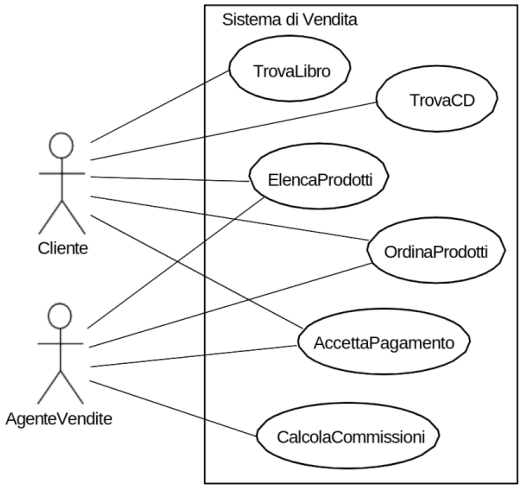
1 Lezione del 03-10-25

Cotinuiamo a parlare dell'analisi dei requisiti, in particolare riguardo ai casi d'uso, introdotti in 2.3.3.

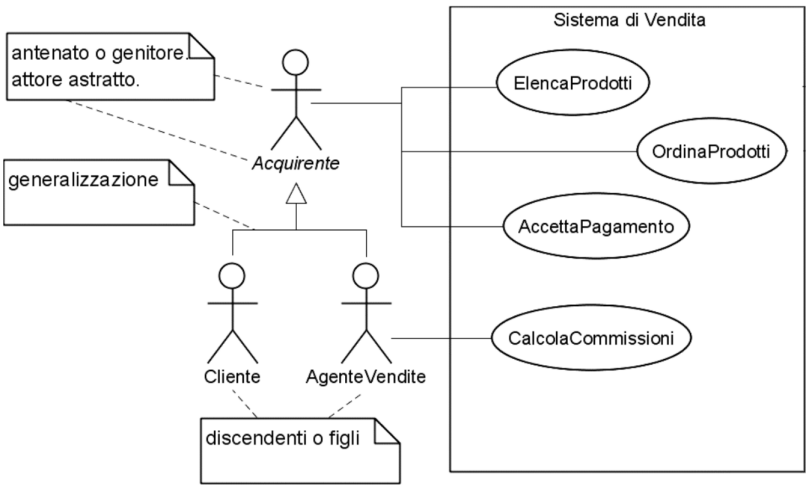
1.1 Modellizzazione casi d'uso avanzata

La modellizzazione dei casi d'uso già vista, comprensiva di *soggetto* (il dominio del sistema), *attori* (entità che interagiscono col sistema) e *casi d'uso* (modalità secondo le quali si interagisce col sistema) può essere espansa introducendo alcuni costrutti dal paradigma della programmazione ad oggetti. Vediamone alcuni:

- Può essere utile stabilire **relazioni** fra attori e casi d'uso, cioè semplicemente stabilire quali attori usino quali casi d'uso (senza negare a più attori di usare lo stesso caso d'uso). Vediamo ad esempio come a due attori diversi, *Cliente* e *AgenteVendite*, possono interessare casi d'uso differenti all'interno dello stesso sistema:

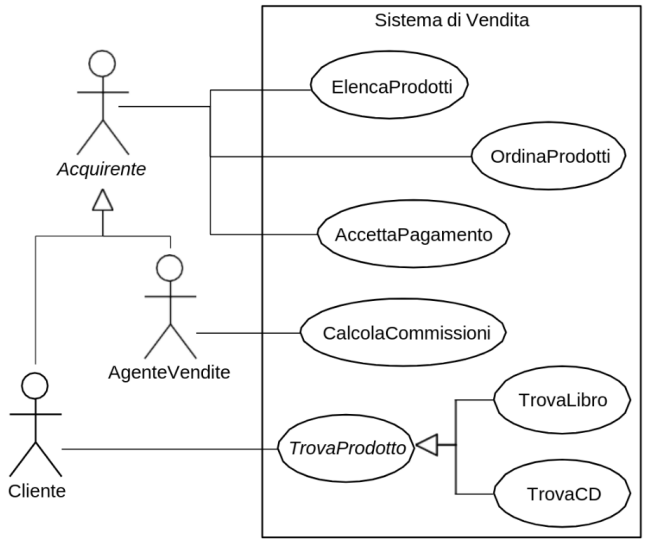


- La complessità data da più attori che interagiscono con gli stessi casi d'uso può essere ridotta introducendo la **generalizzazione degli attori**: più *sottoattori* potrebbero ereditare relazioni con casi d'uso da un *superattore* comune e *generico*. Riguardo all'esempio precedente, vediamo come si può introdurre un *Acquirente*:



Gli attori *Cliente* e *AgenteVendite* diventano a questo punto generalizzazioni di *Acquirente*. Notiamo come questo ci permette di spiegare elegantemente come mai entrambi gli attori sono interessati a casi d'uso comuni come *ElencaProdotti*, ecc...

