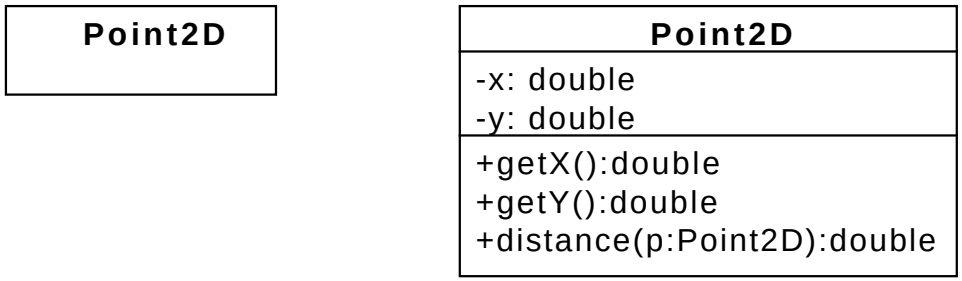
Diagrammi delle classi
Class Diagram
Concetti Fondamentali

a cura di **Angelo Furfaro**
da “UML Distilled”
Martin Fowler

Dipartimento di
Ingegneria Informatica, Elettronica, Modellistica e Sistemistica
Università della Calabria, 87036 Rende(CS) - Italy
Email: a.furfaro@unical.it
Web: http://angelo.furfaro.dimes.unical.it

