipende dall'ordine di esecuzione, e così è differito al momento dell'esecuzione (*run-time*).

Il legame statico/dinamico

Esempio: Si consideri il seguente programma Pascal:

```
program dynamic(input,output);
```

```
var x: integer;
```

```
  procedure A;
```

```
  begin
```

```
    ...; write(x); ...
```

```
  end; {A}
```

```
  procedure B;
```

```
    var x: real;
```

```
  begin
```

```
    ...; A; ...
```

```
  end; {B}
```

```
begin
```

```
..., B; ... A; ...
```

```
end {dynamic}
```

Il legame statico/dinamico

Poiché il Pascal adotta la regola dell'ambito statico, l'uso della variabile x in A è legato alla variabile intera x nel programma principale. Questo permette di tradurre la `write(X)` semplicemente in una chiamata a una funzione di libreria di I/O per la scrittura degli interi.

Tuttavia se il legame nome-dichiarazione fosse dinamico, l'uso di x in A sarebbe vincolato alla dichiarazione di x più recente. Pertanto, quando la procedura A è chiamata dalla procedura B , l'uso di x in A viene vincolato alla dichiarazione della variabile reale x nella procedura B , mentre quando A è chiamata nel programma principale, l'uso di x sarebbe vincolato alla dichiarazione della variabile intera x nel programma principale.

Legame e tipizzazione

Con un dynamic-binding, la traduzione della chiamata della `write(x)` può essere determinata solo al run-time.

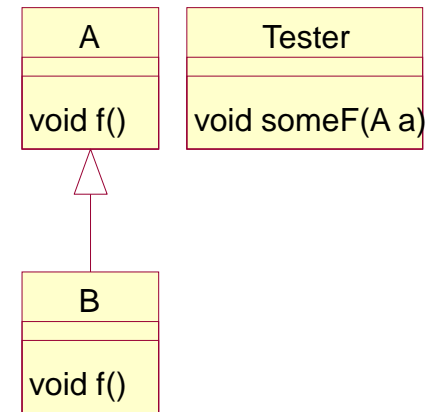
Questo non vuol dire che il controllo di tipo non possa essere effettuato, ma solo che viene ritardato al momento dell'esecuzione, quando è noto il tipo al quale `x` è vincolato.

Si osservi che con lo static-binding il legame dei nomi ai tipi (*name-type binding*) è anch'esso fissato al momento della compilazione.

Legame dinamico e polimorfismo di inclusione

Nella programmazione orientata a oggetti il legame fra nome di funzione e sua realizzazione può essere determinato al run-time.

```
class A {  
    public void f(){cout << 'A'}obj1 = new B();  
... }  
class B : public A {  
    public void f() {cout << 'B'} // overriding  
... }  
class Tester{  
    public void someF(A a){ a.f()..}  
...}
```



```
obj1 = new B();  
obj2 = new Tester();  
obj2.someF(obj1);
```

← Cosa stamperà? 'A' o 'B'?

Legame dinamico e polimorfismo di inclusione

Tutto dipende da quando si effettua il legame fra l'identificatore di funzione f in someF e la sua implementazione.

In C++ è possibile definire la funzione f come **virtuale**.

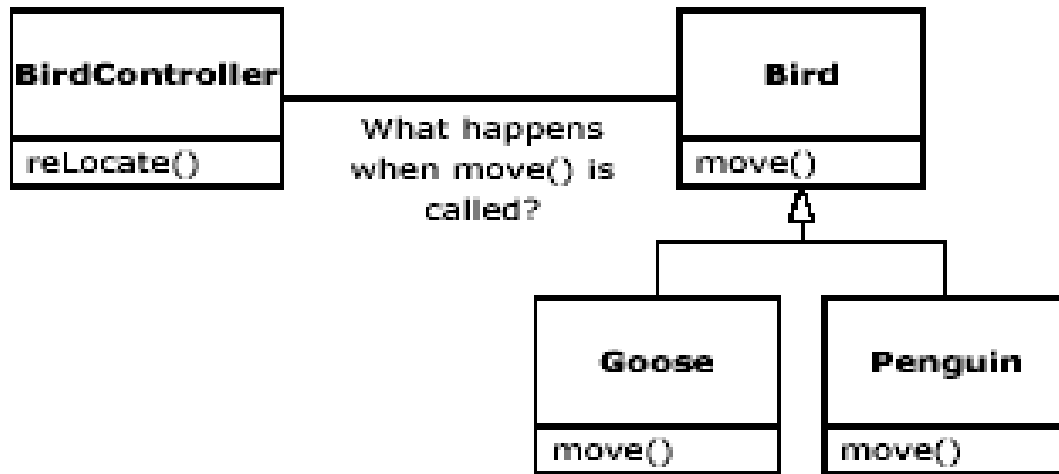
Questo significa che il legame deve essere **dinamico**.

Quindi verrà stampata una 'B'.

Diversamente, il legame sarebbe **statico** e **verrebbe stampata una 'A'**.