- Come si generalizzano gli attori, potrebbe essere utile stabilire la **generalizzazione dei casi d'uso**: i sottocasi ereditano tutte le caretteristiche (relazioni, punti d'estensione, precondizioni, postcondizioni, flussi principali e alternativi) del supercaso, e possono sovrascriverle (tranne relazioni e punti d'estensione). Ad esempio, nell sistema visto finora si potrebbe voler generalizzare il caso d'uso *TrovaProdotto* sulla base del prodotto cercato:



  Potrebbe essere utile definire alcune regole su come i sottocasi possono sovrascrivere i flussi dei supercasi, assunti questi come liste di operazioni:

  – Si indicano fra parentesi le operazioni ereditate, ad esempio 3. (3.) o 4. (3.) se si è rinumerata;
  – Si prefiggono con una "o" loe operazioni sovrascritte, ad esempio 3. (o3.) o 4. (o3.) se si è rinumerata;
  – Infine, le operazioni aggiunte da zero si indicano semplicemente con numeri non usati nel supercaso.

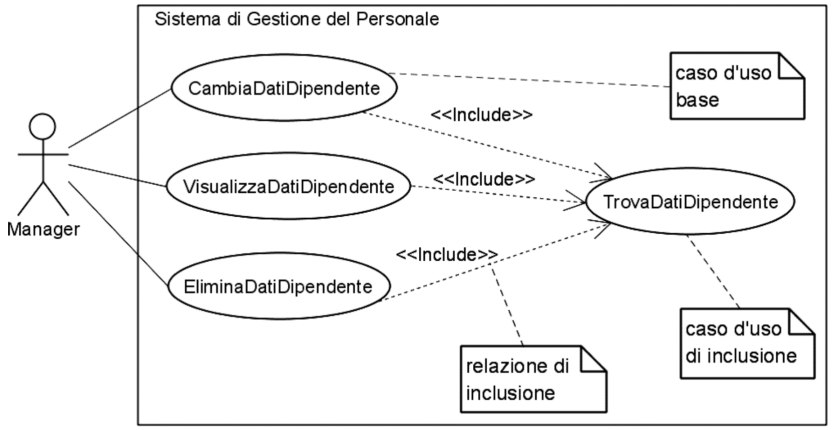
- Se si prevedono casi d'usi e attori "figli", possiamo anche prevedere casi d'uso e attori **astratti**, cioè incompleti e pensati per definire caratteristiche base. In particolare, preferiamo definire i **casi d'uso astratti** come casi d'uso con sequenze di operazioni incomplete e da definire, mentre gli **attori astratti** possono essere una variante più forte degli *attori generici* visti quando abbiamo introdotto la *generalizzazione degli attori*.

1.1.1 Relazioni `<<include>>` e `<<extend>>`

Altri due meccanismi di estensione della modellizzazione dei casi d'uso sono rappresentati dalle relazioni `<<include>>` e `<<extend>>`. Abbiamo che queste relazioni sollevano la specifica del flusso dalle relazioni di ereditarietà fra casi d'uso, mantenendo solo il collegamento semantico. Vediamole nel dettaglio:

- La relazione `<<include>>` definisce, appunto, l'*inclusione* di un caso d'uso (detto **caso d'inclusione**) all'interno di un altro (detto **caso base**). Dal punto di vista del programmatore, il caso d'inclusione rappresenta un frammento di comportamento che viene ripetuto in più casi d'uso base allo stesso modo (la corrispondenza più significativa dal lato implementativo è una banale chiamata di funzione).

