



# MODELLAZIONE CON UML

---

# RIFERIMENTI

- Martin Fowler, UML Distilled,
- Capitolo 1 (introduzione a UML),
- Capitolo 3 e 5 (class diagram)
- Capitolo 4 (sequence diagram)
- Capitolo 7 (package diagram)



# SOFTWARE PER LA MODELLAZIONE CON UML

- Visual Paradigm è un ottimo strumento per disegnare diagrammi UML

- <https://www.visual-paradigm.com/>

- E' disponibile anche una Community Edition di libero utilizzo

- <https://www.visual-paradigm.com/solution/freeumltool/>

- Basta seguire le istruzioni per l'attivazione gratuita



- Occasionalmente utilizzeremo anche funzionalità interne a IntelliJ Idea per la co-generazione di codice e diagrammi
  - Plug-in e opzione chiamata Diagrams (dovrebbe essere già inclusa nella Ultimate)



# INTRODUZIONE A UML

UML Distilled, Capitolo 1



# MODELLAZIONE DEL DOMINIO DEL PROBLEMA

- Diversi tipi di modelli possono descrivere il dominio del problema. Tali modelli sono detti modelli di *dominio* oppure modelli *concettuali* o, talvolta, modelli di *business*
  - Attenzione: non bisogna confondere i *modelli di dominio*, che descrivono *il problema*, dai *modelli di progetto*, che descrivono *la soluzione*
- Si distingue tra:
  - Modelli statici, che descrivono gli elementi del dominio del problema e le relazioni tra loro
  - Modelli dinamici, che descrivono i comportamenti riconoscibili nel dominio del problema



# MODELLAZIONE DI SISTEMA

- Per descrivere tali modelli si utilizzano linguaggi di modellazione
- In passato, diversi linguaggi di modellazione erano utilizzati a supporto delle metodologie che si applicavano nelle varie fasi del processo di sviluppo del software
  - Ad esempio:
    - Diagrammi di flusso
    - Data Flow Diagrams
    - ...
- Negli ultimi anni il linguaggio UML si sta affermando come linguaggio unificato che possa essere utilizzato in tutte le attività di modellazione, nonché in molte altre attività del ciclo di vita del software



# CHE COSA È UML?

- UML (Unified Modelling Language) nasce come linguaggio grafico standard per modellare software object oriented
  - Tra la fine degli anni '80 e gli inizi degli anni '90 fecero la comparsa i primi processi di sviluppo object-oriented
  - La proliferazione di metodi e notazioni diverse creavano confusione
  - Due importanti metodologi, Rumbaugh e Booch, decisero di fondere i loro approcci nel 1994.
    - Cominciarono a lavorare insieme alla Rational Software Corporation
  - Nel 1995, un altro metodologo, Jacobson, si unì al gruppo
    - Il suo lavoro si concentrava sugli use cases
  - Nel 1997 l'Object Management Group (OMG) cominciò il processo di standardizzazione di UML
  - Nel Luglio 2005 è stata rilasciata la prima release di UML 2
- Su <http://www.omg.org/spec/UML/> è possibile leggere la storia di tutte le specifiche UML
  - Una specifica UML in pratica definisce un **metamodello** che indica secondo quali regole sia possibile costruire modelli UML



# USO DI UML

- UML può essere usato:
  - Come *abbozzo*, cioè per tracciare un modello di massima di un sistema da realizzare
  - Come *progetto*, cioè per realizzare un modello completo della soluzione architetturale del sistema
  - Come *linguaggio di programmazione* in grado di modellare in maniera completa e precisa il sistema software
    - L'approccio MDA (Model Driven Architectures) esplora la possibilità di usare UML come linguaggio di programmazione
      - PlantUML è un recente strumento per la descrizione testuale di diagrammi UML in forma di codice





# REGOLE DI UML

- UML è dotato sia di regole *prescrittive* che di regole *descrittive*
  - Le regole prescrittive sono regole stabilite da organismi standardizzanti che caratterizzano precisamente lessico, sintassi e semantica
  - Le regole descrittive, invece, hanno come scopo solo la comunicazione del significato dei diagrammi, con estensioni, libere ma intuitive, delle regole base
- UML ha come scopo principale la descrizione efficace di situazioni reali, cosicchè le regole descrittive sono al momento predominanti
- Regola delle informazioni soppresse:
  - L'assenza di qualche informazione in un diagramma UML non significa, in generale, che tale informazione non esista o sia nulla, ma semplicemente che si tratti di un aspetto del problema non ancora trattato nella fase in cui è stato tracciato il diagramma (o che non si voglia palesare in tale diagramma per poi inserirlo in diagrammi ulteriori)



# TIPI DI MODELLI IN UML 2

- UML 2 possiede 13 differenti tipi di diagrammi, appartenenti a tre categorie:
  - **Diagrammi Comportamentali**, quali use-case diagrams, activity diagrams, state machine diagrams
  - **Diagrammi di Interazione**, per modellare interazioni fra entità del sistema, quali sequence diagrams e communication diagrams.
  - **Diagrammi Strutturali**, che modellano l'organizzazione del sistema, quali class diagrams, package diagrams, e deployment diagrams.
- In questo primo corso che tratta UML, ci concentreremo sui due diagrammi più utilizzati:
  - **Class diagrams**
  - **Sequence Diagrams**



# CLASS DIAGRAM

UML Distilled, Capitolo 3, Capitolo 5



# CLASS DIAGRAM

- Il più diffuso diagramma compreso in UML è il diagramma delle classi
- Si tratta di un diagramma statico che può essere utilizzato:
  - Per la modellazione concettuale del dominio di un problema
  - Per la modellazione delle specifiche richieste ad un sistema
  - Per modellare l'implementazione (object-oriented) di un sistema software
- I concetti fondamentali di un class diagram sono estensioni dei concetti fondamentali dei paradigmi object-oriented
- Nel seguito verrà presentato il class diagram nella sua accezione più completa, relativa alla modellazione dell'implementazione di sistemi object-oriented



# ASPETTI PRINCIPALI

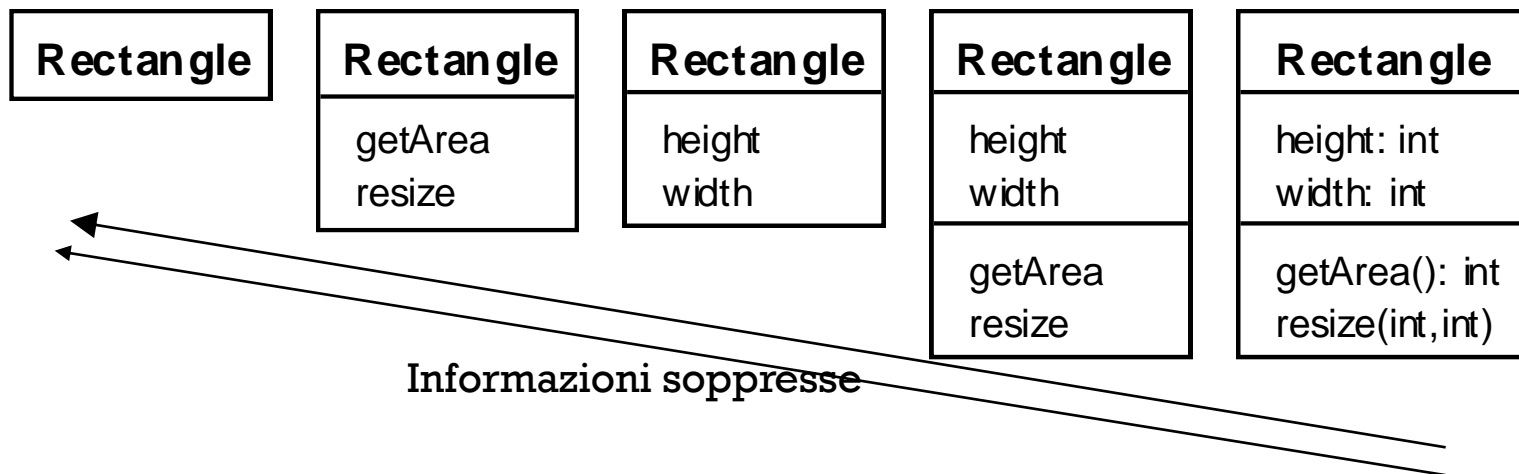
- *I principali elementi dei class diagram sono:*
  - *Classi*
    - Rappresentanti i tipi di dati presenti in un sistema
  - *Associazioni*
    - Rappresentano i collegamenti fra istanze di classi
  - *Attributi*
    - Sono i dati semplici presenti nelle classi e nelle loro istanze
  - *Operazioni*
    - Rappresentano le funzioni svolte dalle classi e dalle loro istanze
  - *Generalizzazioni*
    - Raggruppano le classi in gerarchie di ereditarietà



# CLASSI

- Una classe è semplicemente rappresentata da un rettangolo con il nome della classe all'interno
  - Il concetto di classe è lo stesso dell'OO
  - La signature completa di un'operazione è:

**operationName(parameterName: parameterType ...): returnType**



# ATTRIBUTI

**visibilità nome molteplicità: tipo = default {proprietà}**

Sono consentiti tre livelli di visibilità:

- + Pubblico:** L'utilizzo viene esteso a tutte le classi
- # Protetto:** L'utilizzo è consentito soltanto alle classi che derivano dalla classe originale
- Privato:** Soltanto la classe originale può utilizzare gli attributi e le operazioni definite come tali.

Il **nome** dell'attributo é l'unico parametro necessario

Il **tipo** dell'attributo può essere un tipo primitivo (int, double, char, etc...) oppure il nome di una classe definita nello stesso diagramma (in tal caso forse l'attributo andrebbe indicato con un'associazione ...)

**Default** rappresenta il valore di default dell'attributo



# ATTRIBUTI

**visibilità nome molteplicità: tipo = default {proprietà}**

La **molteplicità** indica il quantitativo degli attributi (ad esempio la dimensioni per un array). Tramite la molteplicità è possibile indicare come attributi degli array o matrici. Il valore di default é 1.

Alcuni valori possibili sono:

- **1** (uno e uno solo). E' il valore di default
- **0..1** (al più uno)
- **\*** (un numero imprecisato, eventualmente anche nessuno; equivalente a 0..\*)
- **1..\*** (almeno uno)

Gli elementi di una molteplicità sono considerati come un insieme.

Se essi sono dotati anche di ordine si aggiunge l'indicazione {ordered}.

Se sono possibili valori duplicati si aggiunge l'indicazione {nonunique}

{**proprietà**} rappresenta caratteristiche aggiuntive dell'attribute (ad esempio la sola lettura)

Esempio: `name: String [1] = "Untitled" {readOnly}`





# METODI

**visibilità nome (lista parametri) : tipo-ritornato {proprietà}**

La **visibilità** e il **nome** seguono regole analoghe a quelle degli attributi.

**Lista parametri** contiene nome e tipo dei parametri della funzione, secondo la forma:

**direzione nome parametro: tipo = valore-di-default**

**direzione**: input (*in*), output (*out*) o entrambi (*inout*). Il valore di default é *in*

**nome, tipo e valore di default** sono analoghi a quelli degli attributi

**Tipo-ritornato** é il tipo del valore di ritorno: dovrebbe essere un tipo appartenente ad una classe standard

Esempio:

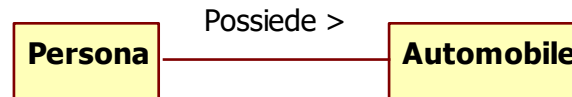
+ balanceOn (date: Date) : Money



# ASSOCIAZIONI

- Un'associazione rappresenta una relazione (fisica o concettuale) tra classi

Esempio: **Persona possiede Automobile**



- **Il verso dell'associazione indicato in figura indica in che direzione deve essere *letta* l'associazione**
  - **In questo caso indica che è la Persona a possedere l'Automobile e non l'Automobile a possedere la Persona!**

- **Possibile implementazione:**

```
class Persona{
    Automobile automobilePosseduta;
}
class Automobile{
}
```



# ASSOCIAZIONI

- In alternativa, si può indicare il *ruolo* di uno dei due estremi dell'associazione



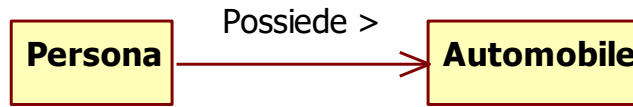
- **Possibile implementazione:**

```
class Persona{
}
class Automobile{
    Persona proprietario;
}
```



# VERSO DI NAVIGAZIONE

- Verso di navigazione di un'associazione
  - Il verso di navigazione è un'informazione utile soprattutto in fase di progetto di dettaglio
  - Indica in quale direzione è possibile reperire le informazioni



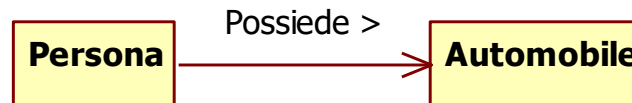
- Nell'esempio, nota una persona è possibile sapere quali sono le automobili che possiede (se ne possiede)
- Viceversa, non è possibile conoscere il possessore di una data automobile
- Non ci sono, però, indicazioni sul quantitativo di automobili possedute, nè sul numero di proprietari di un automobile (da questo diagramma non possiamo sapere se si tratti di informazioni non note o di informazioni soppresse)
- Di solito, il verso di navigazione rappresenta una scelta di progetto, per cui non è presente nei diagrammi concettuali



# VERSO DI NAVIGAZIONE

- Implementazione possibile:

```
class Persona{  
    Automobile automobilePosseduta;  
}  
class Automobile{  
}
```



- Implementazione errata:

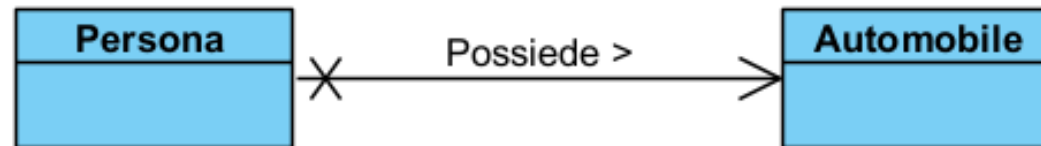
```
class Persona{  
}  
class Automobile{  
    Persona possessore;  
}
```

- Data la persona non è possibile risalire alla sua automobile!



