UML

Riferimenti

- Ian Sommerville, Ingegneria del Software, capitolo 8, 14
- Martin Fowler, UML Distilled, capitoli 3, 5
- Carlo Savy, Da C++ a UML, capitolo 34

Tipi di Modello 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.

- Tali diagrammi rappresentano i deliverables di diverse fasi del ciclo di vita del software, tra cui attività di analisi dei requisiti e attività di progettazione, sia di alto che di basso livello

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 il progetto** del sistema
  - Per **modellare l'implementazione** (object-oriented) di un sistema software

Class Diagram

- I concetti fondamentali di un class diagram sono estensioni dei concetti fondamentali del paradigma object-oriented

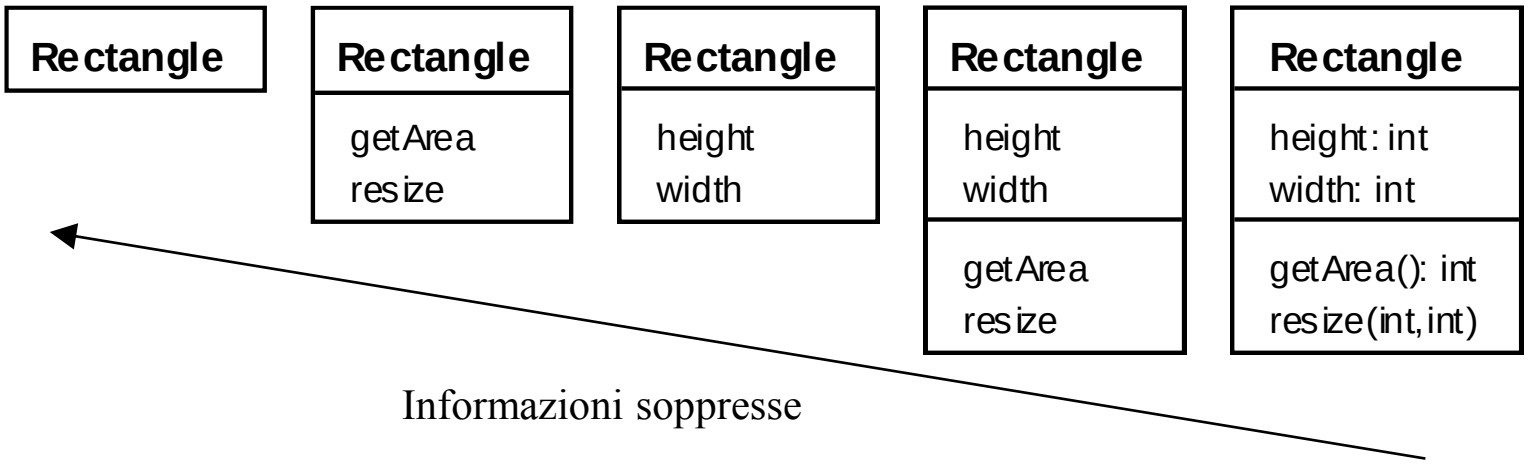
- Nel seguito verrà presentato il class diagram nelle sue varie accezioni, fino alla modellazione dell'implementazione di sistemi object-oriented

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.



- La signature completa di un'operazione è:

  operationName(parameterName: parameterType …): returnType

Attributi

visibilità nome molteplicità: tipo = default {proprietà}

- Sono consentiti tre livelli di visibilità:
  - **+ Livello pubblico**: L'utilizzo viene esteso a tutte le classi
  - **# Livello protetto**: L'utilizzo è consentito soltanto alle classi che derivano dalla classe originale
  - **- Livello 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 standard (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

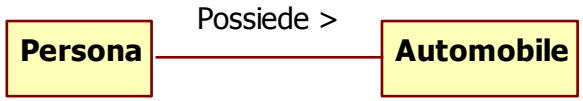
Esempio:

+ balanceOn (date: Date) : Money

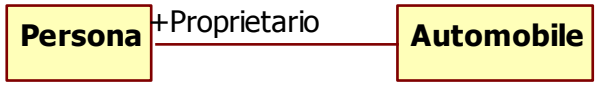
Associazioni

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

Esempio: **Persona *possiede* Automobile**

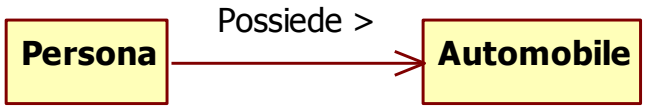


- Il verso dell'associazione indicato in figura indica in che direzione deve essere *letta* l'associazione
  - In questo caso indica che è la Persona a posseder l'Automobile e non l'Automobile a possedere la Persona!
- In alternativa, si può indicare il *ruolo* di uno dei due estremi dell'associazione