La selezione del metodo, nel caso di funzioni virtuali, avviene al run-time (dinamicamente).

Legame dinamico e polimorfismo di inclusione



Il metodo *reLocate* di *BirdController* invoca *move*.
A quale realizzazione di *move* si farà riferimento?

N.B.: Il legame dinamico ci permette di far evolvere il programma aggiungendo nuove sottoclassi di **Bird** e specificando il codice di `move()` senza che si modifichi il codice della `reLocate()` per operare su nuove istanze di nuove sottoclassi di **Bird**.

Combinando polimorfismo di inclusione con legame dinamico si hanno i vantaggi di **estendibilità** del codice.

Il polimorfismo di inclusione ...

C++ è un esempio di linguaggio dove la tipizzazione è statica, ma il legame può essere dinamico (*statically-typed dynamic binding*).

N.B.: Il legame si riferisce a (*nome di funzione, sua realizzazione*).

Lo stesso dicasi per il Java. Ma in questo linguaggio, il legame dinamico è la regola, non l'eccezione.

Solo nei metodi `final/static` il legame diventa statico.

Overloading + Coercizione + Inclusione = Mal di testa da C++

Che cosa stamperà il seguente codice?

```
void f(int) { cout << "int"; }  
void f(char) { cout << "char"; }  
void f(char *) { cout << "char *"; }
```

```
g()  
{  
    f(0);  
}
```

Diversi linguaggi di programmazione OO proibiscono la ridefinizione dei metodi e la coercizione per evitare questi problemi.

I diagrammi UML

- Sono tipicamente grafi che connettono elementi mediante archi che rappresentano le relazioni fra gli elementi.
- I diagrammi consentono la visualizzazione del sistema da diverse prospettive (proiezioni del sistema).
- Uno stesso elemento potrebbe comparire in tutti i diagrammi o, come accade generalmente, solo in alcuni.
- Ci sono diversi tipi di diagrammi in UML. Sono raggruppabili in due grosse categorie:
 - **livello logico**: class diagram, object diagram, use case diagram, sequence diagram, collaboration diagram, statechart diagram e activity diagram
 - **livello fisico**: component diagram e deployment diagram

Class diagram

- Mostra un insieme di classi, interfacce e le loro relazioni
- Mostra una vista statica del sistema (anche se include classi attive)
- In genere un class diagram viene utilizzato per rappresentare una semplice collaborazione
 - una collaborazione non è altro che una società di classi, interfacce e altri elementi che collaborano per raggiungere un obiettivo comune