# VERSO DI NAVIGAZIONE

- Più recentemente è stato introdotto un ulteriore simbolo per indicare il verso nel quale una associazione NON può essere navigata

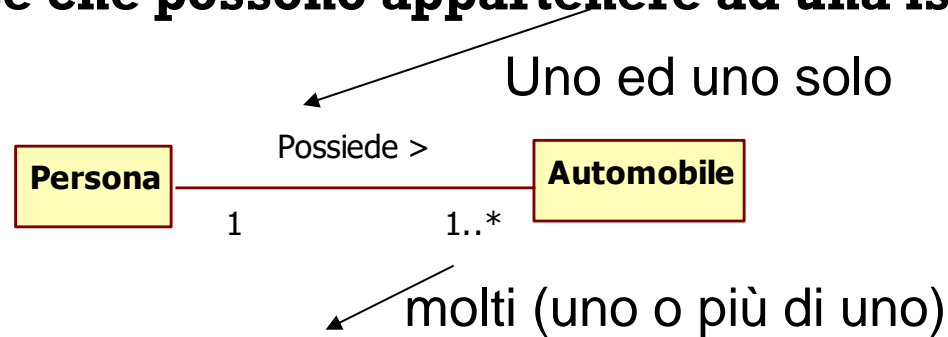


- Data la persona, è possibile conoscere l'automobile che possiede
- Data l'automobile NON è possibile conoscere la persona che la possiede



# MOLTEPLICITÀ DELLE ASSOCIAZIONI

- La molteplicità delle associazioni indica il numero di istanze di oggetti di ogni classe che possono appartenere ad una istanza della associazione



- **Nell'esempio,**
  - una **Persona** possiede almeno una **Automobile**
    - evidentemente le persone che non possiedono **Automobile** non fanno parte del problema in oggetto
  - Un'**Automobile** può essere posseduta da una e una sola **Persona**
    - Evidentemente, non è nel problema in oggetto il mantenimento di informazioni riguardo i proprietari di automobili di seconda, terza mano, etc.
  - Quest'esempio si configura come associazione uno a molti (una **Persona**, molte **Automobili**)



# MOLTEPLICITÀ DELLE ASSOCIAZIONI

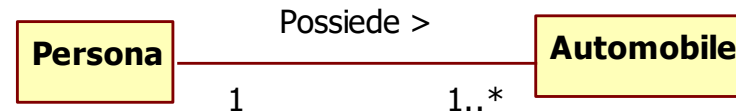
## ■ Possibile implementazione:

```
class Persona{
    ArrayList<Automobile> automobiliPossedute;
    Persona(Automobile a){
        automobilePosseduta.add(a);
        a.proprietario = this; }
    Persona(){
        automobilePosseduta.add(
            new Automobile (this));}
}
```

```
class Automobile{
    Automobile (Persona p){
        proprietario=p;
    }
    public Persona proprietario;
}
```

Il primo costruttore istanzia una persona affidandogli il possesso di un'automobile esistente della quale diventa proprietario.

Il secondo istanzia una persona affidandogli il possesso di una nuova automobile della quale diventa proprietario



In tutti i casi non esisteranno mai persone senza almeno un'automobile e automobili senza proprietario

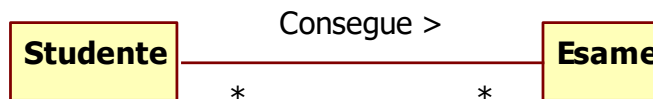




# ASSOCIAZIONI MOLTI A MOLTI

## ▪ Associazioni multi-a-molti

- Uno studente può conseguire un numero potenzialmente non limitato di esami
- Un esame può essere conseguito da un numero potenzialmente non limitato di studenti
- Possono esserci studenti che non hanno conseguito esami
- Possono esserci esami non conseguiti (ancora) da nessuno studente



Possibile implementazione:

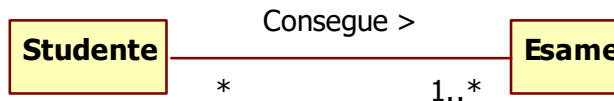
```
class Studente{
    ArrayList<Esame> esameConseguito; }
class Esame{
    ArrayList<Studente> studenteEsaminato; }
```



# ASSOCIAZIONI MOLTI A MOLTI

## ■ Associazioni multi-a-molti

- Se avessimo voluto modellare il caso in cui uno studente era considerato solo dal momento del conseguimento del primo esame, allora sarebbe stato



Possibile implementazione:

```
class Studente{
    Studente(Esame e) {
        esameConseguito.add(e) ;
        e.studenteEsaminato.add (this) ;
    }

    ArrayList<Esame> esameConseguito; }
class Esame{
    public ArrayList<Studente> studenteEsaminato; }
```

Soluzione valida se  
studenteEsaminato è public,  
altrimenti è necessario un metodo  
pubblico di Esame che permetta di  
aggiungere uno studente  
esaminato



# ASSOCIAZIONI UNO A UNO

## ▪ Uno-a-uno

- Ogni studente ha uno e un solo badge
  - Non è possibile modellare, in caso di smarrimento e rilascio di un nuovo badge, l'elenco di tutti i badge avuti da uno studente nel tempo
- Un badge identifica uno e un solo studente



## ▪ Possibile implementazione:

```
class Studente{
    Badge badge;
    Studente(){ badge = new Badge(); badge.studente = this}
}
class Badge{
    Studente studente;
    Badge (){ studente = new Studente (); studente.badge = this}
}
```

Una possibile soluzione è  
quella simmetrica



# ASSOCIAZIONI UNO A UNO

## ■ Uno-a-uno

- Se avessimo voluto considerare anche studenti che, magari temporaneamente, non abbiano un badge, allora diventa:



La soluzione simmetrica qui non è ugualmente valida

## ■ Possibile implementazione:

```
class Studente{
    Badge badge;
}
```

```
class Badge{
    Studente studente;
    Badge (Studente s){ studente=s; s.badge = this;}
}
```

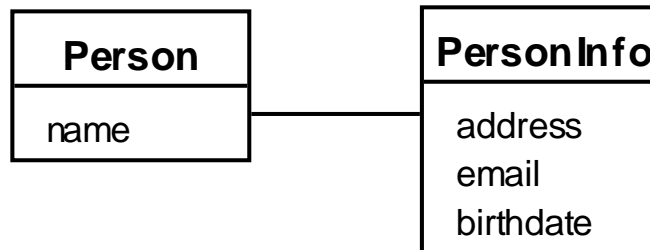
Da notare che lo studente può esistere anche senza Badge, quindi il costruttore di Badge non deve necessariamente istanziare un nuovo studente



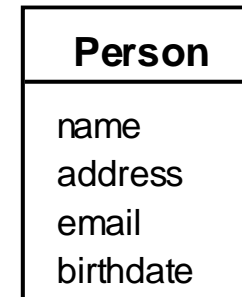
# CONTROESEMPI

- A volte le associazioni uno a uno sono inutili

Evitare



migliore soluzione!



# ENUMERATION

- Le enumerazioni sono utilizzate, anche in Java, per indicare un elenco di possibili valori di un dato
- In UML si indicano con lo stesso simbolo delle classi, con la sovraindicazione <<enumeration>>
- I tipi enumerativi possono essere collegati alle classi col simbolo dell'associazione (di solito non si indicano ruoli o cardinalità)
- Negli attributi della classe si può indicare il tipo enumerativo



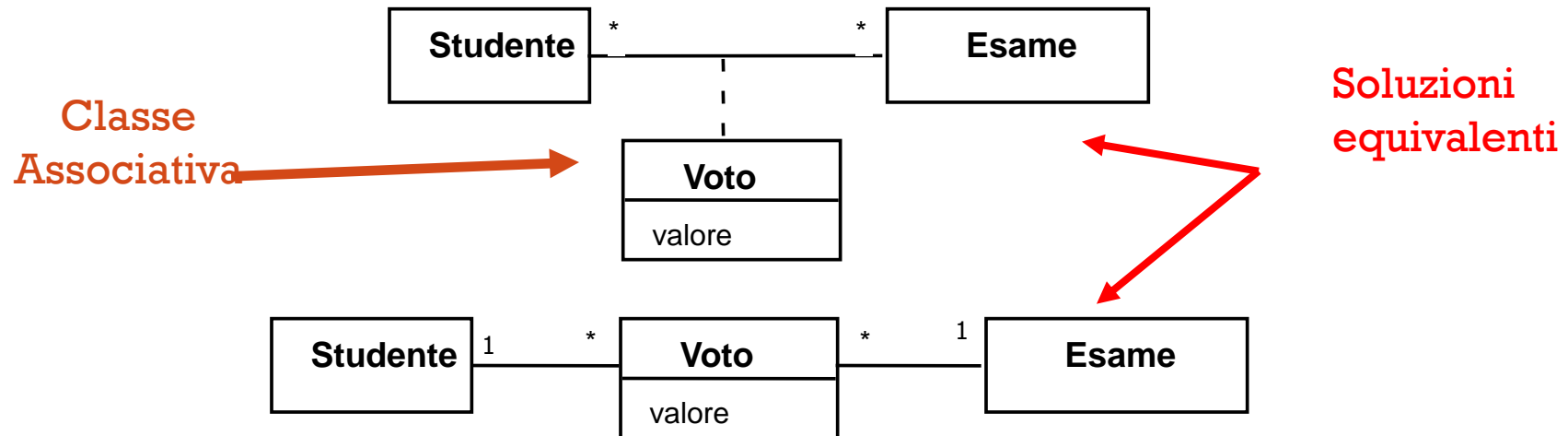
# UN ESEMPIO PIÙ COMPLESSO

- Una prenotazione si riferisce sempre ad un solo passeggero
  - Non esistono prenotazioni con zero passeggeri
    - Ciò implica che prima di creare una prenotazione deve esistere un passeggero
  - Una prenotazione non può mai riferirsi a più di un passeggero.
- Un Passeggero può avere più prenotazioni
  - Un passeggero potrebbe avere zero prenotazioni
  - Un passeggero potrebbe avere più di una prenotazione
- Una prenotazione si riferisce ad un volo
- Un volo può avere più passeggeri prenotati



# CLASSI ASSOCIATIVE

- In alcuni casi, un attributo che si riferisce a due classi collegate non può essere riferito a nessuna delle due
- Può esistere nel caso di molteplicità multi-a-molti



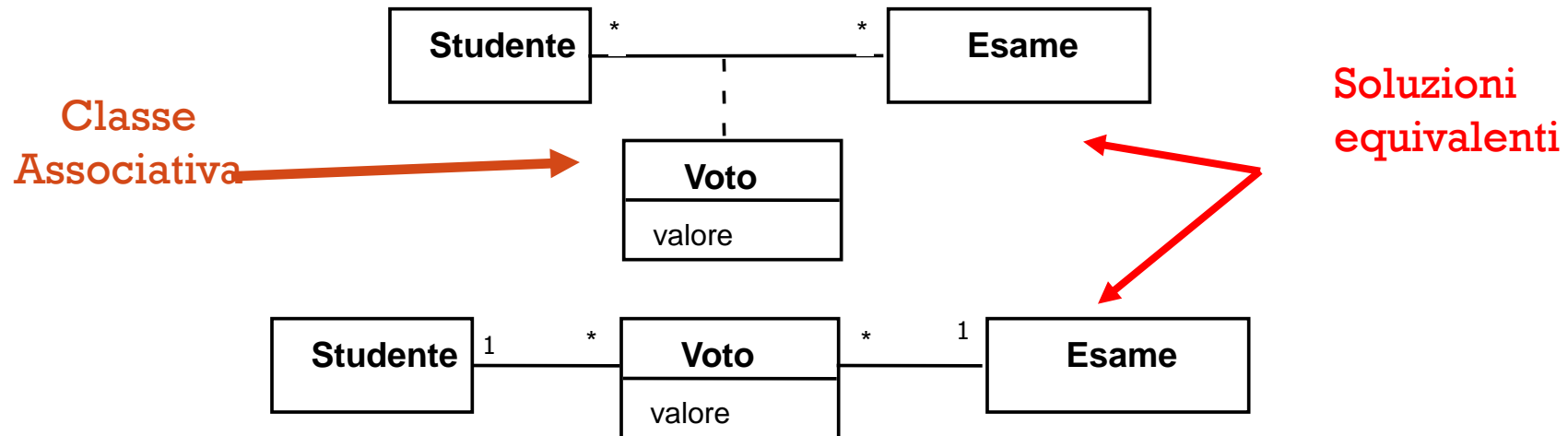
- Prima di creare un'istanza della classe Voto, devono esistere le istanze delle classi collegate
- La classe associativa può avere attributi, metodi, altre associazioni





# CLASSI ASSOCIATIVE

- In alcuni casi, un attributo che si riferisce a due classi collegate non può essere riferito a nessuna delle due
- Può esistere nel caso di molteplicità multi-a-molti

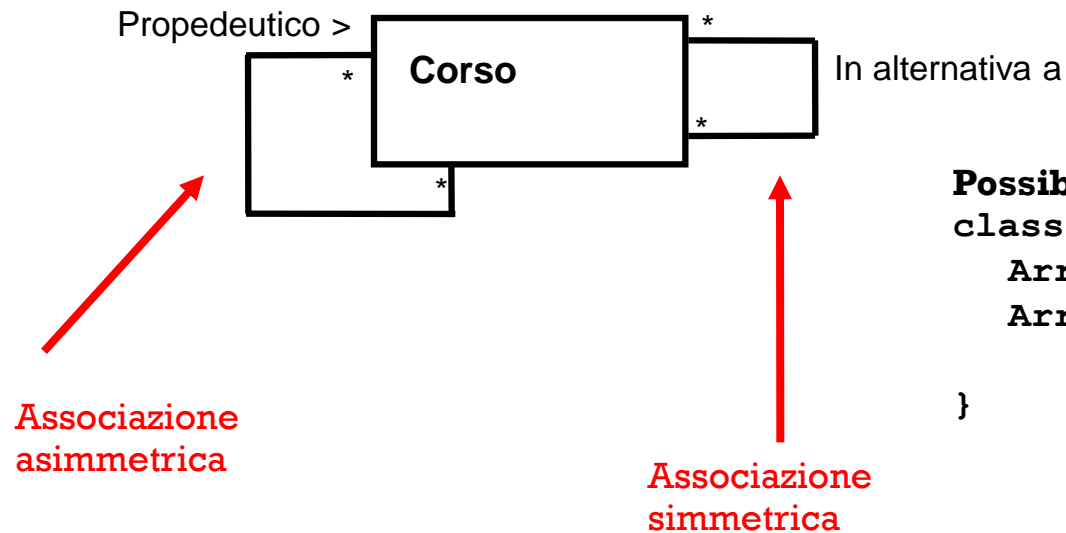


```
Studente s=new Studente();
Esame e=new Esame();
Voto v=new Voto(s,e,30);
```



# ASSOCIAZIONI RIFLESSIVE

- Associazioni che collegano una classe con se' stessa



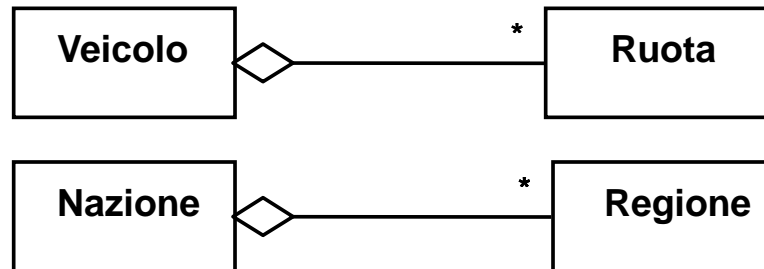
**Possibile implementazione:**

```
class Corso{  
    ArrayList<Corso> corsoPropedeutico;  
    ArrayList<Corso> corsoAlternativo;  
}
```