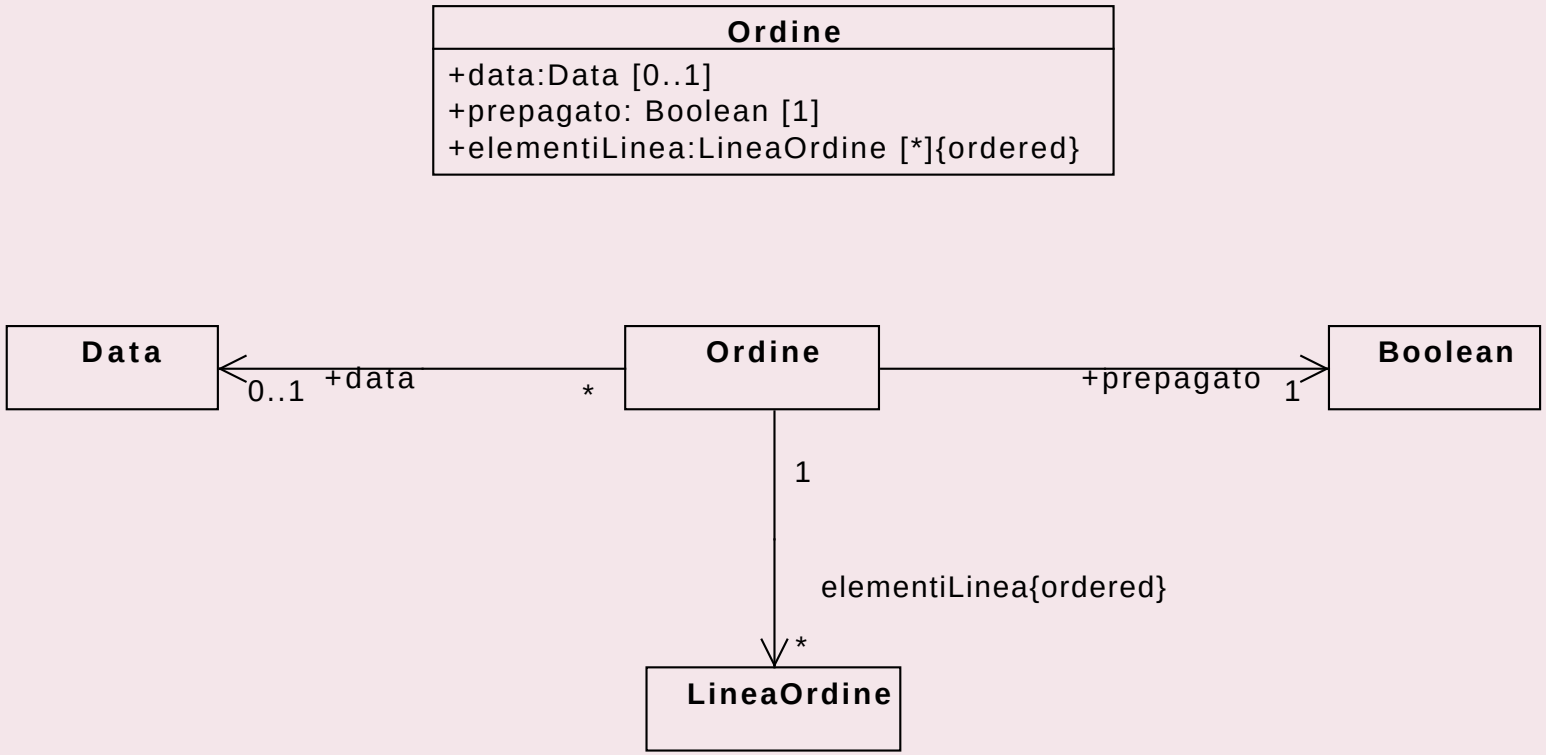
Introduzione

- Il diagramma delle classi è il diagramma UML più ampiamente utilizzato
- Si presta a rappresentare il maggior numero di concetti
- Descrive il **tipo degli oggetti** che fanno parte del sistema e le varie tipologie di **relazioni statiche** tra di essi
- Mostra le **proprietà** e le **operazioni** (metodi) delle classi ed i **vincoli** tra gli oggetti istanze delle classi
- UML usa il termine **caratteristica** (*feature*) per indicare sia le proprietà che le operazioni di una classe
- Le classi sono graficamente rappresentate come dei rettangoli (**box**) contenenti all'interno (almeno) il nome della classe
- I box sono solitamente divisi in tre compartimenti: nome della classe, attributi, operazioni



Proprietà

- Le proprietà rappresentano le caratteristiche strutturali di una classe.
- In prima approssimazione si può affermare che le proprietà corrispondano ai campi di una classe
- Sono rappresentate mediante due possibili notazioni: **attributi** e **associazioni**
- Nell'esempio le stesse proprietà sono espresse in entrambe le notazioni



Attributi

Un attributo descrive una proprietà come una stringa all'interno del relativo compartimento secondo il seguente formato:

`visibilità nome: tipo molteplicità = default {stringa-di-proprietà}`

Esempio:

`- titolo: String = ''Hello world'' {read-only}`

- La `visibilità` specifica se l'attributo è pubblico (+), privato (-), protetto (#) o ha visibilità di package (~)
- Il `nome` di un attributo corrisponde al nome di un campo
- Il `tipo` è un vincolo sugli oggetti che possono corrispondere all'attributo (il tipo del campo nel linguaggio di programmazione)
- Il `default` è il valore assunto dall'attributo se non diversamente specificato
- La `{stringa-di-proprietà}` indica caratteristiche aggiuntive di un attributo (ad esempio vincoli). Nell'esempio {read-only} indica un attributo il cui valore non è modificabile dai client.
- La `molteplicità` esprime un vincolo sul numero di oggetti che possono essere *collegati* all'attributo (si veda in seguito)

Associazioni

- L'altra notazione per rappresentare una proprietà è in forma di associazione
- Un'associazione che modella una proprietà è raffigurata come una linea continua che collega due classi, orientata dalla classe *sorgente* a quella *destinazione*
- Il nome della proprietà e la molteplicità sono indicate vicino all'estremità dell'associazione relativa alla destinazione
- La classe destinazione corrisponde al tipo della proprietà
- Di solito si utilizzano gli attributi per le cose "piccole" (oggetti valore)
- Le associazioni sono usate per le proprietà le cui classi sono più "significative"

Molteplicità

La molteplicità di una proprietà indica quanti oggetti possono entrare a farvi parte. Le più comuni sono:

- **1** (Un ordine deve essere fatto da un solo cliente.)
- **0..1** (Un'azienda cliente può avere un rappresentante o meno.)
- * (da zero a molti: un cliente può anche non fare un ordine, ma non c'è limite superiore)
- Si possono specificare degli intervalli (ad esempio **2..4**) fornendo esplicitamente i limiti inferiore e superiore
- Se i due estremi coincidono si usa un solo numero (**1** equivale a **1..1**)
- * è un'abbreviazione per **0..***

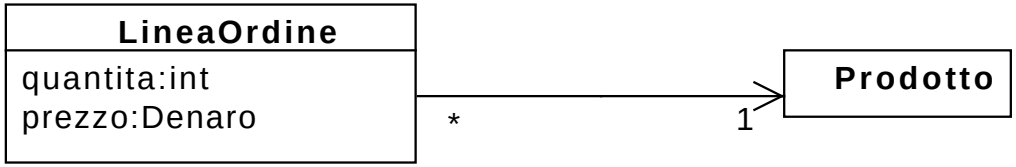
Terminologia attributi

- **Opzionale**: implica **0** come limite inferiore.
- **Obbligatorio**: implica un limite inferiore di **1** (almeno uno)
- **Ad un sol valore**: implica un limite superiore di **1** (al più uno)
- **A più valori**: implica un limite superiore maggiore di **1** (di solito *)

Molteplicità

- Di default gli elementi coinvolti in una molteplicità a più valori formano un set: non è definito un ordinamento tra di essi (è sottointesa la proprietà {`unordered`})
- Per specificare il vincolo che gli elementi devono essere ordinati si aggiunge l'indicazione {`ordered`}
- UML 1 consentiva l'uso di molteplicità discontinue come **2,4**
- La molteplicità di default di un attributo è **[1]**