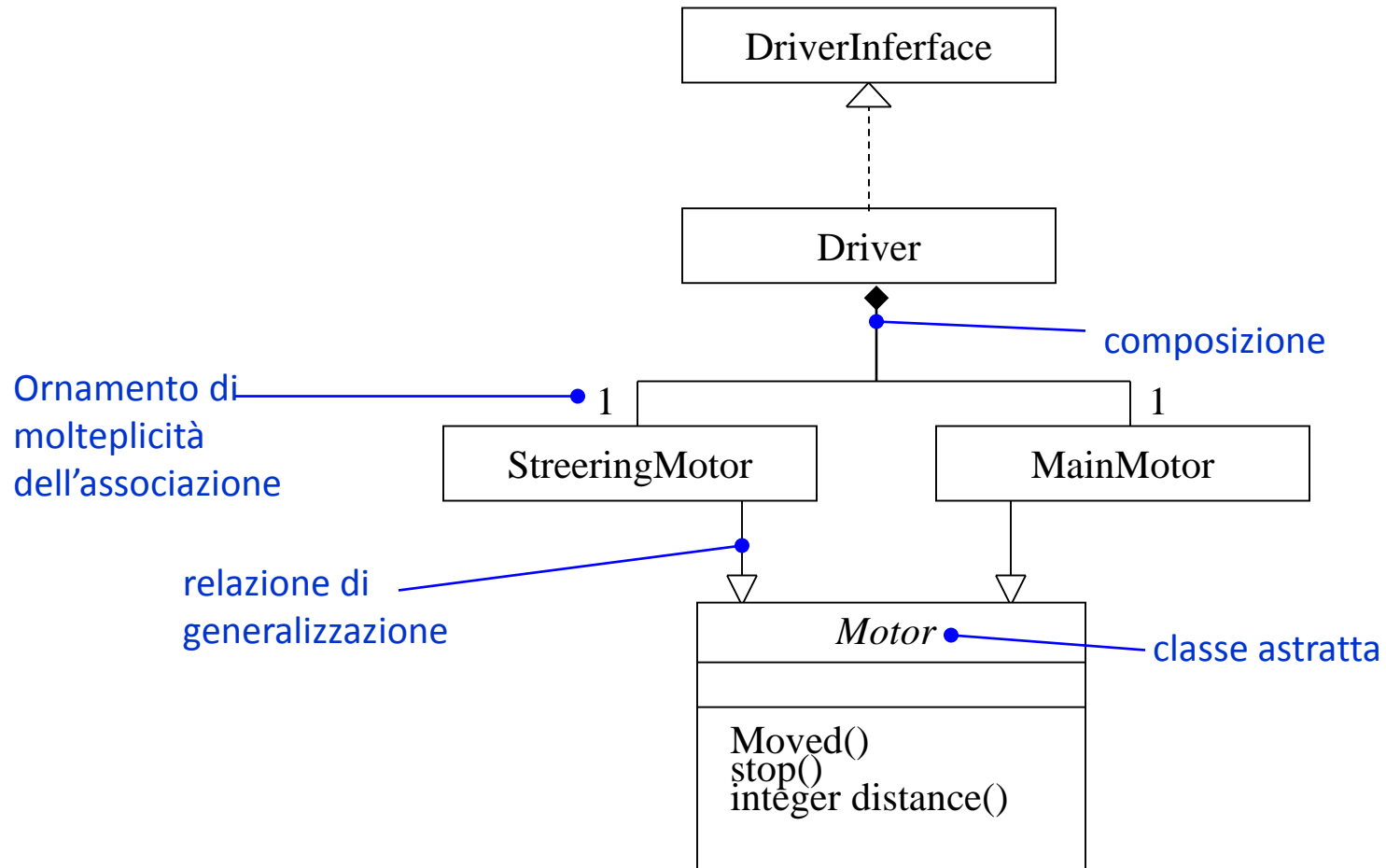
Class diagram

In una buona modellazione è necessario che ogni *class diagram* rappresenti una sola collaborazione.

Per modellare una collaborazione è necessario:

- identificare il **meccanismo** che si vuole modellare. Un meccanismo rappresenta alcune funzioni o comportamento di una parte del sistema che stiamo modellando
- per ogni meccanismo, si identificano **le classi, interfacce** e altri elementi che partecipano alla collaborazione. Vengono inoltre individuate le **relazioni** tra questi elementi
- assicurarsi che non vi siano parti del modello **omesse** o **semanticamente sbagliate**
- per ogni classe analizzare le **responsabilità affidate**, assicurandosi che siano ben distribuite, e ricavare da tali responsabilità gli attributi e le operazioni che consentano alla classe di soddisfarle

Class diagram



Qualità del SW

Fra le qualità del software si annoverano:

- **robustezza**: la misura con cui il sistema si comporta in modo ragionevole in situazioni impreviste, non contemplate dalle specifiche

A questa qualità contribuisce l'applicazione dei principi dell'astrazione di controllo, e in particolare la gestione delle *eccezioni*.

Nella progettazione OO occorrerà prevedere delle classi specifiche per la descrizione di eccezioni che potranno essere sollevate e quindi catturate e gestite dal software. Nella parte di programmazione Java si chiariranno questi concetti.

Qualità del SW

- **estendibilità**: facilità con cui il software può essere adattato a modifiche delle specifiche

A questa qualità contribuiscono:

- *l'ereditarietà*, che rende possibile la derivazione di classi con comportamenti specifici (overriding e aggiunta di attributi/metodi);
- la definizione di *interfacce*, che rende possibile specificare dei contratti che verranno poi implementati.

Qualità del SW

- **riusabilità**: facilità con cui il software può essere re-impiegato in applicazioni

A questa qualità contribuiscono:

- *l'occultamento dell'informazione e l'incapsulamento*, che rendono possibile l'uso di oggetti ricorrendo solo a metodi pubblici;
- *l'ereditarietà e la composizione* di classi, che rendono possibile il riutilizzo di metodi definiti in altre classi per implementare nuovi servizi;
- *l'astrazione generica*, che consente di definire delle soluzioni il più possibile svincolate dal tipo di dati su cui lavorano gli algoritmi.

Riferimenti bibliografici

UML

James Rumbaugh, Ivar Jacobson, Grady Booch
The Unified Modeling Language reference manual
Addison-Wesley (1999)

Grady Booch, James Rumbaugh, Ivar Jacobson
The Unified Modeling Language user guide
Addison-Wesley (1999)

Riferimenti bibliografici

Paradigma orientato a oggetti

Tim Korson, John D. McGregor

Understanding Object-Oriented: A Unifying Paradigm

Communications of the ACM, (1990)

Riferimenti bibliografici

Ereditarietà

Bertrand Meyer

The many faces of inheritance: A taxonomy of taxonomy,
Computer, vol. 29, n. 5, pp. 105-108, Maggio, 1996

G. Masini, A. Napoli, D. Colnet, D. Léonard, & K. Tombre
Linguaggi per la Programmazione a Oggetti (cap. 2-3, 6)
Gruppo Editoriale Jackson, 1989

Riferimenti bibliografici

Ereditarietà vs. composizione

M. Cadoli, M. Lenzerini, P. Naggar, A. Schaerf
Fondamenti della progettazione dei programmi (cap. 5)
Città Studi Edizioni, 1997.

Polimorfismo

L. Cardelli, P. Wegner
On understanding types, data abstraction, and polymorphism.
ACM Computing Surveys, vol. 17, no. 4, pp. 471-522, December 1985