# AGGREGAZIONE

- Le Aggregazioni sono speciali associazioni che rappresentano una relazione 'tutto-parti'.
  - Il lato del 'tutto' è spesso chiamato *l'aggregato*
  - La molteplicità dal lato del tutto, quando è sottintesa, vale 0..1
    - Il rombo vuoto indica 0..1



# QUANDO USARE UNA AGGREGAZIONE

- Una associazione diventa una aggregazione se:
  - È possibile affermare che:
    - Le parti sono 'parte di' un insieme
    - L'aggregato è 'composto da' parti
- Quando qualcosa possiede o controlla l'aggregato, allora esso possiede o controlla anche le sue parti
  - Per quanto le aggregazioni siano importanti dal punto di vista della espressività del modello, spesso sono implementate in maniera identica rispetto ad associazioni di pari cardinalità



# POSSIBILE IMPLEMENTAZIONE

```
class Veicolo{  
    ArrayList<Ruota> ruota;  
}  
class Ruota{  
    Veicolo v=null;  
}
```

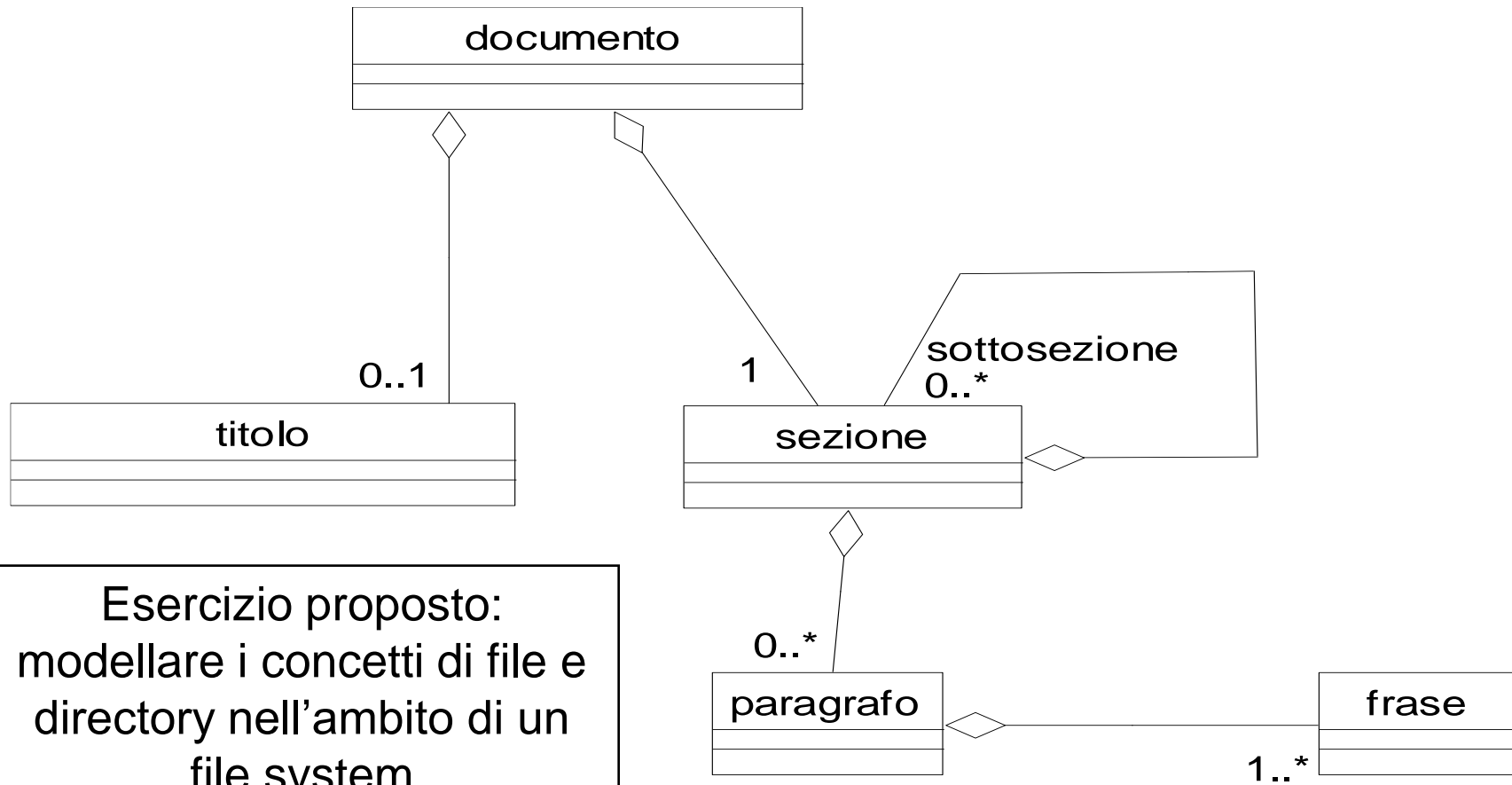
L'attributo v in Ruota è consigliato perché una ruota può essere in un veicolo e così possiamo navigare direttamente questa relazione.

Ma potrebbe trattarsi anche di una ruota di scorta che non è in un veicolo.

L'attributo ruota in Veicolo è invece obbligatorio perché esprime il fatto che un veicolo aggrega ruote



# UNA GERARCHIA DI AGGREGAZIONE

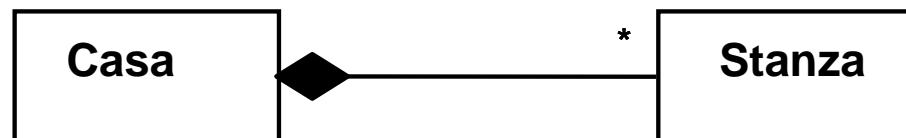


Esercizio proposto:  
modellare i concetti di file e  
directory nell'ambito di un  
file system



# COMPOSIZIONE

- Una *composizione* è una forma forte di aggregazione
  - Se l'aggregato viene distrutto, anche le sue parti saranno distrutte (le parti non esistono senza il tutto)
  - In questo dominio ipotizziamo che non abbia senso parlare di stanze fintantochè esse non siano state legate alla casa in cui si trovano
  - La cardinalità dell'aggregazione, se sottintesa, vale 1
    - Il rombo pieno indica 1



# POSSIBILE IMPLEMENTAZIONE

```
class Casa{  
    ArrayList<Stanza> stanza;  
}  
  
class Stanza{  
    Stanza(Casa casa){  
        c=casa;  
    }  
    Casa c;  
}
```

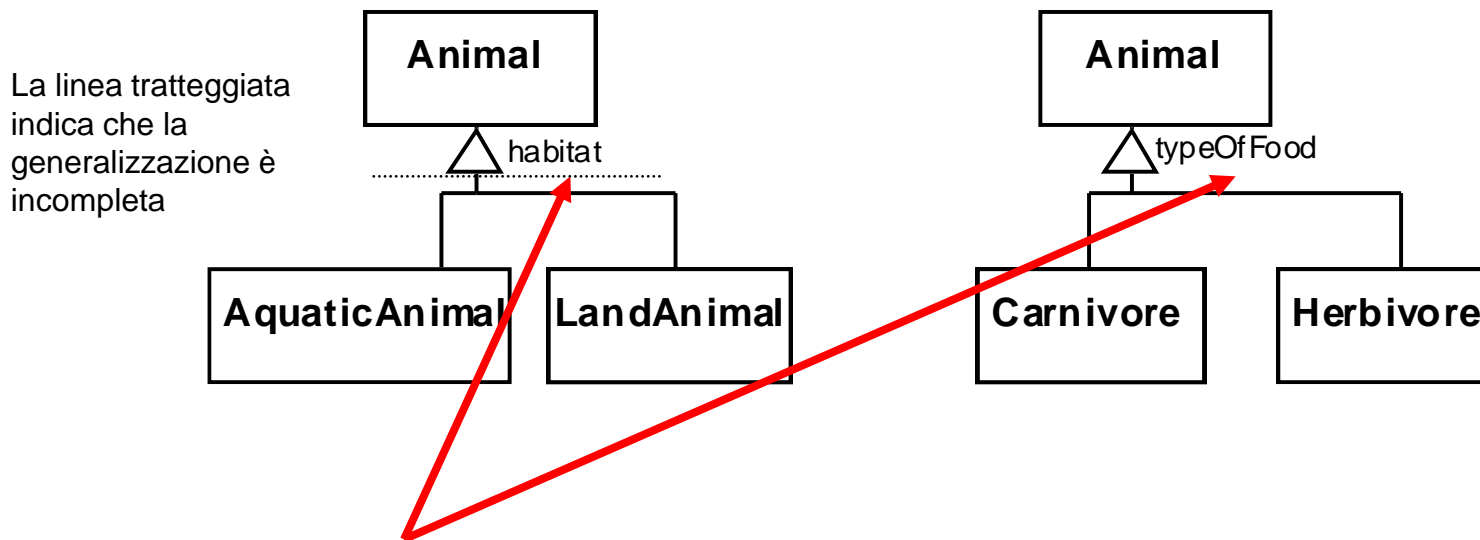
Rispetto all'esempio  
precedente ora  
l'attributo c in Stanza  
è obbligatorio e deve  
essere inizializzato dal  
costruttore, cosicchè  
non possa esistere una  
Stanza che non sia in  
associazione con una  
casa





# GENERALIZZAZIONI

- I concetti di generalizzazione e specializzazione UML sono del tutto analoghi a quelli object oriented



- Un discriminatore potrà essere implementato come un attributo con valori diversi nelle due sottoclassi



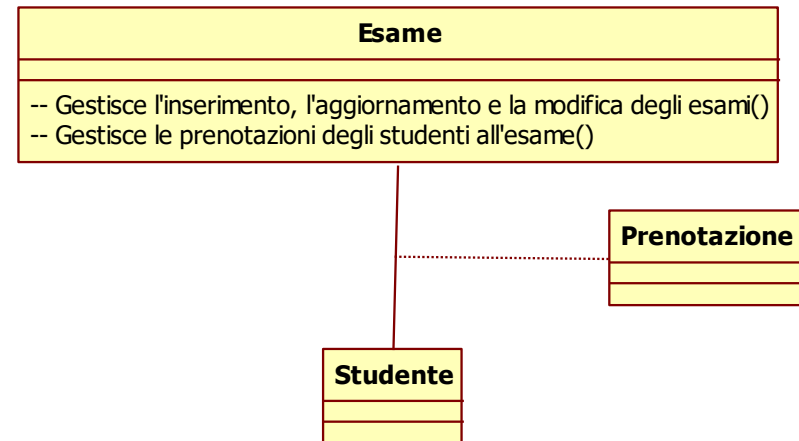
# GENERALIZZAZIONE

- I concetti di generalizzazione e specializzazione UML sono del tutto analoghi a quelli object oriented
- A livello concettuale, una gerarchia di generalizzazione esprime una relazione is-a tra un concetto generale e le sue specializzazioni
- A livello di progetto di dettaglio, invece, può essere interpretato:
  - Come una relazione di ereditarietà tra due classi concrete
    - Le classi derivate (figlie) ereditano attributi e metodi public e protected dalla classe padre



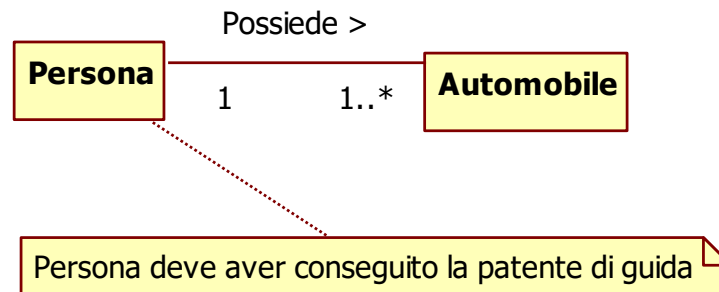
# RESPONSABILITÀ

- Nei diagrammi che descrivono il dominio del problema non si inseriscono di solito metodi perchè essi rappresenterebbero un elemento del dominio della soluzione
- In alternativa si possono utilizzare le Responsabilità
  - Una responsabilità è un insieme di servizi e compiti che la classe dovrebbe garantire
  - Sono utili per controllare la completezza del modello di dominio



# NOTE E TESTO DESCRITTIVO

- Si possono aggiungere ulteriori informazioni direttamente in linguaggio naturale ad un qualsiasi diagramma UML utilizzando le annotazioni
- **Note:**
  - Una nota è un pezzo di testo incluso *in* un diagramma UML
  - È come un commento in un linguaggio di programmazione



# DOMINIO DEL PROBLEMA VS DOMINIO DELLA SOLUZIONE

- **Modello del dominio del problema:**
  - Modello che tiene conto della descrizione dei requisiti del problema da risolvere
    - Senza nessun riferimento ad elementi di quella che sarà la soluzione che poi verrà realizzata per risolvere tale problema
    - Unica eccezione: nel caso in cui c'è una parte di soluzione esistente che deve essere riutilizzata, allora essa può essere parte del dominio del problema
      - ad esempio se bisogna realizzare un software che vada a comunicare con un Vecchio software esistente, le specifiche di quell software saranno considerate nel dominio del problema
- **Modello del dominio della soluzione:**
  - Modello che tiene conto delle scelte tecnologiche e realizzative della soluzione al problema che stiamo realizzando
    - Può modellare ad esempio le classi che realizzeremo direttamente in Java, oppure può essere il modello relazionale di un DB in MySQL



# UN PROCESSO DI ATRAZIONE

1. Identifica un primo insieme di **classi** candidate
2. Aggiungi **associazioni** ed **attributi** a queste classi
3. Trova le **generalizzazioni**
4. Trova le principali **responsabilità** di ogni classe
5. **Itera** il processo finchè il modello ottenuto è soddisfacente



# 1. IDENTIFICARE LE CLASSI

- Quando si sviluppa un domain model si tende a *scoprire* classi che fanno parte del dominio
- Nel lavorare all'interfaccia utente o all'architettura, si tende ad *inventare* classi
  - Necessarie a risolvere un problema di design
  - Si possono inventare classi anche per il domain model!
- Ad una classe dovrebbe corrispondere un'entità del dominio del problema
  - In pratica, bisogna applicare i concetti generali di modellazione Object Oriented



# COME È FATTA UNA BUONA CLASSE DI ANALISI?

- Il suo nome ne rispecchia l'intento.
- è una astrazione ben definita che modella un elemento del dominio del problema
- ha un insieme ridotto e ben definito di responsabilità
- ha una massima coesione interna
  - Una classe è tanto più coesa quanto più riesce ad assolvere da sola alle proprie responsabilità





# ANALISI DEI NOMI

- Semplice tecnica per scoprire le classi del dominio
  - Analizzare la documentazione di partenza, come la descrizione dei requisiti
  - Estrarre *nomi* e *predicati nominali* (aggettivi di nomi)
  - Eliminare nomi che:
    - Sono ridondanti (rappresentano la stessa classe)
    - Rappresentano istanze e non classi
    - Sono vaghi, troppo generici
    - Corrispondono a classi che non sono necessarie al livello considerato
  - Fare attenzioni a classi nel domain model che rappresentano *tipi di utente* o altri attori (servono davvero?)