Verso di navigazione

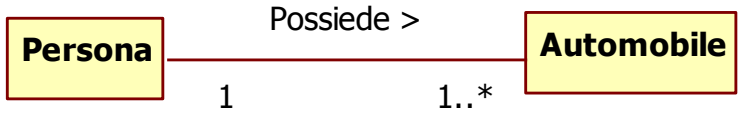
- Verso di navigazione di un'associazione
  - Il verso di navigazione è un'informazione utile soprattutto in fase di progetto di dettaglio
  - Indica in quale direzione è possibile reperire le informazioni



  - 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

Molteplicità delle associazioni

• La molteplicità delle associazioni indica quante istanze di una classe possono essere associate con una singola istanza di un'altra classe

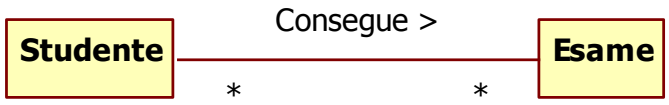


•Nell'esempio,

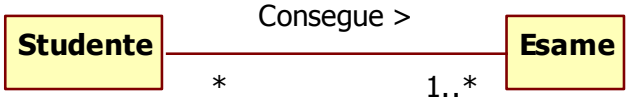
- una Persona possiede almeno una Automobile
  - evidentemente le persone che non possiedono Automobile non fanno parte del problema in oggetto
- Un'Automobile può essere posseduta da una e una sola Persona
  - Evidentemente, non è nel problema in oggetto il mantenimento di informazioni riguardo i proprietari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

Associazioni molti a molti

- **Associazioni molti-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



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



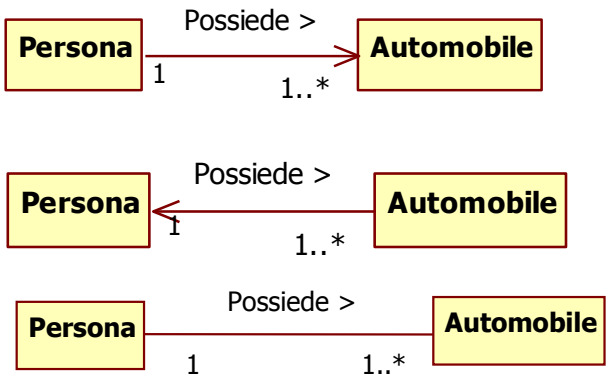
Associazioni uno a uno

- **Uno-a-uno**
  - Ogni studente ha uno e un sol 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

  - Se avessimo voluto considerare anche studenti privi di badge, allora la soluzione sarebbe stata:

Codifica delle associazioni

- **Molti a uno** (es.: una Persona, molte Automobili)
  - La classe Automobile ha un attributo Persona
  - oppure la classe Persona ha un attributo array di Automobile
  - o entrambe le cose



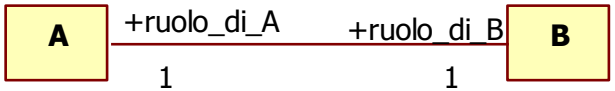
    - La differenza tra le tre soluzioni è identificata dai versi della frecce di navigazione
    - Si è parlato genericamente di array ma in realtà potrebbe trattarsi di una qualsiasi struttura vettoriale

Codifica delle associazioni

- Molti a molti (es: molti Studente, molti Esame)
  - La classe Studente è implementata avendo anche un attributo array di Esame,
  - Oppure la classe Esame è implementata avendo anche un attributo array di Studente
  - O entrambe le cose
- Uno a Uno (es: uno Studente, un Badge)
  - La classe Badge è implementata avendo un attributo Studente
  - Oppure la classe Studente è implementata avendo un attributo Badge
  - O entrambe le cose

Esempi Java

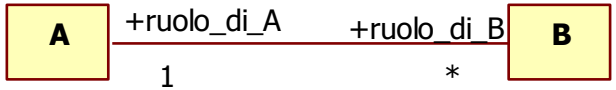
```
class A {
A(B b) {this.link(b)}
public link(B linkato)
        {this.ruolo_di_B =linkato}
private B ruolo_di_B;
};

class B {
B(A a) {this.link(a)}
public link(A linkato)
        {this.ruolo_di_A=linkato}
private A ruolo_di_A;
};

public static void main() {
A a;
B b;
a=new A(b);
b.link(a);
};
```



Associazione uno a uno

Associazione uno a molti

```
class A {
public aggiungi(B newobj);
public rimuovi(B oldobj);
private ArrayList<B> lista;
};

class B {
B(A a) {this.link(a)}
public link(A linkato)
        {this.ruolo_di_A=linkato}
private A ruolo_di_A;
};

public static void main(){
A a=new A();
a.aggiungi (new B(a));
}
```



Esempi C++

```
class A {
Public: link(B* linkato):ruolo_di_B(linkato) {}
Private: B* ruolo_di_B;
};

Class B {
Public: link(A* linkato):ruolo_di_A(linkato) {}
Private: A* ruolo_di_A;
};

Void main() {
A a;
B b;
a.link(&b);
b.link(&a);
};
```