  Un esempio di inclusione di casi d'uso potrebbe essere il seguente:



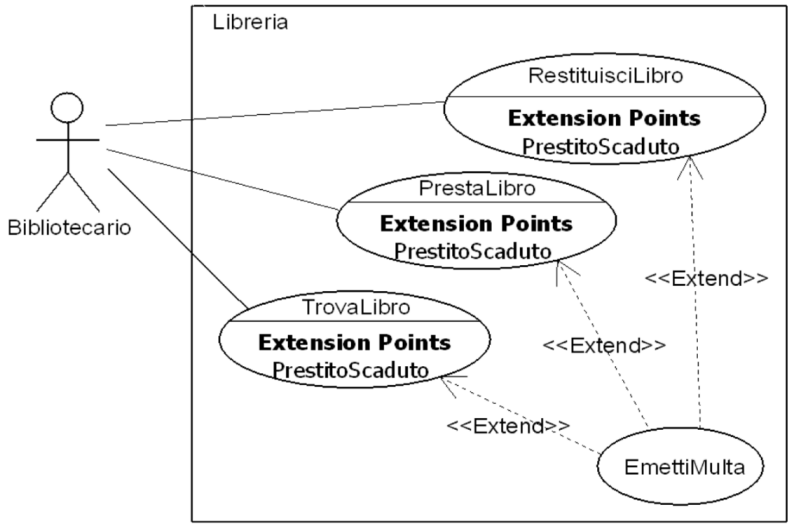
  dove il frammento di comportamento *TrovaDatiDipendente* è incluso nei casi d'uso *CambiaDatiDipendente*, *VisualizzaDatiDipendente*, *EliminaDatiDipendente*.

  Abbiamo che sicuramente i casi base non sono completi senza casi d'inclusione. Di contro, anche i casi di inclusione possono essere incompleti senza un caso base che li includa: in questo caso si dicono **non istanziabili**.

- La relazione `<<extend>>` definisce una relazione apparentemente simile all'inclusione, ma più legata alla gestione di eccezioni ai flussi principali: definiamo infatti **punti di estensione** i punti di un flusso dove si può aggiungere comportamento. In questo, i **casi di estensione** *estendono* i **casi base** nei punti di estensione. Nulla nega che lo stesso caso d'estensione estenda più punti d'estensione.

  Per approfondire, abbiamo che ogni caso d'uso d'estensione non è completo, e consiste di più frammenti detti **segmenti di inserzione**. Ogni segmento di inserzione deve combaciare con un punto d'estensione del caso base che si va ad estendere. Inoltre, si può avere che due casi di estensione estendono lo stesso punto d'estensione: in questo caso si prevedono più comportamenti d'estensione, il cui ordine di esecuzione è indeterminato.

Vediamo come si possono estendere i casi d'uso degli esempi precedenti introducendone uno, *EmettiMulta*, e usandolo per estendere i flussi alternativi di fallimento:



Infine, abbiamo che i casi di estensione possonoo essere vincolati da *precondizione*, e vincolare lo stato successivo del sistema dopo la loro esecuzione attraverso *postcondizioni*. Per completare il quadro assieme all'esempio fatto sopra, la corrispondenza implementativa più banale è quella di una chiamata condizionale di funzione.

1.2 Workflow analisi

Il workflow analisi ha il compito di creare un modello che catturi *cosa* il sistema deve fare (il *come* è prerogativa del workflow progetto). Gli artefatti prodotto in questo workflow sono:

- **Classi di analisi**: modellano i concetti chiave del soggetto;
- **Realizzazioni di casi d'uso**: illustrano com ele istanze delle classi di analisi possono interagire per realizzare i casi d'uso registrati nel workflow requisiti.

In questo caso, questa parte corrisponde con le *strutture statiche* e il *comportamento dinamico* che avevamo discusso in 2.1.

1.2.1 Classi e oggetti

Un'**oggetto** è un entità discreta con un confine ben definito che ne incapsula stato e comportamento. La struttura dell'oggetto e il modo in cui ci si interagisce sono definite dalla **classe** dell'oggetto. La classe si distingue dall'oggetto in quanto l'oggetto è un **istanza** della classe.

Tutti gli oggetti hanno **proprietà** comuni:

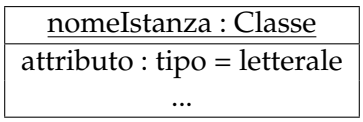
- **Identità**: un numero o qualche altro identificativo che individua univocamente l'oggetto. Per quanto ci riguarda, un *riferimento* all'istanza di classe;

- **Stato**: determinato dallo stato interno corrente dell'oggetto e le relazioni che al momento questo ha con altri oggetti. Per quanto ci riguarda, il valore dei *campi* dell'oggetto;
- **Comportamento**: le operazioni che l'oggetto può compiere. Chiaramente, questo può essere influenzato dallo stato corrente dell'oggetto. Per quanto ci riguarda, sono i *metodi* dell'oggetto.