# CRC CARDS

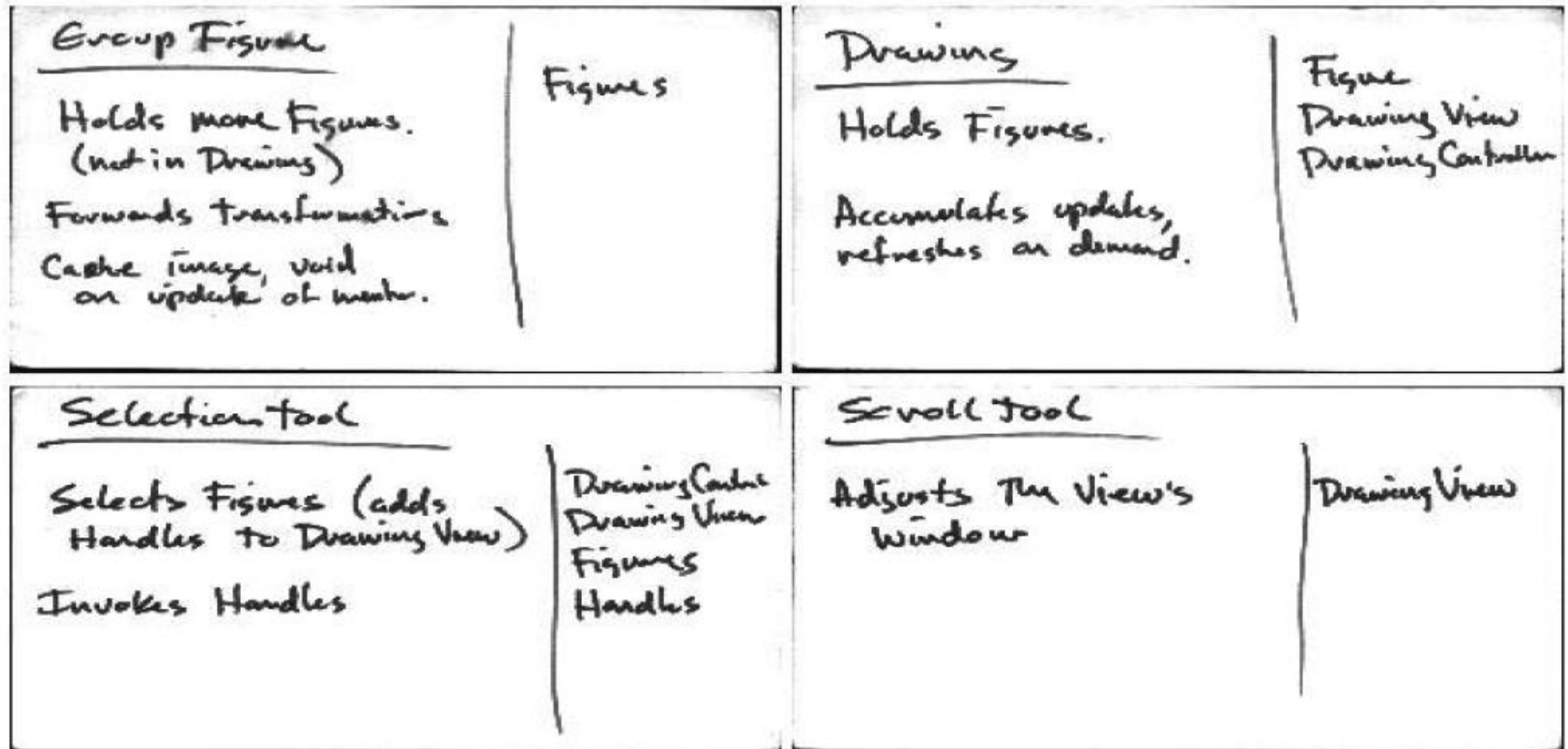
- CRC sta per Class Responsibility Collaboration
- Per ogni classe identificata, porre il nome della classe su una scheda (Card)
- Man mano che vengono individuati attributi e responsabilità, elencarli sulle Card
- Sistemare le card su una lavagna per creare il Class diagram
- Disegnare le linee corrispondenti ad associazioni e generalizzazioni.
  - L'utilizzo delle card serve per imporre, quanto meno psicologicamente, all'analista di non realizzare classi con un numero troppo elevato di attributi e metodi → Se la card è piena allora probabilmente bisogna dividere la classe in due o più classi più semplici



<u>Nome classe</u> - Responsabilità 1 - Responsabilità 2 - Responsabilità 3	Collaboratore 1 Collaboratore 2
--	------------------------------------

**Figura 17.2** Template per una scheda CRC.





**Figura 17.3** Quattro esempi di schede CRC. Questo esempio ha semplicemente lo scopo di mostrare il livello tipico di dettaglio, e non il testo specifico.



## 2. IDENTIFICARE ASSOCIAZIONI ED ATTRIBUTI

- Parti con le classi ritenute **centrali** ed importanti
- Stabilisci i dati ovvi e chiari che esse contengono e le loro relazioni con altre classi.
- Procedi con le classi meno importanti.
- Evita di aggiungere troppi attributi ed associazioni ad una classe.
  - Un sistema è più semplice se manipola meno informazioni
  - Le classi devono avere poche dipendenze da altre classi



# SUGGERIMENTI PER TROVARE E SPECIFICARE ASSOCIAZIONI VALIDE

- Una associazione dovrebbe esistere se una classe:

- *possiede*
- *controlla*
- *è collegata a*
- *si riferisce a*
- *è parte di*
- *ha come parti*
- *è membro di oppure*
- *ha come membri*

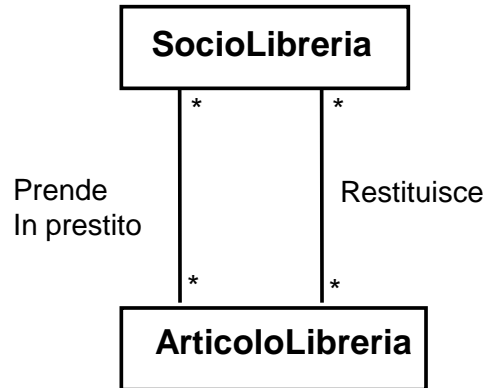
qualche altra classe del modello

- Specificare le molteplicità da entrambi i lati.
- Assegnare un nome chiaro all'associazione.



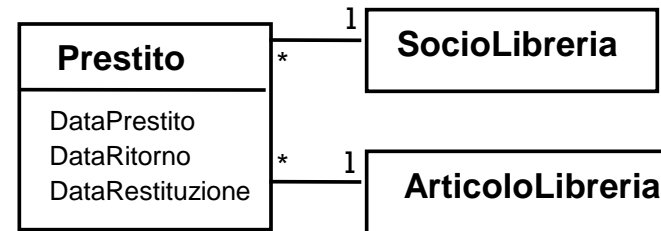
# AZIONI VS ASSOCIAZIONI

- Un errore comune consiste nel considerare *azioni* come se fossero *associazioni*.



Errore!

Nell'implementazione della classe **Prestito** compariranno due riferimenti a un oggetto **SocioLibreria** e un oggetto **ArticoloLibreria**



L'operazione **presta** crea un **Prestito** e

L'operazione **restituisce** setta la data di restituzione

Entrambe le operazioni vanno assegnate alla classe **Prestito**



## 2.A. IDENTIFICARE GLI ATTRIBUTI

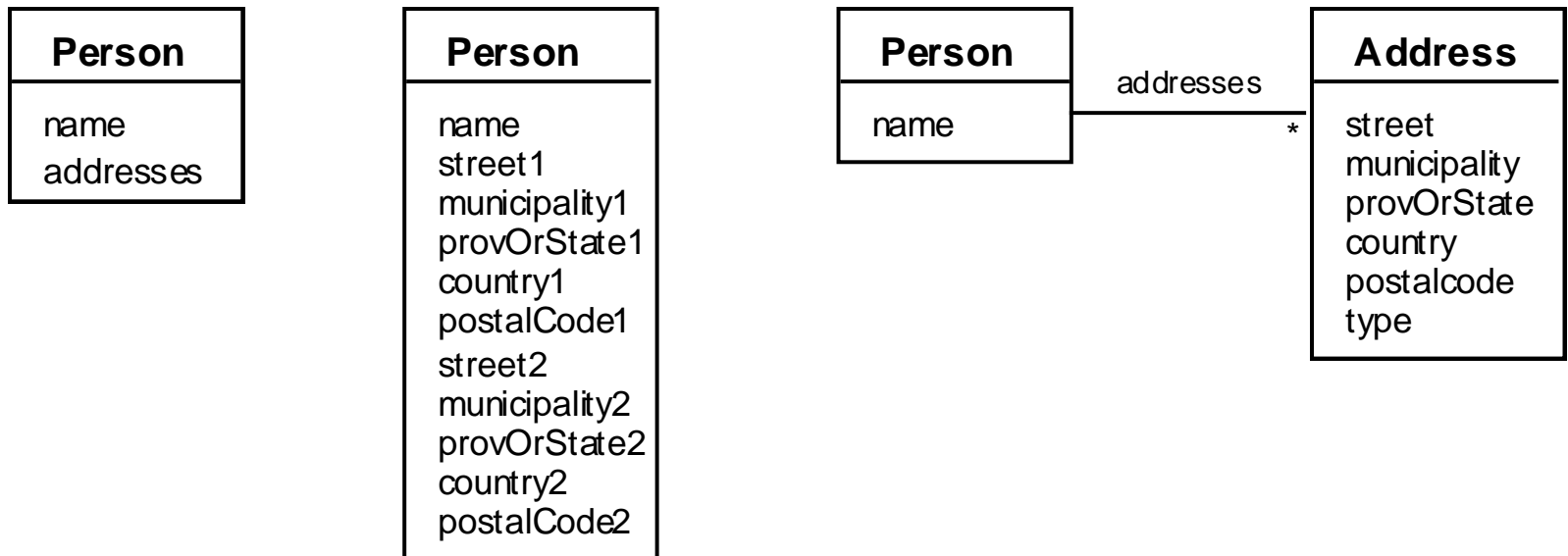
- Cercare le informazioni che devono essere conservate per ciascuna classe
- È possibile che nomi che sono stati scartati come classi, possano ora essere considerati attributi
- Un attributo dovrebbe in genere contenere un solo valore
  - Es. stringa, numerico





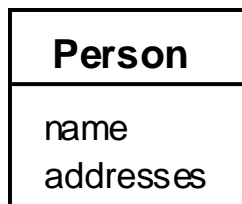
# SUGGERIMENTI PER IDENTIFICARE E SPECIFICARE ATTRIBUTI VALIDI

- È bene non avere molti attributi duplicati
- Se un sottoinsieme degli attributi di una classe forma un gruppo coerente, crea una classe distinta per questi attributi

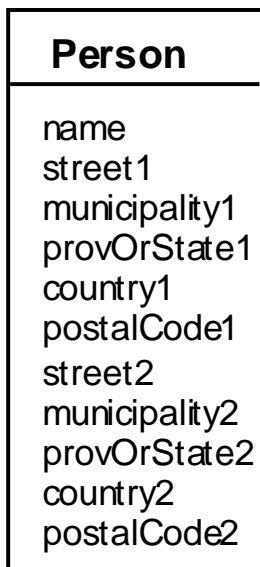


# SUGGERIMENTI PER IDENTIFICARE E SPECIFICARE ATTRIBUTI VALIDI

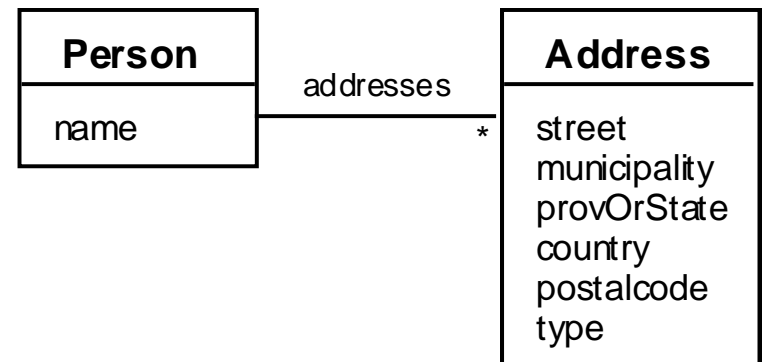
- È bene non avere molti attributi duplicati
- Se un sottoinsieme degli attributi di una classe forma un gruppo coerente, crea una classe distinta per questi attributi



**Errore: uso  
di attributi al  
plurale:  
sostituire  
con address  
[1..\*], ad  
esempio**



**Errore: troppi attributi  
duplicati, e incapacità  
di gestire più indirizzi**



**Bene: l'attributo  
tipo indica il tipo  
di indirizzo**



### 3. IDENTIFICARE GENERALIZZAZIONI E INTERFACCE

- Due modi per trovare le generalizzazioni:
  - bottom-up
    - Raggruppo classi simili creando una nuova superclasse
  - top-down
    - Cerco prima le classi più generali, e poi specializzo



## 4. ASSEGNARE LE RESPONSABILITÀ ALLE CLASSI

- Una *responsabilità* è un qualcosa che è richiesto al sistema.
  - La responsabilità di **ogni requisito funzionale** deve essere attribuita ad una delle classi, anche se tale requisito potrà essere svolto mediante una collaborazione fra più classi.
    - Tutte le responsabilità di una classe dovrebbero essere *chiaramente correlate*.
    - Se una classe ha troppe responsabilità, valutare l'ipotesi di *dividerla* in più classi
    - Se una classe non ha responsabilità, potrebbe essere *inutile*
    - Quando una responsabilità non può essere attribuita a nessuna delle classi esistenti, dovrebbe essere creata una *nuova* classe
- Per stabilire le responsabilità:
  - Cercare verbi e nomi che descrivono *azioni* nella descrizione del sistema



# CATEGORIE DI RESPONSABILITÀ

- Set e get dei valori degli attributi
  - Creare ed inizializzare nuove istanze
  - Prelevare da o memorizzare dati in una memoria persistente
  - Distruggere istanze
  - Aggiungere e cancellare istanze di associazioni
  - Copiare, convertire, trasformare, trasmettere o fornire in output dati.
  - Calcolare risultati numerici
  - Navigare e cercare dati di particolari istanze
  - Altro lavoro specifico
- 
- In un diagramma del dominio (quindi senza elementi del progetto di dettaglio e dell'implementazione) la maggior parte di queste responsabilità non sono indicate
    - Viceversa, le responsabilità riconosciute in fase di analisi dovrebbero essere sempre considerate



# ERRORI COMUNI

- I nomi delle classi sono sempre al singolare
  - da non confondersi con i nomi delle tabelle del database
- Il «Sistema» non è praticamente mai una classe
  - Tutto il modello rappresenta il dominio del problema che verrà risolto dal «sistema»
- Gli «id» non sono quasi mai attributi del dominio del problema
  - «id» è di solito un termine utilizzato per indicare una chiave primaria artificiale di una tabella, che non ha corrispettivi nel dominio del problema
  - Esempio: idStudente non è un vero attributo (nessuno studente è schedato con un numero semplice) mentre numeroMatricola lo è (allo studente viene assegnato un numero di matricola utilizzato per richiedere un servizio, registrare un esame, etc.)