Associazione uno a uno

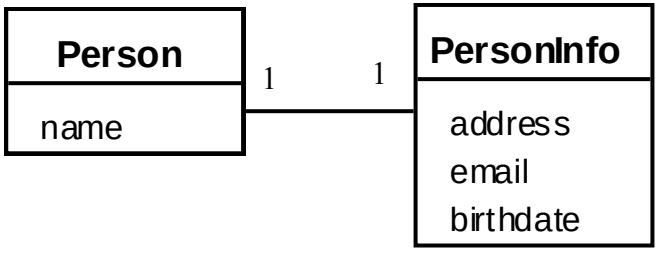
```
class A {
Public:     aggiungi(B* newobj);
            rimuovi(B* oldobj);
Private:    lista_puntatori_a_B;
};

Class B {
Public: link(A* linkato):ruolo_di_A(linkato) {}
Private: A* ruolo_di_A;
};
```

Associazione uno a molti

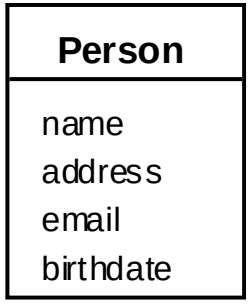
Contro-esempi

- A volte le associazioni *uno a uno* sono inutili

Evitare …

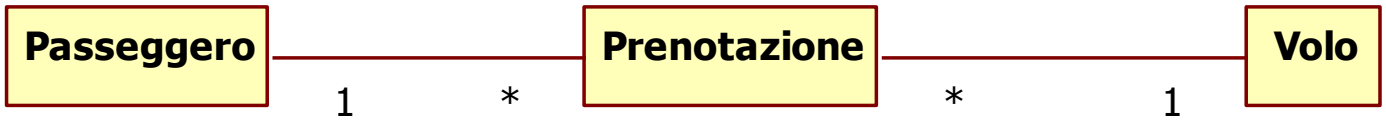


migliore soluzione!