Vediamo quindi come l'UML mira a modellizzare sistemi **orientati agli oggetti**, dove più oggetti *istanze* di *classi* interagiscono (**scambiano messaggi**) mantenendo privato il loro stato interno (**incapsulamento**).

1.2.2 Notazione di oggetti

In UML per identificare oggetti si usa un formato tabulare del tipo:



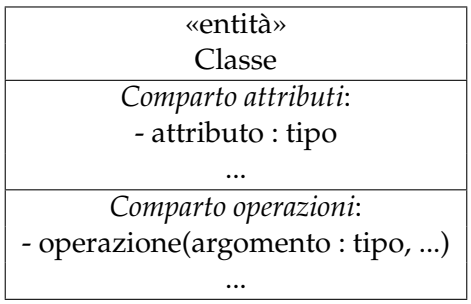
dove notiamo che:

- L'identificatore di oggetto è sempre sottolineato;
- Il nome dell'oggetto può essere risparmiato nel caso di *oggetti anonimi*;
- L'identificatore può di contro essere composto solo dal nome dell'oggetto: questo è utile in fase di analisi prima che una classe vera e propria venga definita.

Le operazioni defnite sull'oggetto non vengono riportate in quanto sono comuni a tutte le istanze di classe (vanno cercate nella definizione di classe).

1.2.3 Notazione di classi

Per identificare classi in UML si usa ancora una volta una struttura tabulare:



dove notiamo che:

- Distinguiamo in *comparto attributi* e *comparto operazioni*:
  - *Comparti attributi*: la visibilità viene solitamente omessa. A volte, si possono usare gli ornamenti (+, -, #, ~), che corrispondono a (*public*, *private*, *protected*, *package* (o *friendly*)). Inoltre, si può indicare una *molteplicità*, che per i nostri scopi definisce array;

  - *Comparti operazioni*: di queste si possono definire i parametri in entrata e in uscita usando le parole chiave `in`, `out` e `inout` nella lista degli argomenti.

Notiamo inoltre che si possono definire operazioni specifiche all'implementazione del paradigma OOP (tipicamente il **costruttore**), e che si possono sottolineare i membri, in qualsiasi comparto, che hanno **visibilità di classe**. Quest'ultima caratteristica è per noi assimilabile all'idea dei membri *statici*: come sappiamo la classe può accedere ai suoi membri statici, mentre le istanze possono accedere sia ai membri di istanza che ai membri statici.

Una volta definite classi e istanze di queste, si può usare la relazione `<<instantiate>>` (diretta da istanza a classe) per definirne la dipendenza.

1.2.4 Trovare le classi

Il problema fondamentale del design OOP diventa quindi trovare tassonomie di classi che descrivono bene il soggetto e implementano con relazioni semplici e compatte i casi d'uso. Non esiste una metodologia prefissata per lo sviluppo di tali tassonomie. Alcuni metodi che possiamo usare sono:

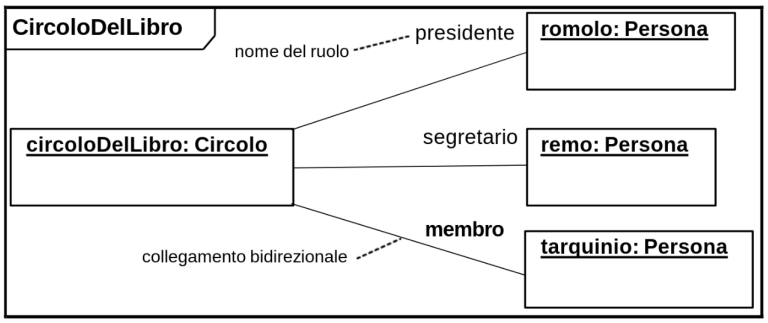
- Analisi **nomi-verbi**: i nomi ed i verbi nel testo indicano rispettivamente classi e responsabilità. Chiaramente, questo metodo richiede una buona conoscenza del soggetto, un buon glossario (in modo da individuare sinonimi ed omonimi), e potrebbe richiedere ulteriori fasi di elaborazioni delle classi candidate trovate (magari fusioni di classi simili, rimozione di classi inutili, ecc...);
- Analisi **CRC** (*Classe, Responsabilità, Collaboratore*): possiamo intendere questo metodo come estensione del precedente. Per *collaboratori* si intendono altre classi che possono collaborare con la classe definita per realizzare parte della funzionalità del sistema;
- Analisi **RUP** (*Rational Unified Process*): questa tecnica si basa sull'individuare classi sulla base di tre modelli:
  - Classi **boundary** (*confine*): mediano l'interazione tra soggetto e attori esterne. Rappresentano quindi interfacce, che siano con gli utenti, con altri sistemi o con dispositivi fisici;
  - Classi **control** (*controllo*): rappresentano controllori che mediano il comportamento associato a casi d'uso. Queste non devono essere troppo grandi, ed anzi è buona pratica suddividerle in più unit funzionali;
  - Classi **entity** (*entità*): caso principe dell'OOP, queste modellano oggetti reali all'interno del soggetto. Vengono gestite dalle classi di controllo per implementare i casi d'uso, e vengono scambiate fra le classi di confine.