# ESEMPIO: VIDEOGIOCO

- Si vuole progettare un videogioco nel quale il giocatore impersona un agente di borsa, rivivendone le interazioni e basandosi sulle azioni delle società effettivamente quotate alla borsa di New York.
- Ogni giocatore ha una dotazione monetaria iniziale, pari a 100 milioni di dollari, che può essere investita acquistando delle azioni. Possono essere acquistate azioni di ognuna delle società quotate nel listino della borsa di New York.
- Le azioni di una società vengono acquistate al prezzo cui sono quotate nell'istante dell'acquisto. In fase di acquisto, il giocatore acquirente deve specificare anche il quantitativo di azioni che vuole acquistare. Il sistema provvederà a valutare l'effettiva disponibilità di liquidi da parte dell'acquirente e, in caso di acquisto possibile, scalerà dalla sua liquidità il denaro necessario all'acquisto delle azioni.
- Il giocatore potrà anche vendere le azioni che possiede, al prezzo cui sono quotate in quel momento (il sistema provvederà ad aggiornare la liquidità).
- Acquisti e vendite di azioni potranno essere effettuati soltanto nel periodo di tempo tra l'orario di apertura e l'orario di chiusura della borsa di New York.
- I giocatori possono anche scegliere di monitorare le azioni di alcune società. In tal caso il sistema manterrà un valore del prezzo delle azioni di ognuna delle società monitorate per ogni giorno nel quale il giocatore vorrà monitorarla.
- I giocatori sono organizzati in squadre di al più 4 elementi. Ogni squadra ha un nome e il sistema assegnerà un premio mensile alla squadra che ha il bilancio migliore. Una bacheca manterrà tutti i premi mensili ricevuti.



# RICERCA CLASSI

- Si vuole progettare un videogioco nel quale il giocatore impersona un agente di borsa, rivivendone le interazioni e basandosi sulle azioni delle società effettivamente quotate alla borsa di New York.
- Ogni **giocatore** ha una dotazione monetaria iniziale, pari a 100 milioni di dollari, che può essere investita acquistando delle **azioni**. Possono essere acquistate azioni di ognuna delle **società quotate** nel listino della **borsa** di New York.
- Le **azioni** di una **società** vengono acquistate al prezzo cui sono quotate nell'istante dell'**acquisto**. In fase di acquisto, il **giocatore** acquirente deve specificare anche il quantitativo di **azioni** che vuole acquistare. Il sistema provvederà a valutare l'effettiva disponibilità di liquidi da parte dell'acquirente e, in caso di acquisto possibile, scalerà dalla sua liquidità il denaro necessario all'**acquisto** delle **azioni**.
- Il **giocatore** potrà anche vendere le **azioni** che possiede, al prezzo cui sono quotate in quel momento (il sistema provvederà ad aggiornare la liquidità).
- Acquisti e vendite di azioni potranno essere effettuati soltanto nel periodo di tempo tra l'orario di apertura e l'orario di chiusura della **borsa** di New York.
- I **giocatori** possono anche scegliere di monitorare le **azioni** di alcune società. In tal caso il sistema manterrà un valore del prezzo delle **azioni** di ognuna delle **società monitorate** per ogni giorno nel quale il **giocatore** vorrà monitorarla.
- I **giocatori** sono organizzati in **squadre** di al più 4 elementi. Ogni **squadra** ha un nome e il sistema assegnerà un **premio mensile** alla squadra che ha il bilancio migliore. Una **bacheca** manterrà tutti i **premi mensili** ricevuti.





# RICERCA ASSOCIAZIONI ED ATTRIBUTI

- Si vuole progettare un videogioco nel quale il giocatore impersona un agente di borsa, rivivendone le interazioni e basandosi sulle azioni delle società effettivamente quotate alla borsa di New York.
- Ogni **giocatore** ha una **dotazione monetaria iniziale**, pari a 100 milioni di dollari, che può essere investita acquistando delle **azioni**. Possono essere acquistate azioni di ognuna delle **società quotate** nel **listino della borsa** di New York.
- Le **azioni** di una **società** vengono acquistate al **prezzo** cui sono quotate nell'**istante** dell'**acquisto**. In fase di acquisto, il **giocatore** acquirente deve specificare anche il **quantitativo** di **azioni** che vuole acquistare. Il sistema provvederà a valutare l'effettiva **disponibilità di liquidi** da parte dell'acquirente e, in caso di acquisto possibile, scalerà dalla sua **liquidità** il **denaro necessario** all'**acquisto** delle **azioni**.
- Il **giocatore** potrà anche vendere le **azioni** che **possiede**, al **prezzo** cui sono quotate in quel momento (il sistema provvederà ad aggiornare la **liquidità**).
- Acquisti e vendite di azioni potranno essere effettuati soltanto nel periodo di tempo tra **l'orario di apertura** e **l'orario di chiusura** della **borsa** di New York.
- I **giocatori** possono anche scegliere di monitorare le **azioni** di alcune società. In tal caso il sistema manterrà un valore del **prezzo** delle **azioni** di ognuna delle **società monitorate** per ogni giorno nel quale il **giocatore** vorrà monitorarla.
- I **giocatori** sono **organizzati in squadre** di al più 4 elementi. Ogni **squadra** ha un **nome** e il sistema **assegnerà** un **premio mensile** alla squadra che ha il **bilancio** migliore. Una **bacheca** **manterrà** tutti i **premi mensili** ricevuti.



# RICERCA GENERALIZZAZIONI

- Si vuole progettare un videogioco nel quale il giocatore impersona un agente di borsa, rivivendone le interazioni e basandosi sulle azioni delle società effettivamente quotate alla borsa di New York.
- Ogni **giocatore** ha una **dotazione monetaria iniziale**, pari a 100 milioni di dollari, che può essere investita acquistando delle **azioni**. Possono essere acquistate azioni di ognuna delle **società quotate** nel **listino della borsa** di New York.
- Le **azioni** di una **società** vengono acquistate al **prezzo** cui sono quotate nell'**istante** dell'**acquisto**. In fase di acquisto, il **giocatore** acquirente deve specificare anche il **quantitativo** di **azioni** che vuole acquistare. Il sistema provvederà a valutare l'effettiva **disponibilità di liquidi** da parte dell'acquirente e, in caso di acquisto possibile, scalerà dalla sua **liquidità** il **denaro necessario** all'**acquisto** delle **azioni**.
- Il **giocatore** potrà anche vendere le **azioni** che **possiede**, al **prezzo** cui sono quotate in quel momento (il sistema provvederà ad aggiornare la **liquidità**).
- Acquisti e vendite di azioni potranno essere effettuati soltanto nel periodo di tempo tra l'**orario di apertura** e l'**orario di chiusura** della **borsa** di New York.
- I **giocatori** possono anche scegliere di monitorare le **azioni** di alcune società. In tal caso il sistema manterrà un valore del **prezzo** delle **azioni** di ognuna delle **società monitorate** per ogni giorno nel quale il **giocatore** vorrà monitorarla.
- I **giocatori** sono **organizzati in squadre** di al più 4 elementi. Ogni **squadra** ha un **nome** e il sistema **assegnerà** un **premio mensile** alla squadra che ha il **bilancio** migliore. Una **bacheca** **manterrà** tutti i **premi mensili** ricevuti.



# RICERCA E ASSEGNAZIONE DI RESPONSABILITÀ

- Si vuole progettare un videogioco nel quale il giocatore impersona un agente di borsa, rivivendone le interazioni e basandosi sulle azioni delle società effettivamente quotate alla borsa di New York.
- Ogni **giocatore** ha una **dotazione monetaria iniziale**, pari a 100 milioni di dollari, che può **essere investita acquistando** delle **azioni**. **Possono essere acquistate azioni** di ognuna delle **società quotate** nel **listino della borsa** di New York.
- Le **azioni** di una **società vengono acquistate** al **prezzo** cui sono quotate nell'**istante** dell'**acquisto**. In fase di **acquisto**, il **giocatore** acquirente deve specificare anche il **quantitativo** di **azioni** che vuole acquistare. Il sistema provvederà a valutare l'effettiva **disponibilità di liquidi** da parte dell'acquirente e, in caso di acquisto possibile, **scalerà** dalla sua **liquidità** il **denaro necessario** all'**acquisto** delle **azioni**.
- Il **giocatore** potrà anche **vendere** le **azioni** che **possiede**, al **prezzo** cui sono quotate in quel momento (il sistema provvederà ad **aggiornare** la **liquidità**).
- Acquisti e vendite di azioni potranno essere effettuati soltanto nel periodo di tempo tra **l'orario di apertura** e **l'orario di chiusura** della **borsa** di New York.
- I **giocatori** possono anche **scegliere di monitorare** le **azioni** di alcune società. In tal caso il sistema manterrà un valore del **prezzo** delle **azioni** di ognuna delle **società monitorate** per ogni giorno nel quale il **giocatore** vorrà monitorarla.
- I **giocatori sono organizzati in squadre** di al più 4 elementi. Ogni **squadra** ha un **nome** e il sistema **assegnerà** un **premio mensile** alla squadra che ha il **bilancio** migliore. Una **bacheca manterrà** tutti i **premi mensili** ricevuti.



# ESEMPIO: AZIENDA ALIMENTARE

- Una azienda produttrice di prodotti alimentari, vuole organizzare un sistema informativo aziendale. Tutti gli utenti dell'applicazione devono essere in grado di visualizzare informazioni relative al catalogo dei prodotti. Inoltre i dipendenti devono essere in grado di accedere ad informazioni relative alle loro mansioni.
- I prodotti sono organizzati in linee di prodotto che accomunano prodotti dello stesso tipo: ad esempio, due linee possono essere pasta e sughi. I prodotti possono essere confezionati in diversi stabilimenti. I dipendenti si suddividono in diverse categorie: i manager, che sono responsabili di una o più linee di produzione (ogni linea é però gestita esattamente da tre manager, per assicurare una gestione equa), i supervisori della produzione che sono responsabili di tutti i prodotti di una specifica linea in uno specifico stabilimento, e gli operai che lavorano su uno specifico prodotto in un determinato stabilimento.
- L'applicazione deve consentire ai manager di accedere alle informazioni relative ai dipendenti di cui sono responsabili, e a tutti di accedere alle informazioni sui prodotti.



# RICERCA CLASSI

- Una azienda produttrice di prodotti alimentari, vuole organizzare un sistema informativo aziendale. Tutti gli utenti dell'applicazione devono essere in grado di visualizzare informazioni relative al **catalogo** dei prodotti. Inoltre i **dipendenti** devono essere in grado di accedere ad informazioni relative alle loro mansioni.
- I **prodotti** sono organizzati in **linee di prodotto** che accomunano **prodotti** dello stesso tipo: ad esempio, due linee possono essere pasta e sughi. I **prodotti** possono essere confezionati in diversi **stabilimenti**. I **dipendenti** si suddividono in diverse categorie: i **manager**, che sono responsabili di una o più **linee di produzione** (ogni **linea** é però gestita esattamente da tre **manager**, per assicurare una gestione equa), i **supervisori della produzione** che sono responsabili di tutti i **prodotti** di una specifica **linea** in uno specifico **stabilimento**, e gli **operai** che lavorano su uno specifico **prodotto** in un determinato **stabilimento**.
- L'applicazione deve consentire ai **manager** di accedere alle informazioni relative ai **dipendenti** di cui sono responsabili, e a tutti di accedere alle informazioni sui **prodotti**.



# RICERCA ASSOCIAZIONI ED ATTRIBUTI

- Una azienda produttrice di prodotti alimentari, vuole organizzare un sistema informativo aziendale. Tutti gli utenti dell'applicazione devono essere in grado di visualizzare informazioni relative al **catalogo dei prodotti**. Inoltre i **dipendenti** devono essere in grado di accedere ad informazioni relative alle loro **mansioni**.
- I **prodotti sono organizzati in linee di prodotto** che accomunano **prodotti** dello stesso **tipo**: ad esempio, due linee possono essere pasta e sughi. I prodotti **possono essere confezionati in diversi stabilimenti**. I **dipendenti** si suddividono in diverse categorie: i **manager**, che sono responsabili di una o più linee di produzione (ogni **linea é però gestita esattamente da tre manager**, per assicurare una gestione equa), i **supervisor della produzione** che **sono responsabili di tutti i prodotti di una specifica linea in uno specifico stabilimento**, e gli **operai** che **lavorano su uno specifico prodotto in un determinato stabilimento**.
- L'applicazione deve consentire ai **manager** di accedere alle informazioni relative ai **dipendenti** di cui **sono responsabili**, e a tutti di accedere alle informazioni sui **prodotti**.



# RICERCA GENERALIZZAZIONI

- Una azienda produttrice di prodotti alimentari, vuole organizzare un sistema informativo aziendale. Tutti gli utenti dell'applicazione devono essere in grado di visualizzare informazioni relative al **catalogo dei prodotti**. Inoltre i **dipendenti** devono essere in grado di accedere ad informazioni relative alle loro **mansioni**.
- I **prodotti sono organizzati in linee di prodotto** che accomunano **prodotti** dello stesso **tipo**: ad esempio, due linee possono essere pasta e sughi. I prodotti **possono essere confezionati in diversi stabilimenti**. I **dipendenti** si suddividono in diverse categorie: i **manager**, che sono responsabili di una o più linee di produzione (ogni **linea é però gestita esattamente da tre manager**, per assicurare una gestione equa), i **supervisor della produzione** che **sono responsabili di tutti i prodotti di una specifica linea in uno specifico stabilimento**, e gli **operai** che **lavorano su uno specifico prodotto in un determinato stabilimento**.
- L'applicazione deve consentire ai **manager** di accedere alle informazioni relative ai **dipendenti** di cui **sono responsabili**, e a tutti di accedere alle informazioni sui **prodotti**.