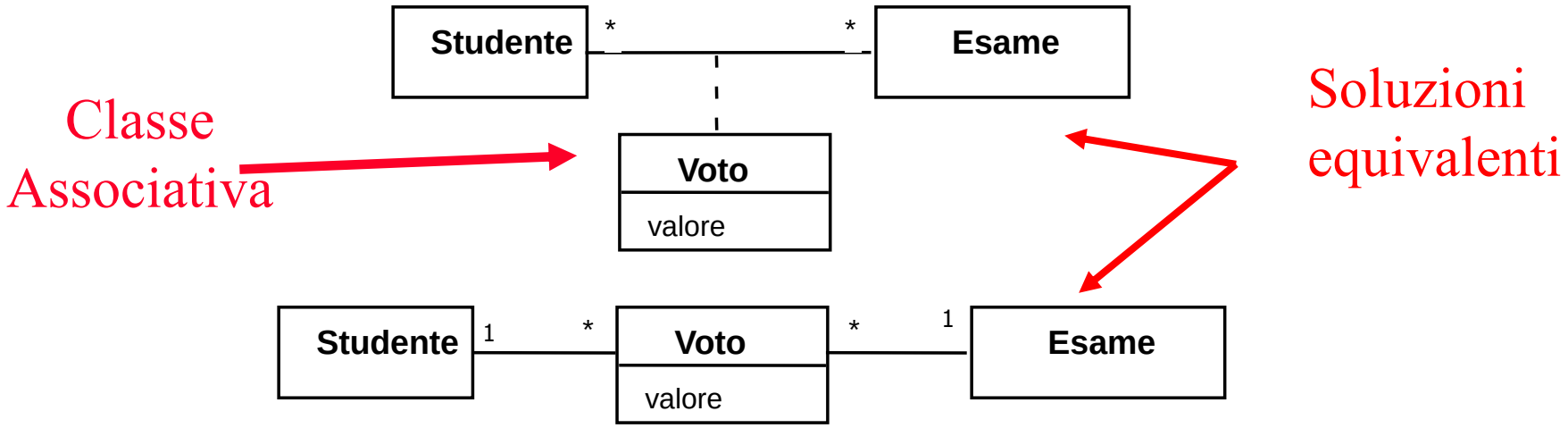
Un esempio più complesso

- 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 molti-a-molti

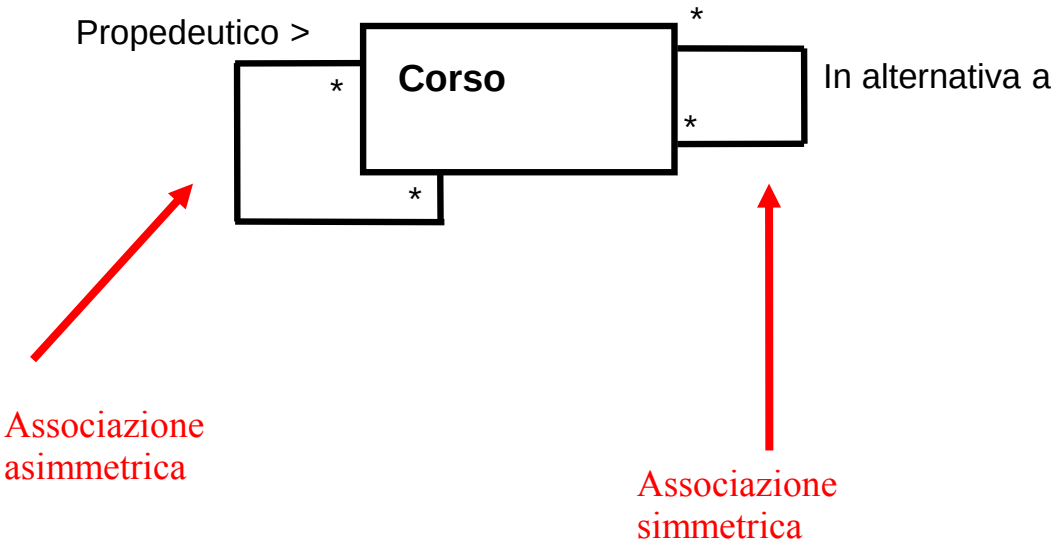


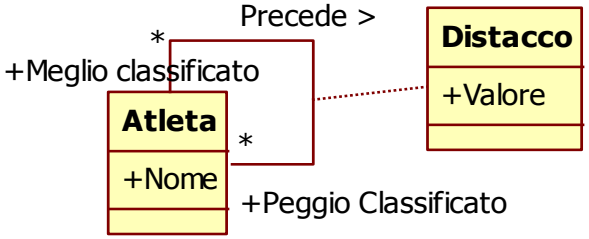
- Prima di creare un'istanza della classe Voto, devono esistere le istanze delle classi collegate

La classe associativa può avere attributi, metodi, altre associazioni

Associazioni riflessive

- Associazioni che collegano una classe con se' stessa

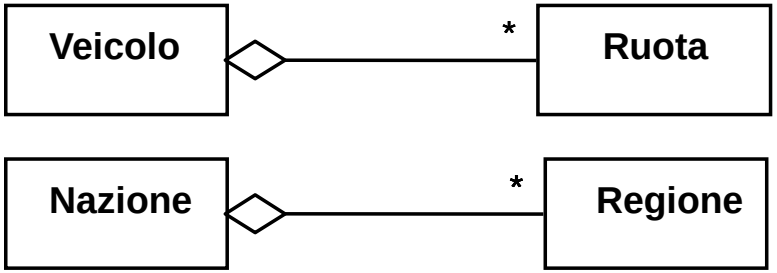




Permette, per ogni atleta, di risalire al distacco da tutti gli altri atleti

Aggregazione

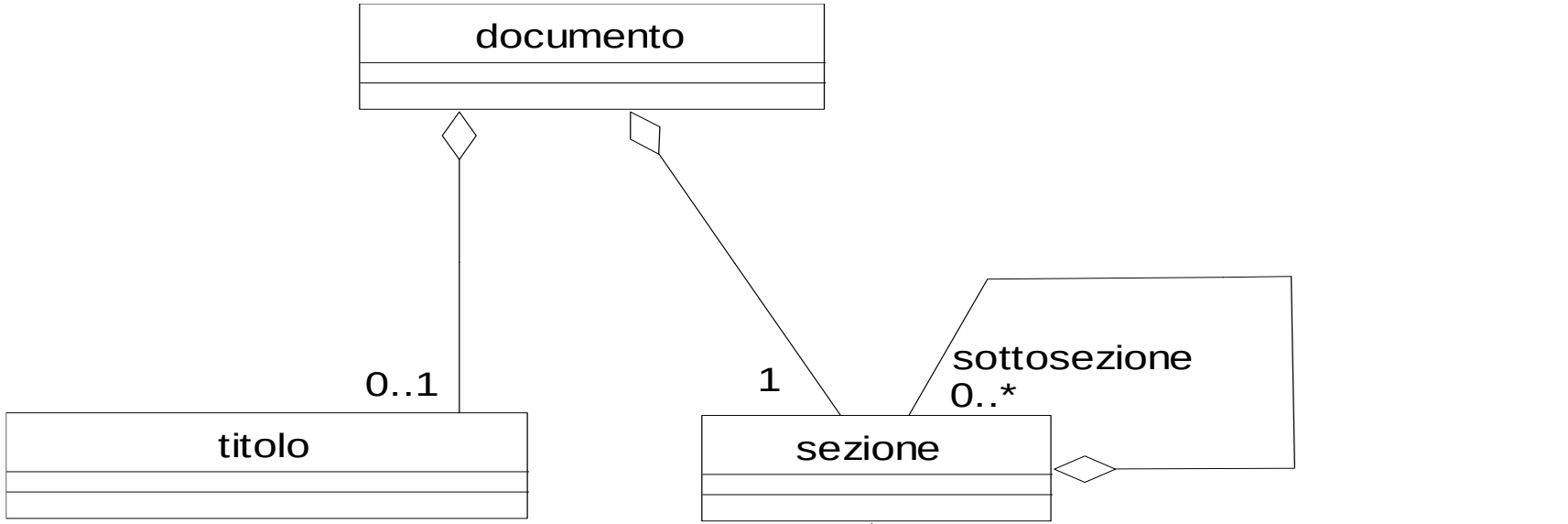
- Le Aggregazioni sono speciali associazioni che rappresentano una relazione ‘tutto-parti’.
  - Il lato del ‘tutto’ è spesso chiamato *l’aggregato*
  - La molteplicità dal lato del tutto, quando è sottintesa, vale 0..1