Oltre a queste metodologie si possono individuare le classi sfruttando oggetti fisici, documenti d'ufficio, usando interfacce, adottando modellizzazioni in *componenti* del soggetto o entità concettuali.

1.2.5 Relazioni fra classi

Una volta individuate le classi, cioè la struttura *statica* del sistema, è opportuno definire la natura *dinamica* della collaborazione che ci aspettiamo fra le stesse. Questa si definisce attraverso **collegamenti** fra *oggetti* e **associazioni** fra *classi*.

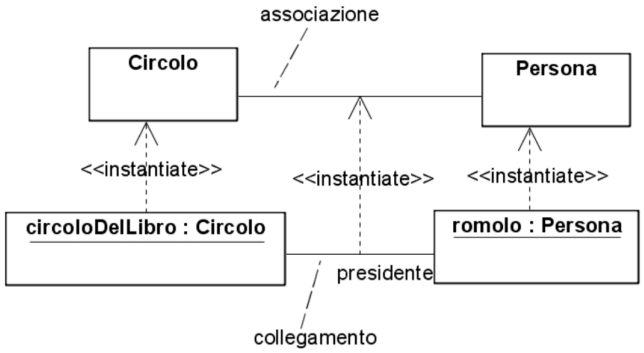
Per definire i collegamenti bisogna usare un linguaggio sollevato delle specifiche del linguaggio di programmazione, come i **diagrammi ad oggetti** dell'UML:



I *collegamenti* in UML sono **dinamici** (non fissi nel tempo), possono essere **bidirezionali** o **monodirezionali**, e devono essere supportati da una *associazione* fra classi.

Le *associazioni* sono solitamente più sofisticate nei diagrammi, e collegate ai rispettivi collegamenti da relazioni `<<instatiate>>`. Nelle associazioni vogliamo definire una stringa che definisce l'associazione, i **ruoli** delle classi coinvolte, e magari indicazioni sulla **molteplicità** dell'associazione (uno a uno, uno a molti, molti a molti e via dicendo).

Vediamo quindi come associzioni e *collegamenti* sono schematizzabili rispettivamente fra **classi** e **oggetti**, cioè:



La molteplicità può poi essere supportata dalla **navigabilità** dell'associazione, intesa nei nostri scopi come la possibilità di raggiungere una classe a partire da quella a cui è associata. Dovrebbe apparire chiaro come, ad esempio, una classe che contiene un riferimento ad un'altra ci permette di navigare verso questa, e non viceversa. Il sistema di navigabilità di UML 2.0 è comunque più sofisticato di quanto effettivamente necessario, e le associazioni *bidirezionali* e *monodirezionali* bastano spesso a riassumere tutta la funzionalità di cui abbiamo bisogno.

1.2.6 Associazioni ed attributi

L'assocazione fra classi può essere espressa anche come particolari attributi di classe (relazioni uno a uno sono riferimenti, uno a molto array di riferimenti, ecc...). In questo caso conviene adottare la regola: se la classe associata è parte integrante del modello, usare un'associazione, altrimenti un'attributo.

1.2.7 Classi di associazioni

Se l'associazione è molti a molti potrebbero sussistere dei campi che si perdono nella semplice specifica di associazione. In questo caso può essere utile costruire **classi di associazione**, che contengono riferimenti ad ognuna delle classi impegnate nell'associazione, e i campi associati a questa.