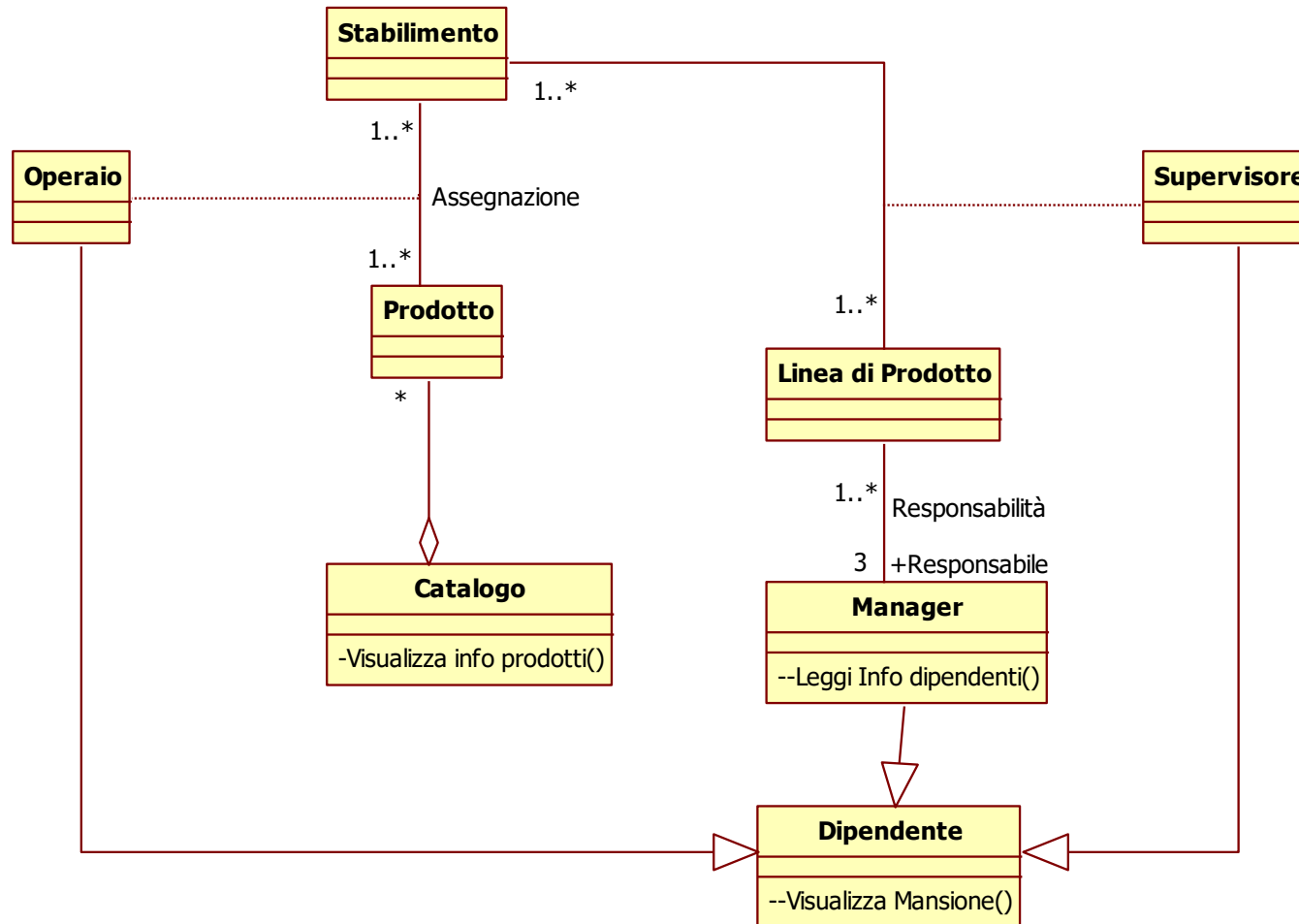
# RICERCA E ASSEGNAZIONE DI RESPONSABILITÀ

- Una azienda produttrice di prodotti alimentari, vuole organizzare un sistema informativo aziendale. Tutti gli utenti dell'applicazione devono essere in grado di visualizzare informazioni relative al catalogo dei prodotti. Inoltre i dipendenti devono essere in grado di accedere ad informazioni relative alle loro mansioni.
- I prodotti sono organizzati in linee di prodotto che accomunano prodotti dello stesso tipo: ad esempio, due linee possono essere pasta e sughi. I prodotti possono essere confezionati in diversi stabilimenti. I dipendenti si suddividono in diverse categorie: i manager, che sono responsabili di una o più linee di produzione (ogni linea é però gestita esattamente da tre manager, per assicurare una gestione equa), i supervisori della produzione che sono responsabili di tutti i prodotti di una specifica linea in uno specifico stabilimento, e gli operai che lavorano su uno specifico prodotto in un determinato stabilimento.
- L'applicazione deve consentire ai manager di accedere alle informazioni relative ai dipendenti di cui sono responsabili, e a tutti di accedere alle informazioni sui prodotti.





# MODELLO DI DOMINIO DEL PROBLEMA





# COMPLEMENTI SUI CLASS DIAGRAM

# DEPENDENCY

- Una Dependency rappresenta una relazione tra le istanze di due classi, che viene a realizzarsi a tempo di esecuzione.
- Esempi: c'è dependency dalla classe A verso la classe B se:
  - Metodi della classe A vanno a leggere/scrivere/modificare il valore di attributi di oggetti della classe B;
  - Metodi della classe A invocano metodi della classe B;
    - caso particolare: la classe A istanzia oggetti della classe B (ovvero ne invoca il costruttore)
- La Dependency si rappresenta con una **linea tratteggiata** che termina con una freccia dall'oggetto dipendente verso quello da cui dipende.



# DEPENDENCY

- **Le relazioni di dependency sono individuate principalmente durante la fase di progetto di dettaglio: esse raramente contribuiscono al modello concettuale**
- Esprimono una dipendenza (**accoppiamento**) tra le classi, nel senso che la classe origine della dipendenza non potrà essere riusata senza la classe destinazione della dependency. Una modifica nella classe da cui c'è dipendenza implicherà modifiche anche nella classe dipendente.
- Analogamente, la correttezza della classe da cui parte una relazione di dipendenza è condizionata alla verifica della correttezza della classe dipendente



# TIPI DI DIPENDENZE IN UML 2.0

- Per distinguere diverse tipologie di dipendenze, UML mette a disposizione gli *stereotipi*, che possono essere indicati con keywords scritti tra parentesi angolari doppie

- Alcuni esempi di stereotipi:

<<call>>

Chiamata di metodo

<<create>>

Creazione di un'istanza di oggetto

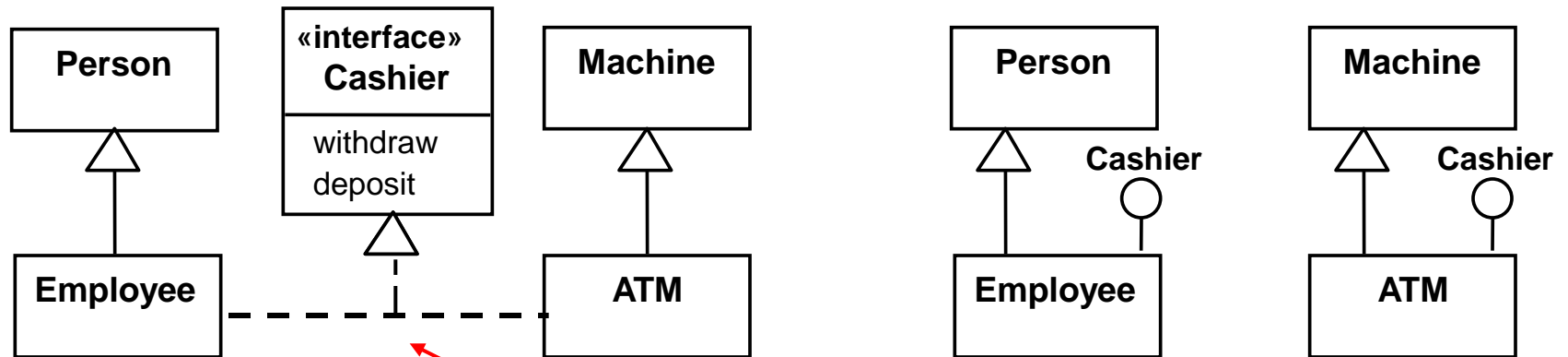
<<use>>

Utilizza un attributo



# INTERFACCIA

- Un'interfaccia descrive una porzione del comportamento visibile di un insieme di oggetti.
  - Un' *interfaccia* è simile ad una classe, tranne che essa non possiede variabili d'istanza nè metodi implementati.
  - Una o più classi possono fornire l'implementazione dell'interfaccia.

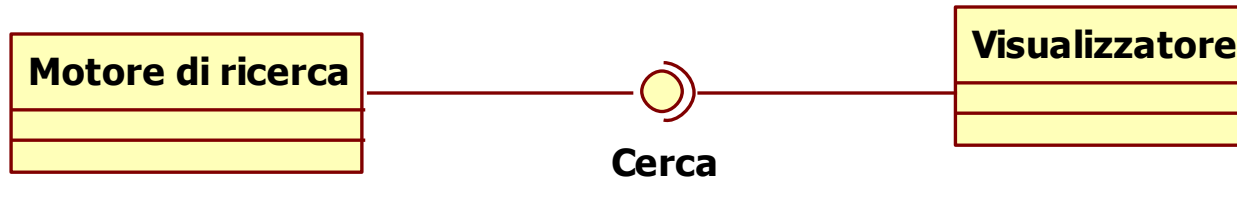


Relazioni di <<realizzazione>>



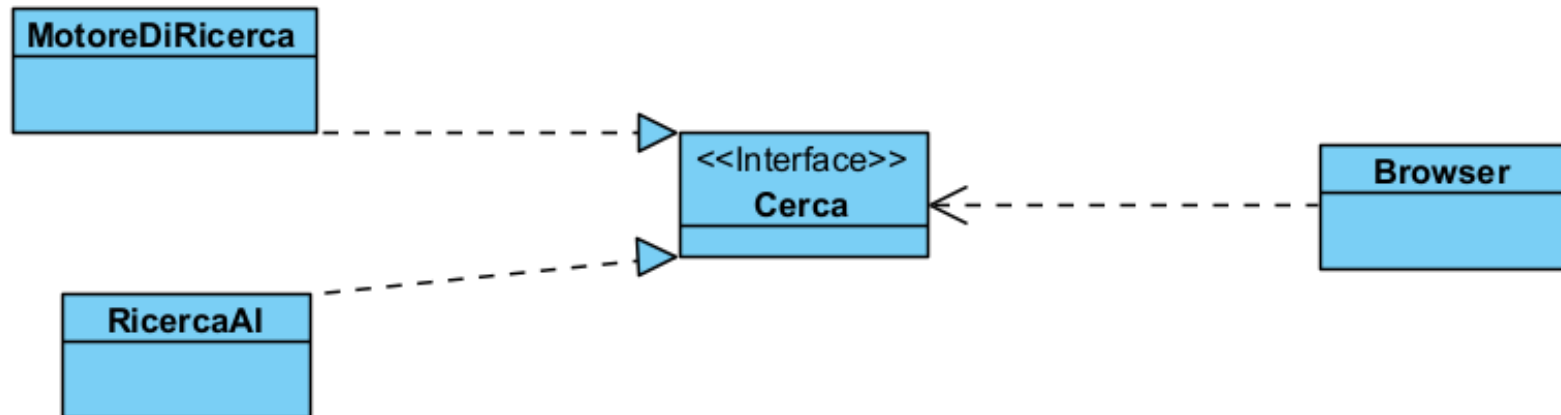
# NOTAZIONE “LOLLIPOP”

- Una notazione talvolta utilizzata per le interfacce è quella a “Lollipop”:
  - La pallina (lollipop) rappresenta l’interfaccia esposta da una classe (potrebbe coincidere con una responsabilità)
  - La linea dalla classe concreta verso la pallina rappresenta una implementazione dell’interfaccia (realization)
  - Il semicerchio (socket) rappresenta la richiesta dell’operazione fornita dall’interfaccia (è una dependency)
- Esempio :
  - un motore di ricerca fornisce la possibilità di accedere al proprio servizio Cerca tramite un’interfaccia
  - La classe visualizzatore richiama la ricerca



# NOTA

- Visual Paradigm supporta la notazione «lollipop» soltanto nei Component Diagram
- Si può utilizzare comunque la notazione equivalente:







# PACKAGE DIAGRAM



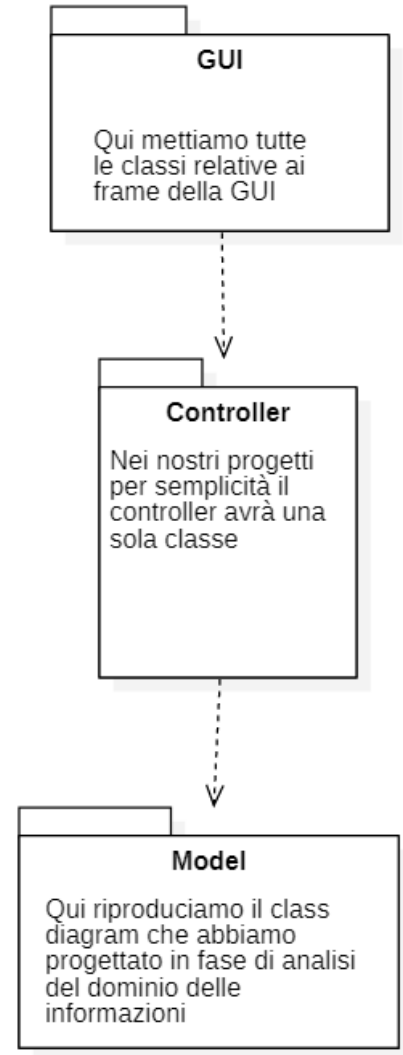
# PACKAGE DIAGRAM

- Si tratta di un diverso tipo di diagramma rispetto al class diagram, anche se ne condivide molti aspetti ed elementi
- In un package diagram compaiono:
  - package
  - le relazioni di dependency tra i package
  - Interface e relazioni di implementation
- Talvolta, package diagram e class diagram possono essere messi in un unico diagramma: in questo caso all'interno di ogni package può essere disegnato un class diagram relative alle classi di quell package