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

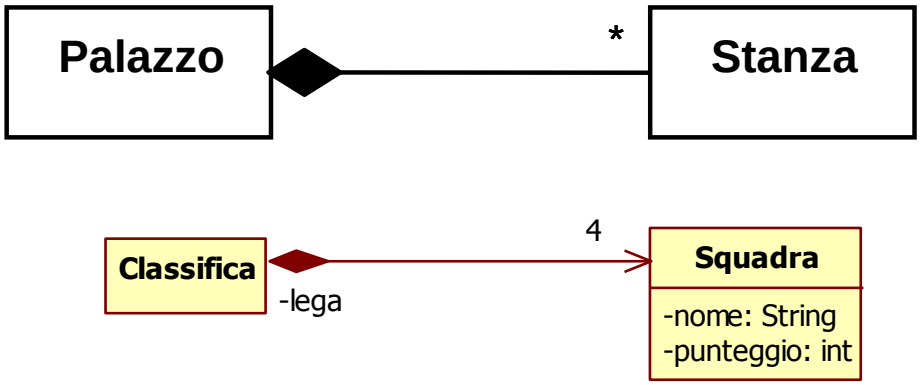
Una gerarchia di aggregazione



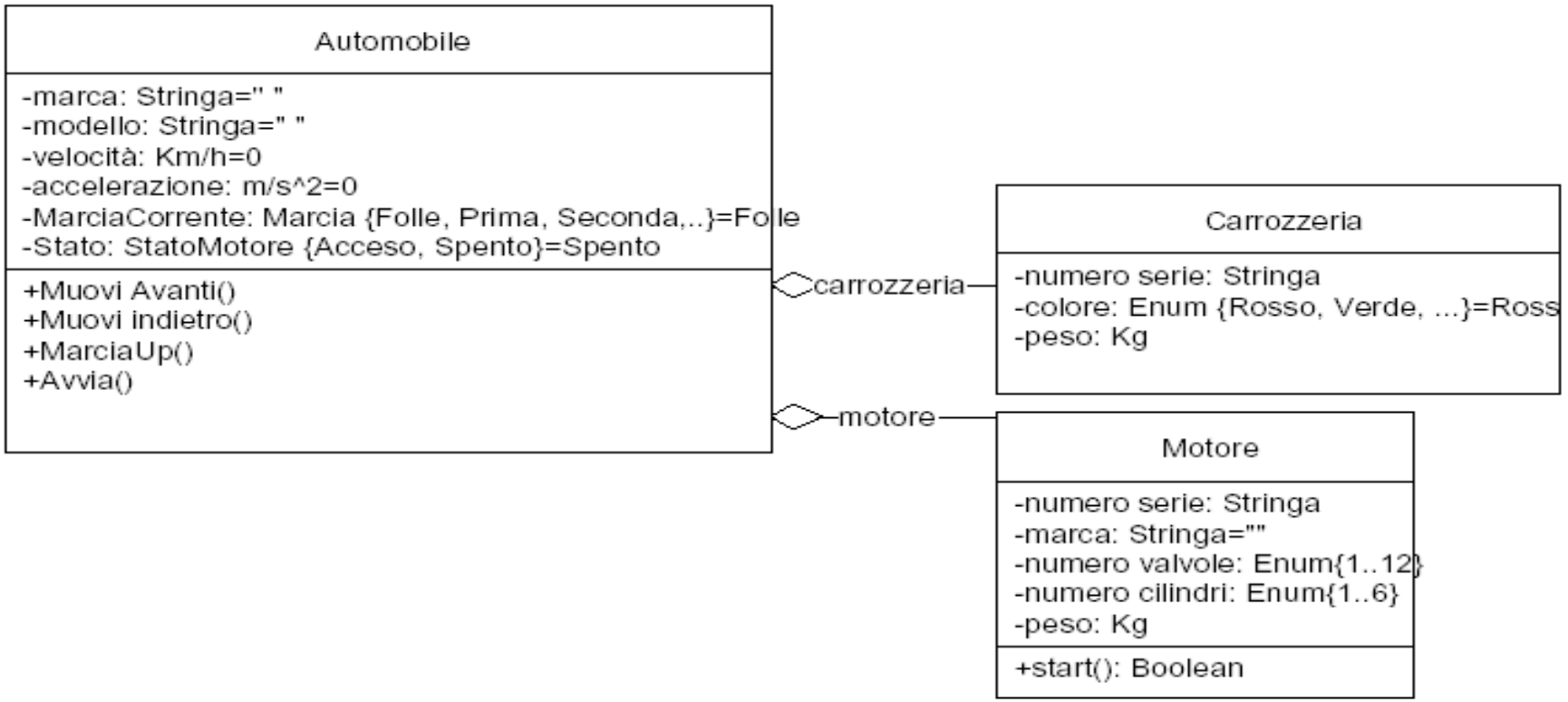
Esercizio proposto:
modellare i concetti di file e directory nell'ambito di un file system