1.2.8 Associazioni qualificate

Le associazioni uno a molti possono essere qualificate da un particolare **qualificatore**, cioè un particolare indice che ci permette di trasformare la relazione in uno a uno, navigando verso la classe associata. L'esempio tipico è il qualificatore *codice fiscale* per una classe che si associa a più classi individuo.

1.2.9 Dipendenza

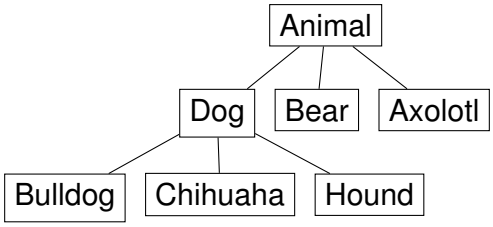
Una dipendenza è una particolare relazione tra due elementi, dove la modifica di un elemento (detto **fornitore**) può influenzare l'altro elemento (detto **cliente**). Esistono 3 tipi di dipendenze in UML:

- Dipendenza di **uso**: se il cliente usa alcuni servizi messi a disposizione dal fornitore. Questa si divide a sua volta in:
  - `<<use>>`: il cliente usa in qualche modo il fornitore;
  - `<<call>>`: il cliente invoca un'operazione eseguita dal fornitore;
  - `<<parameter>>`: il fornitore è parametro di un'operazione del cliente;
  - `<<send>>`: il cliente invia il fornitore a qualche altro bersaglio;
  - `<<instantiate>>`: il cliente è un'istanza del fornitore, già vista.
- Dipendenza di **astrazione**: se le entità sono a livelli di astrazione diversi. Questa si divide a sua volta in:
  - `<<trace>>`: cliente e fornitore rappresentano lo stesso oggetto ma in modelli diversi;
  - `<<substitute>>`: il cliente può sostituire il fornitore;
  - `<<refine>>`: simile a `<<trace>>`, ma riferito ad un'ottimizzazione all'interno dello stesso modello;
  - `<<derive>>`: il cliente può essere derivato in qualche modo dal fornitore.
- Dipendenza di **permesso**: modella la capacità di un'entità di accedere ad un'altra entità. Questa si divide a sua volta in:
  - `<<access>>`: il cliente può accedere ai contenuti pubblici del fornitore, si riferisce principalmente ai *package*;
  - `<<import>>`: simile al precedente ma con la fusione dello spazio dei nomi;
  - `<<permit>>`: permette la violazione dell'incapsulamento in linguaggi come il C++ (pensa funzioni e classi `friend`).

1.2.10 Ereditarietà

La **generalizzazione** o *ereditarietà* è la forma più forte di dipendenza. Questa indica che un entità cliente (detta *specializzata*) può essere usata in qualsiasi contesto dell'entità fornitore (detta *base*), con la differenza che l'entità cliente definisce informazioni e procedure aggiuntive rispetto all'entità fornitore.

Facciamo l'esempio classico di un **diagramma di ereditarietà** che definisce una *tassonomia* di classi (cioè) il modo in cui più classi ereditano funzionailtà l'una dall'altra, dall'alto verso il basso:



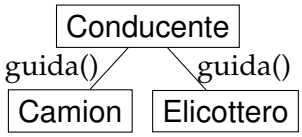
In particolare, in questo caso vogliamo usare l'ereditarietà per modellare una relazione `<<is-a>>` (*"è un"*): un bulldog *è un* cane, che a sua volta *è un* animale, e cosi via.

Sfruttando l'ereditarietà riusciamo quindi a modellare bene il tipo di categorizzazioni che incontreremo naturalmente nel soggetto del sistema che andiamo a sviulppare.

1.2.11 Polimorfismo

L'idea del **polimorfismo** è simile a quella dell'ereditarietà, e indica contesti dove si può inviare lo stesso messaggio a entità di classe differente e aspettarsi risposte semanticamente appropiate.

Per fissare i concetti, una classe base *Veicolo* può essere **generalizzata** in *Camion*, *Elicottero*, ecc... la possibilità di inviare a tutte queste il segnale *guida()* è un'esempio di **polimorfismo**:



Chiaramente, in un linguaggio di programmazione reale si dovranno usare i costrutti tipici dell'OOP per realizzare la stessa funzionalità, come classi o membri *astratti* e il meccanismo dell'*overriding*.