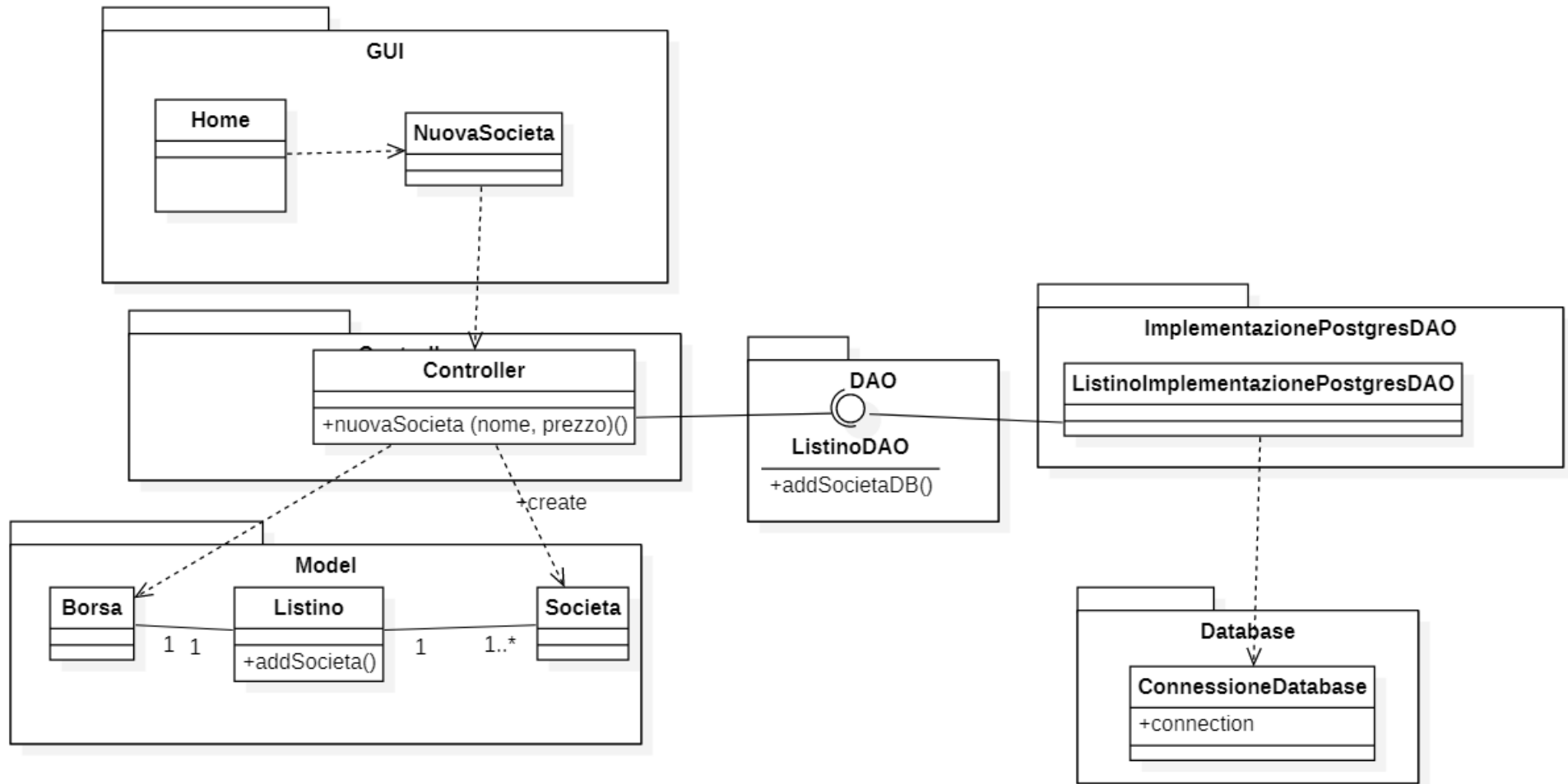


# ESEMPIO DI PACKAGE DIAGRAM

- I package diagram sono di solito utilizzati come diagrammi di **dettaglio** che mostrano la reale implementazione di un Sistema
- Vengono utilizzati quando si sceglie l'architettura del Progetto
- La fase successiva diventa il disegno dei class diagram relative all'implementazione delle classi dei singoli diagrammi



# ESEMPIO DI PACKAGE DIAGRAM CON CLASS DIAGRAM



# SEQUENCE DIAGRAM

UML Distilled, Capitolo 4



# DIAGRAMMI DI INTERAZIONE

- I diagrammi di Interazione sono usati per modellare gli aspetti dinamici di un sistema software, evidenziando in particolare le interazioni tra gli elementi che li compongono (e che sono stati descritti nei diagrammi strutturali)
  - Ci sono quattro tipi di diagrammi di interazione :
    - *Sequence diagrams*
    - *Communication diagrams*
      - *in UML 1 erano chiamati collaboration diagrams*
    - *Interaction Overview Diagrams*
    - *Timing Diagrams*- I diagrammi di interazione appartengono alla famiglia dei *Behaviour Diagram*

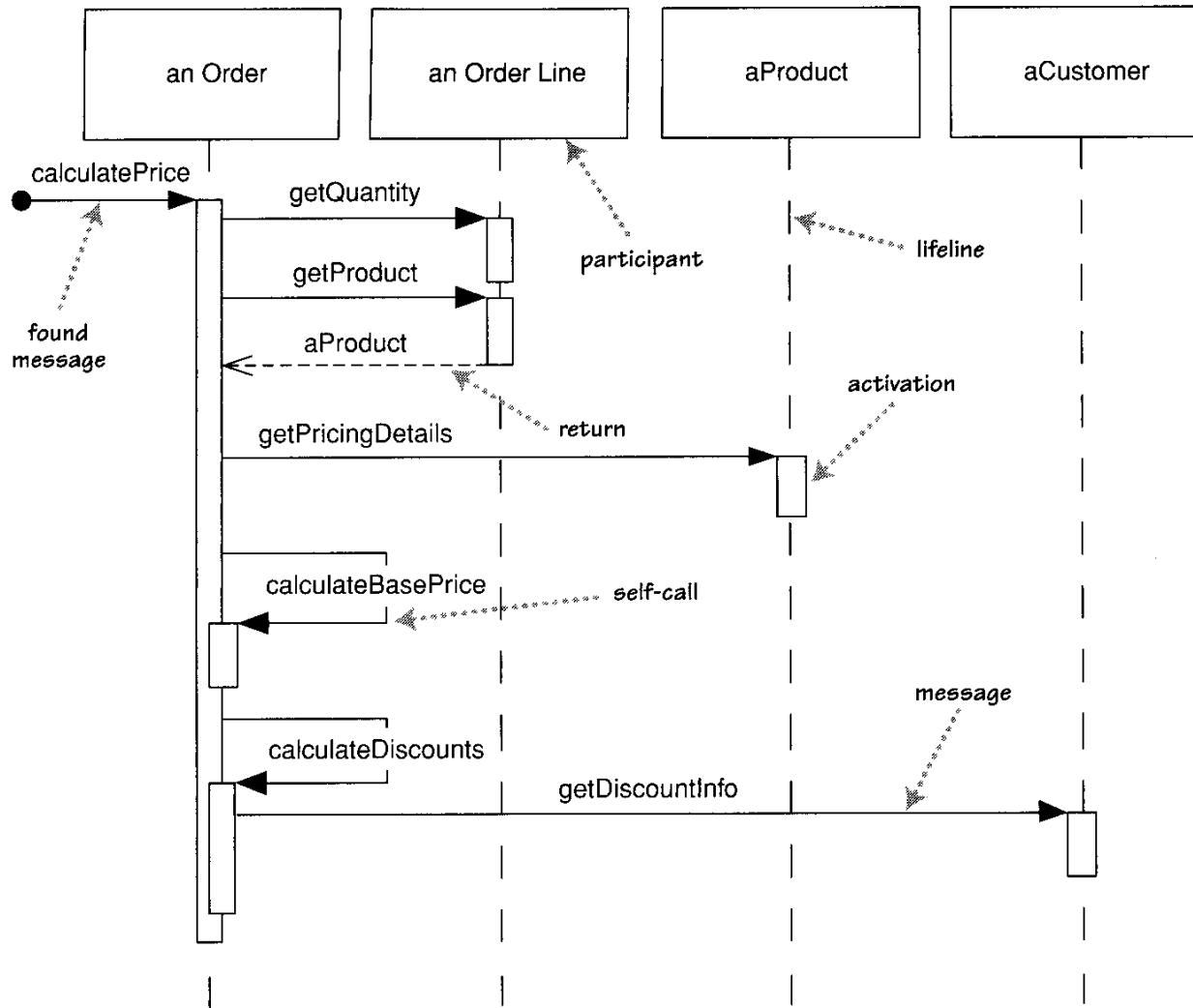


# SEQUENCE DIAGRAMS

- Rappresentano il tipo di diagramma di interazione largamente più utilizzato
- In generale, un sequence diagram modella le interazioni tra uno o più attori e il sistema software, nell'ambito dell'esecuzione di uno scenario di esecuzione
  - Le interazioni tra attori e sistema e tra le varie parti del sistema sono modellate in forma di *messaggi*, così come prevede il paradigma object-oriented



# SEQUENCE DIAGRAMS: ESEMPIO



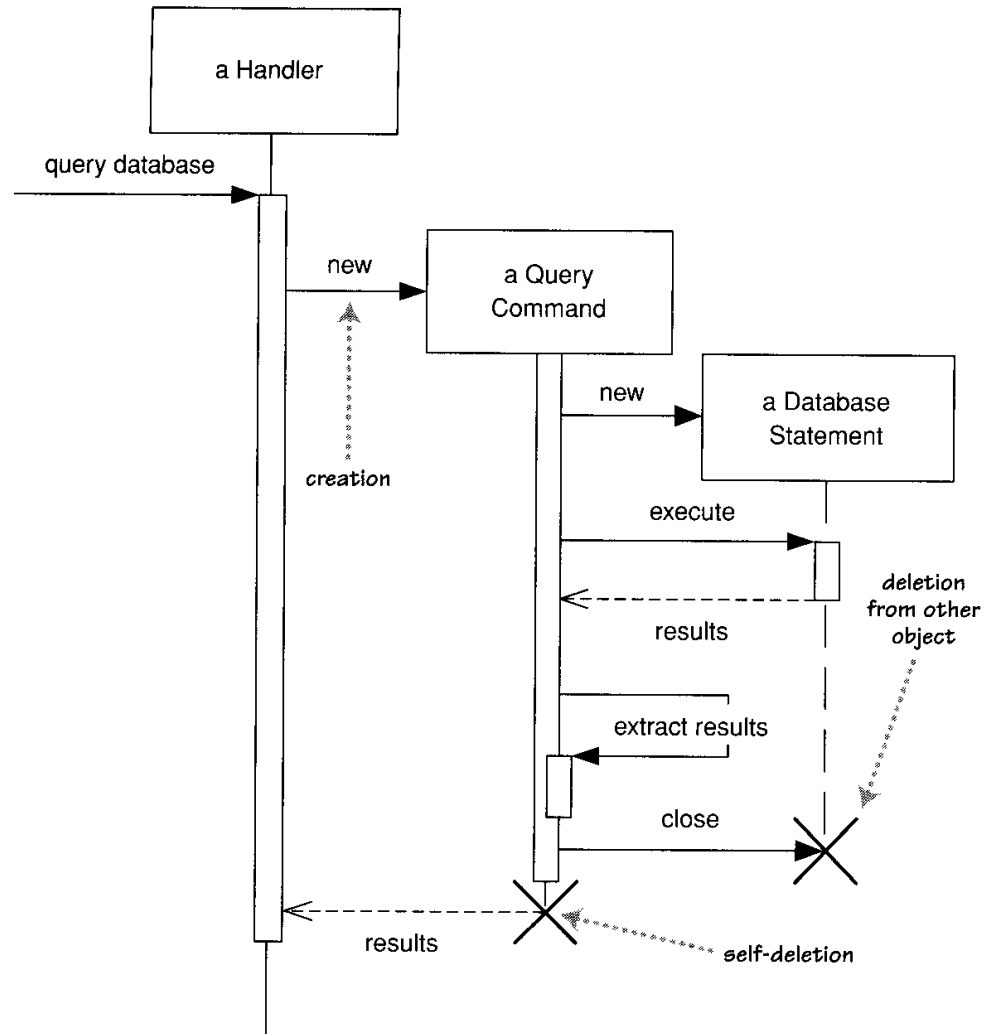


# SEQUENCE DIAGRAM: ELEMENTI

- Istanze di classi (oggetti)
  - Rappresentate da rettangoli col nome della classe e l'identificatore dell'oggetto sottolineati (notazione UML 1) oppure semplicemente con un nome dal quale si evinca che si sta considerando un'istanza della classe (ad esempio *anOrder* oppure *aProduct*)
- Attori o EndPoint
  - Sono riportati sulla sinistra, con frecce di interazione verso oggetti del sistema
  - Possono anche non essere riportati, nel caso in cui lo scenario venga avviato a sua volta da un altro scenario
  - Rappresentano la persona o il metodo che dà l'avvio all'elaborazione
- Messaggi
  - Rappresentati come frecce da un attore ad un oggetto, o fra due oggetti.
  - I messaggi di ritorno (se riportati) hanno una linea tratteggiata
  - I messaggi sincroni terminano con una freccia triangolare piena
  - I messaggi asincroni terminano con una freccia semplice →
  - Un messaggio può anche insistere all'interno di uno stesso oggetto: in tal caso è indicato da un autoanello
  - L'ordine dei messaggi (dall'alto verso il basso) ricalca l'ordine sequenziale con il quale vengono scambiati



# SEQUENCE DIAGRAMS

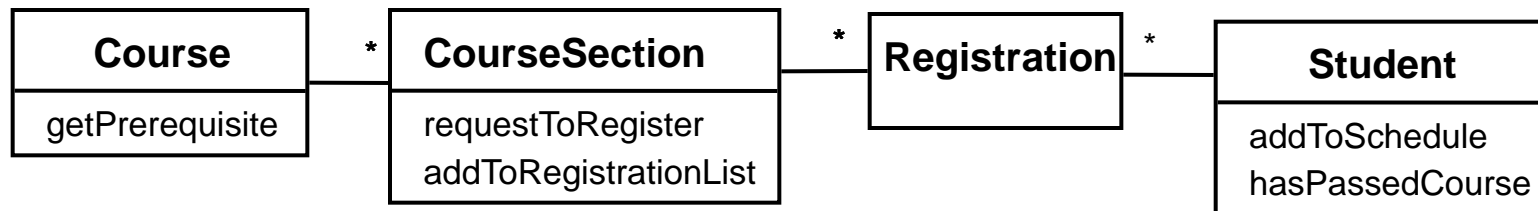
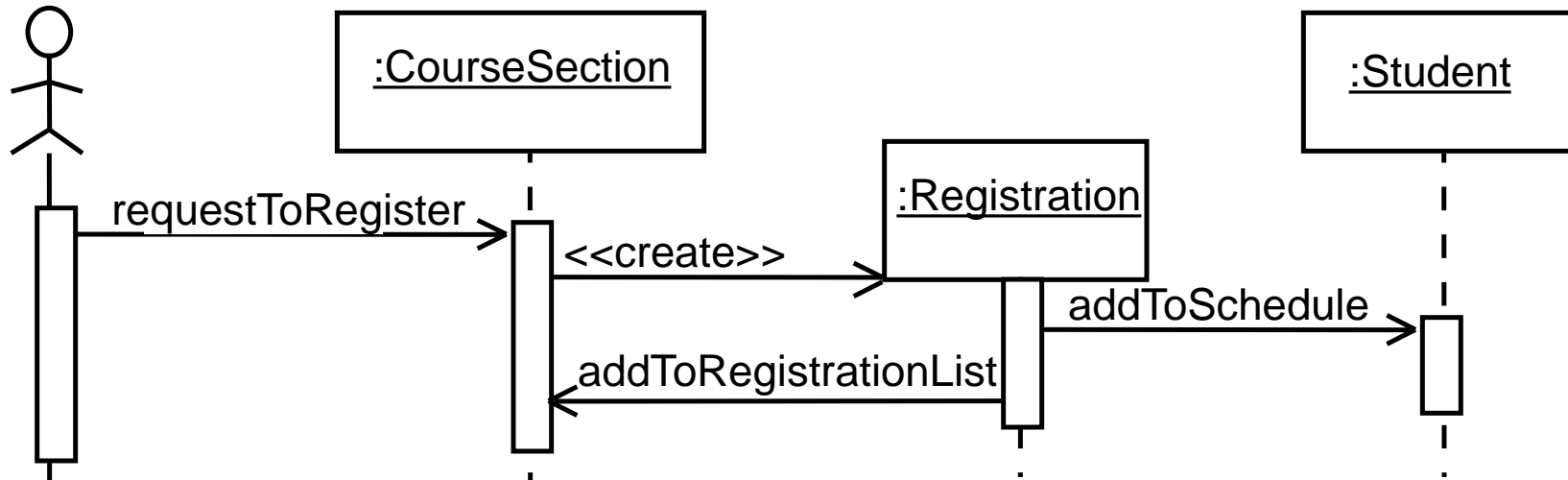