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)
    - Evidentemente, in questo dominio, non ha senso parlare di stanze fintantochè esse non siano state legate alla casa in cui si trovano
    - La cardinalità dell'aggregazione, se sottintesa, vale 1



Esempio di aggregazione



Esempio di aggregazione Java (uno a uno)

```
class Automobile {
public:
//il costruttore inizializza Automobile anche in assenza di Motore e Carrozzeria
Automobile(Stringa marca, Stringa mod,…, Motore mot, Carrozzeria car)
{this.marca=marca; this.modello= mod;
 this.motore=mot; this.carrozzeria=car;}
…
private:
Carrozzeria carrozzeria;        //traduzione aggregazione
Motore motore;                  //traduzione aggregazione
…
};
```

Esempio di aggregazione Java (uno a uno)

Il programma principale dovrà dapprima creare un oggetto di tipo Carrozzeria e Motore ed infine l'oggetto di tipo Automobile. Ad esempio:

```
public static void main (String args[]) {
//definisce e inizializza oggetto di tipo Motore
Motore m=new Motore("AZ123","Ferrari", 6,12,1000);
//definisce e inizializza oggetto di tipo Carrozzeria
Carrozzeria c=new Carrozzeria("12345ASA", "RossoFerrari",1500);
//definisce e inizializza oggetto Automobile, fornendo puntatori
Automobile auto1=new Automobile("Ferrari", “2012”, c,m);
…
}
```

Esempio di aggregazione C++ (uno a uno)

```
//file Automobile.h
#include "Motore.h"
#include "Carrozzeria.h"
class Automobile {
public:
//il costruttore inizializza Automobile anche in assenza di Motore e Carrozzeria
Automobile(Stringa marca= "", Stringa mod="",…, Motore* mot=0, Carrozzeria* car=0);
…
private:
Carrozzeria* carrozzeria;          //traduzione aggregazione
Motore* motore;                    //traduzione aggregazione
…
};

//file Automobile.cpp
//Costruttore di Automobile
Automobile:: Automobile(Stringa marca, Stringa mod,…,Motore* mot, Carrozzeria* car) :
marca(marca),  modello(mod),motore(mot),  carrozzeria(car);
```

Esempio di aggregazione C++ (uno a uno)

Il programma principale dovrà dapprima creare un oggetto di tipo Carrozzeria e Motore ed infine l'oggetto di tipo Automobile. Ad esempio:

```
#include "Automobile.h"
main () {                    //usa classe Automobile
//definisce e inizializza oggetto di tipo Motore
Motore m("AZ123","Ferrari", 6,12,1000);
//definisce e inizializza oggetto di tipo Carrozzeria
Carrozzeria c("12345ASA", "RossoFerrari",1500);
//definisce e inizializza puntatore a motore
Motore* puntatoreMotore=&m;
//definisce e inizializza puntatore a carrozzeria
Carrozzeria* puntatoreCarrozzeria=&c;
//definisce e inizializza oggetto Automobile, fornendo puntatori
Automobile auto1("Ferrari", …, puntatoreCarrozzeria, puntatoreMotore);
…
}    //al termine vengono separatamente distrutti m,c e auto1
```

Esempio di composizione Java (uno a uno)

```
class Contenitore {
public:
Contenitore(String a, String x)
{nomeContenitore=a;
contenuto=new Contenuto(x);}
// composizione
Contenuto contenuto;
String nomeContenitore;
};
```

Affinchè si possa parlare di contenimento stretto (composizione) deve essere garantito che la classe Contenitore sia effettivamente l'unica ad accedere al metodo costruttore della classe Contenuto

```
class Contenuto {
public:
Contenuto(String x) {nomeContenuto=x};
String nomeContenuto;
};
```

Esempio di composizione C++ (uno a uno)

```
//file Contenitore.h
#include "Contenuto.h"
class Contenitore {
public:
Contenitore(String, String);
~Contenitore();

// composizione
Contenuto c; //Nota bene: è un oggetto, non un puntatore!
String nomeContenitore;
};
```

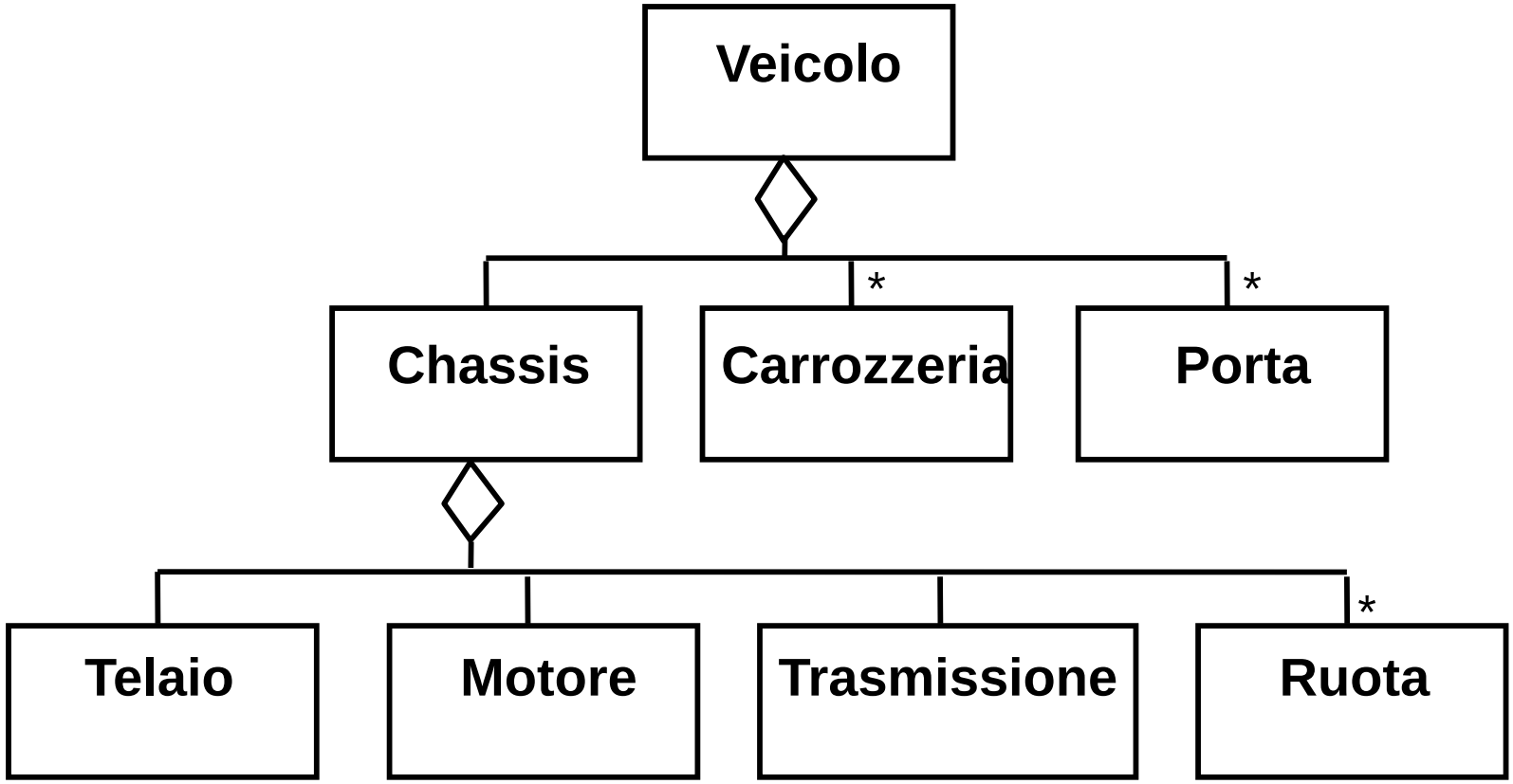
```
//file Contenuto.h
class Contenuto {
public:
//il costruttore inizializza gli attributi propri e richiama il costruttore dell'elemento contenuto
Contenuto(String x);
String nomeContenuto;
};
```

```
//file Contenuto.cpp
#include "Contenuto.h"
Contenuto::Contenuto(String x):nomeContenuto(x){}
```

```
//file Contenitore.cpp
#include "Contenitore.h"

//Costruttore: chiama anche il costruttore del contenuto
Contenitore:: Contenitore (String a, String x):nomeContenitore(a),c(x){}

//Distruttore: chiama anche il distruttore del contenuto
Contenitore::~Contenitore() { delete &c;}
```