# LIFELINES E BARRE DI ATTIVAZIONE

- Lifelines (linee di vita)
  - Si tratta di linee tratteggiate verticali, che partono dal rettangolo rappresentativo dell'oggetto e giungono fino al fondo del diagramma. Indicano il periodo temporale di vita dell'oggetto, dalla sua costruzione alla sua distruzione
    - Le linee di vita di oggetti staticamente definiti (*static*) partono dalla cima del diagramma
    - Una richiesta al costruttore può creare un oggetto: in questo caso l'oggetto viene disegnato al termine della freccia corrispondente al messaggio di creazione
    - La distruzione di oggetti è indicata da una croce (ics) che interrompe la lifeline dell'oggetto
- Activation box (Barra di attivazione )
  - E' rappresentata da un rettangolo che copre una parte della lifeline di un oggetto cui giunge un messaggio
  - Rappresenta idealmente il periodo di tempo necessario per elaborare la richiesta giunta all'oggetto



# SEQUENCE DIAGRAM: UN ESEMPIO

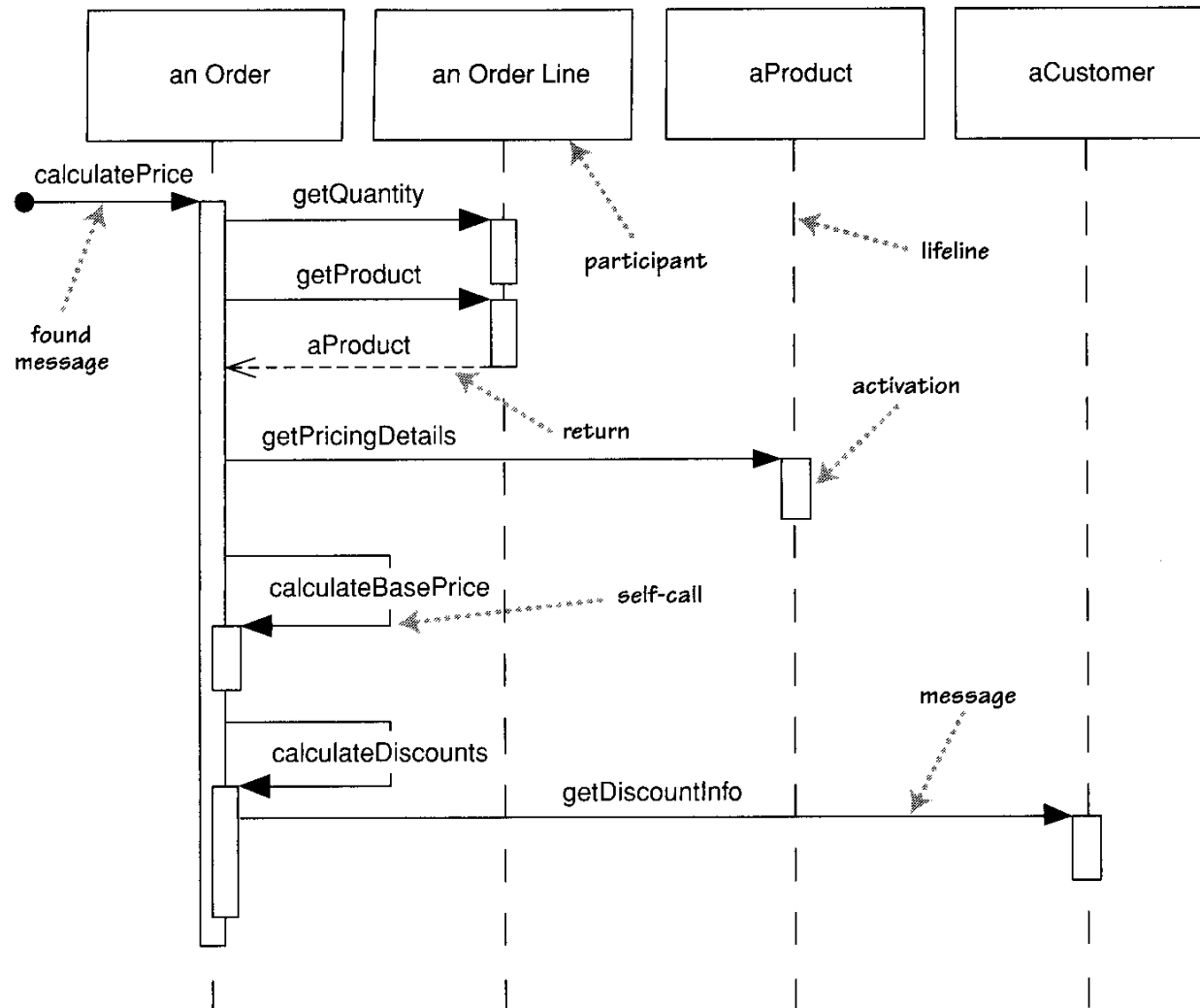


# SEQUENCE DIAGRAM: CAMPO DI APPLICAZIONE

- In fase di **progettazione di dettaglio** i sequence diagram descrivono realizzazioni di metodi
  - Come oggetti compaiono istanze delle classi di progetto di dettaglio
  - Come messaggi compaiono le chiamate di metodo sugli oggetti
    - Al posto degli elementi architetturali dovrebbero comparire oggetti delle classi che ne consentono l'accesso (ad esempio, anzichè *database* potrebbe comparire l'oggetto JDBC sul quale si effettuano le query)
    - Per la descrizione di algoritmi, i sequence diagram non sono lo strumento più adatto: ad essi verranno spesso preferiti activity e statechart diagrams



# SEQUENCE DIAGRAMS: ESEMPIO



# SEQUENCE DIAGRAM: CICLI E CONDIZIONI

- Cicli e condizioni si indicano con un riquadro (*frame*) che racchiude una sottosequenza di messaggi
  - Nell'angolo in alto è indicato il costrutto. Tra i costrutti possibili
    - **Loop** (ciclo while-do o do-while): la condizione è indicata tra parentesi quadra all'inizio o alla fine
    - **Alt** (if-then-else): la condizione si indica in cima; se ci sono anche dei rami *else* allora si usa una linea tratteggiata per separare la zona *then* dalla zona *else* indicando eventualmente un'altra condizione accanto alla parola *else*
    - **Opt** (if-then): racchiude una sottosequenza che viene eseguita solo se la condizione indicata in cima è verificata
      - Sono possibili anche altri costrutti per indicare parallelismo, regioni critiche, etc.
- In realtà, è buona norma utilizzare altri tipi di diagramma quando l'algoritmo da modellare si fa complesso

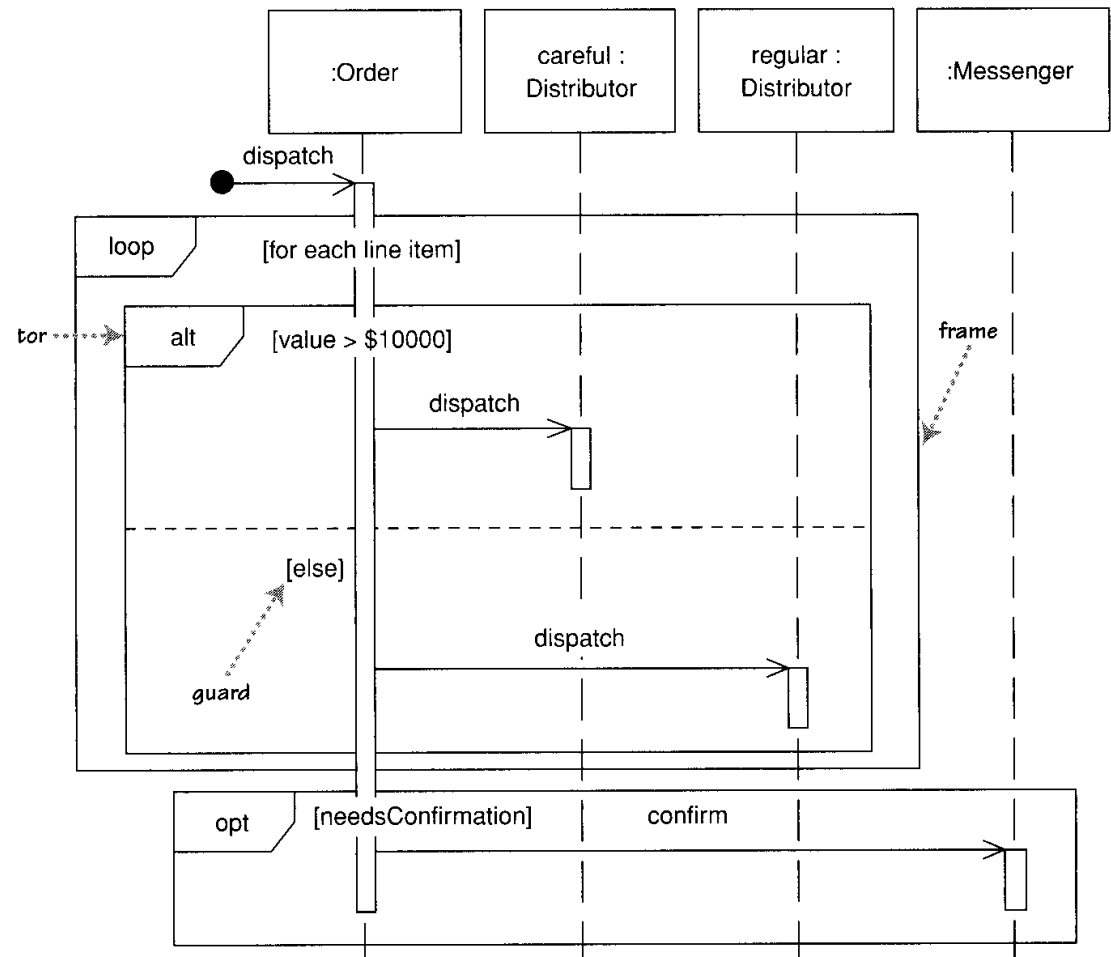


# SEQUENCE DIAGRAMS: ESEMPIO

```

procedure dispatch
  foreach (lineitem)
    if (product.value > $10K)
      careful.dispatch
    else
      regular.dispatch
    end if
  end for
  if (needsConfirmation) messenger.confirm
end procedure
  
```

Questa notazione é stata  
introdotta con UML  
versione 2





# ESERCIZIO : SEQUENCE DIAGRAM DAL CODICE

- In questo caso immaginiamo di voler disegnare il sequence diagram corrispondente ad una funzionalità che è già stata implementata
- Questo sequence diagram sarà inevitabilmente a livello di dettaglio, e in esso compariranno tutte le classi realmente nel codice, comprese quelle dell'interfaccia utente
- Nel seguito viene riportato il codice che fu scritto per la funzionalità **Aggiungi Nuova Società (Nome, Valore Azione)**



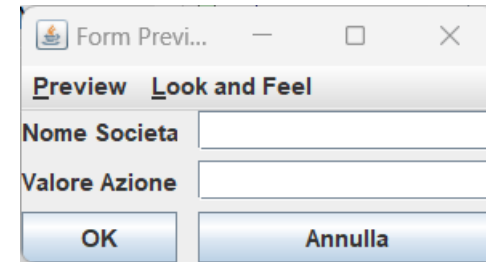
# AGGIUNGI NUOVA SOCIETA (NOME, VALORE AZIONE)

- Siccome per aggiungere una nuova società c'è bisogno di un doppio input, per questa volta scegliamo di creare una nuova form e la chiamiamo dalla Home

```
aggiungiNuovaSocietaButton.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        AggiungiNuovaSocieta aggiungiNuovaSocieta = new  
        AggiungiNuovaSocieta(controller,frame);  
        aggiungiNuovaSocieta.frame.setVisible(true);  
        frame.setVisible(false);  
    }  
});
```



# AGGIUNGI NUOVA SOCIETA (NOME, VALORE AZIONE) : BUTTON OK

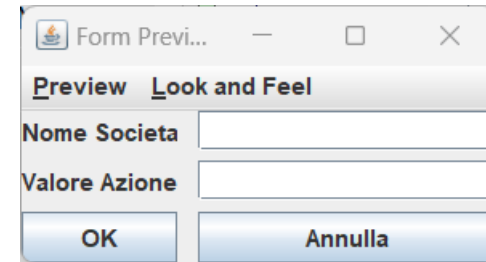


- Alla pressione del button OK bisogna:
  - Verificare che I dati in input siano validi (lo faremo in futuro)
  - Inserire I dati nel model tramite il controller
  - Tornare alla GUI precedente (annulla farà solo quest'ultima operazione)

```
OKButton.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        //aggiungere la società al listino  
        controller.aggiungiSocietaBorsa(nomeSocietaText.getText(), Double.parseDouble(valoreAzioneText.getText()));  
        frame.setVisible(false);  
        frameChiamante.setVisible(true);  
        frame.dispose();  
    }  
});  
}
```



# AGGIUNGI NUOVA SOCIETA (NOME, VALORE AZIONE) : CONTROLLER



Form Previ... — □ ×

Preview Look and Feel

Nome Societa

Valore Azione

OK Annulla

- Il codice nel controller è molto semplice:

```
public void aggiungiSocietaBorsa(String text, double.parseDouble) {  
    borsa.aggiungiSocieta(text,.parseDouble);  
}
```



# AGGIUNGI NUOVA SOCIETA (NOME, VALORE AZIONE) : AGGIUNGISOCIETA NEL MODEL

- L'oggetto borsa del model ha un attributo ArrayList di Societa, che inizialmente è vuoto
- Il metodo aggiungiSocieta istanzia un oggetto società e lo aggiunge all'ArrayList

```
public class Borsa {  
    private ArrayList<Societa> societa;  
  
    public Borsa() {  
        societa = new ArrayList<Societa>();  
    }  
    public void aggiungiSocieta(String text, double.parseDouble) {  
        Societa s = new Societa(text,.parseDouble);  
        societa.add(s);  
    }  
}
```



# AGGIUNGI NUOVA SOCIETA (NOME, VALORE AZIONE) : COSTRUTTORE SOCIETA

- Il costruttore per la Societa è semplicemente:

```
public class Societa {  
    private Double valoreAzione;  
    private String nome;  
  
    public Societa(String n, Double valore) {  
        nome=n;  
        valoreAzione = valore;  
    }  
}
```



sd SequenceDiagram1