Affinchè si possa parlare di contenimento stretto (composizione) deve essere garantito che la classe Contenitore sia effettivamente l'unica ad accedere al metodo costruttore della classe Contenuto

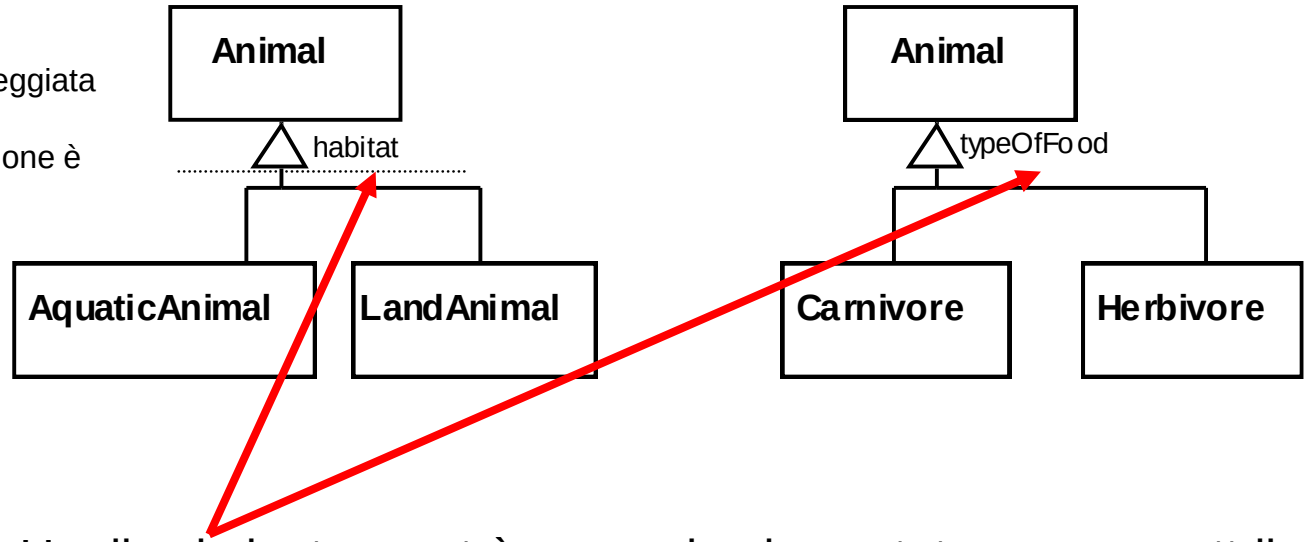
Gerarchia di Aggregazione



Generalizzazioni

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

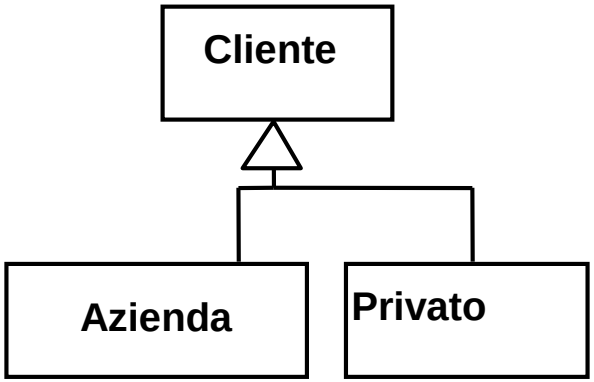
La linea tratteggiata indica che la generalizzazione è incompleta



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

Generalizzazioni

• A livello concettuale, una gerarchia di generalizzazione esprime una relazione ls-a tra un concetto generale e le sue specializzazioni



- **Regola della Sostituibilità**: Ogni elemento della classe Azienda (o Privato) è anche un elemento della classe Cliente.
- Ne consegue che il software potrà riferirsi al tipo Cliente, senza dover distinguere se esso è un'Azienda o un Privato.

Implementazione della generalizzazione

- A livello di progetto di dettaglio, la generalizzazione può essere implementata :
  - Con una **relazione di ereditarietà** tra due classi concrete
    - Le classi derivate (figlie) ereditano attributi e metodi public e protected dalla classe padre
    - Si può ereditare anche da una *classe astratta*: la classe figlia implementa i metodi sfruttando l'overriding
    - **Problemi** quando la classe padre include troppe proprietà non desiderate nelle classi figlie!
- In alternativa, è possibile usare una **relazione di implementazione** (o realizzazione) tra una classe concreta ed una *interfaccia*

Classe astratta ed Interfacce

- La classe astratta è una classe che ha una o più operazioni astratte e non può essere direttamente istanziata.
  - Occorre prima creare una sottoclasse concreta
  - In UML si indica col nome in corsivo o etichetta {abstract}

- Una interfaccia è una classe priva di implementazione (tutte le sue proprietà sono astratte)
  - descrive una porzione del comportamento visibile di un insieme di oggetti
  - In UML si indica con la parola chiave <<interface>>