Interpretazione delle proprietà nella programmazione



- Come per molti aspetti di UML, non esiste un'unica interpretazione per le proprietà a livello di codice
- Tipicamente si tende ad introdurre un campo di una classe o una proprietà se il linguaggio le supporta (C# prevede le proprietà , `Java` no)
- La classe **LineaOrdine** in Java potrebbe apparire come segue:

```
public class LineaOrdine ...{
   private int  Quantita;
   private Denaro prezzo;
   private Prodotto prodotto;
 ...
}
```

- Un attributo corrisponde a campi privati se il linguaggio non supporta le proprietà esplicite, altrimenti corrisponde a proprietà pubbliche

Interpretazione delle proprietà nella programmazione

- Nei linguaggi senza proprietà i campi sono resi visibili all'esterno per mezzo di metodi accessori (`get`) e/o mutatori (`set`)
- Un attributo di sola lettura non avrà un corrispondente metodo `set`
- Usare i campi privati è un'interpretazione legata all'implementazione
- Sarebbe meglio porre l'enfasi più sui metodi di accesso (che fanno parte dell'interfaccia) che sui dati sottostanti
- Secondo tale approccio la classe **LineaOrdine** in Java può essere quella riportata di seguito
- Non c'è un campo dati per il prezzo che viene calcolato in tempo reale

```
public class LineaOrdine ...{
  private int quantita;
  private Prodotto prodotto;

  public int getQuantita(){return quantita;}
  public void setQuantita(int quantita){this.quantita=quantita;}
  public Denaro getPrezzo(){return prodotto.getPrezzo().multiply(quantita);}
 ...
}
```

Interpretazione delle proprietà nella programmazione

Proprietà a più valori

- Se un attributo ha più valori i dati corrispondenti formano una collezione
- La classe `Ordine` fa riferimento ad una collezione di oggetti di tipo `LineaOrdine`
- Poiché la corrispondente associazione è marcata come ordinata (nel senso che ogni elemento è associato ad una posizione) lo stesso deve valere per la collezione che la modella (ad esempio una `List` in Java)
- Una collezione non ordinata dovrebbe essere rappresentata con un insieme (di solito i programmatori usano una lista anche in questo caso)
- Le proprietà a più valori hanno un'interfaccia diversa rispetto a quelle a singolo valore

```
public class Ordine {
  private Set<LineaOrdine> linee=new HashSet<LineaOrdine>();
   ...
  public Set<LineaOrdine> getElementiLinea(){ return Collections.unmodifiableSet(linee); }
  public void aggiungLinea(LineaOrdine lo){ linee.add(lo); }
  public void eliminaLinea(LineaOrdine lo){ linee.remove(lo); }
   ...
}
```

Interpretazione delle proprietà nella programmazione

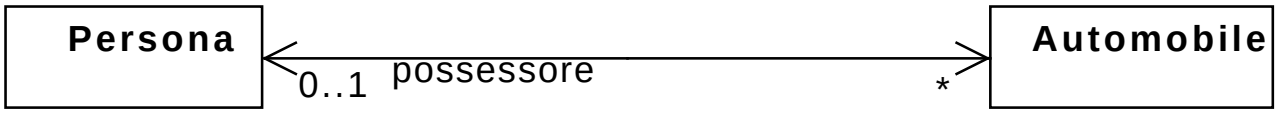
Proprietà a più valori

- Come nell'esempio, le proprietà multiple si modificano aggiungendo o rimuovendo singoli elementi per mezzo degli opportuni metodi
- La collezione che implementa una proprietà multipla non deve mai essere esposta direttamente all'esterno
- Nell'esempio si utilizza un *Proxy* che ha un'interfaccia di sola lettura
- Un'altra possibilità consiste nel restituire una copia della collezione

Conclusioni

- Non c'è una corrispondenza fissa tra UML e codice
- Una proprietà è un qualcosa che l'oggetto può mettere a disposizione indipendentemente se implementata come campo o come valore calcolato in seguito all'invocazione di un metodo
- Le proprietà non devono essere utilizzate per modellare relazioni temporanee (ad esempio, un parametro passato all'invocazione di un metodo ed utilizzato solo durante l'esecuzione dello stesso)

Associazioni bidirezionali



- Un'associazione bidirezionale è costituita da una coppia di proprietà collegate
- Nell'esempio la classe `Automobile` ha proprietà `proprietario:Persona`, mentre `Persona` ha proprietà `automobili:Automobile[*]`
- Il doppio collegamento indica che se si segue il valore di una proprietà e poi il valore di quella collegata si deve tornare ad un insieme che contiene l'elemento di partenza
- Implementare le associazioni bidirezionali è difficoltoso perché occorre assicurarsi che le proprietà coinvolte siano *sincronizzate*
- La cosa più importante è fare in modo che l'associazione sia controllata da una sola delle due parti (possibilmente quella a valore singolo)
- La classe subordinata deve violare temporaneamente l'incapsulamento dei suoi dati per farli arrivare a quella che controlla l'associazione

Associazioni bidirezionali

Esempio

```
class Automobile ...{
    ...
   private Persona proprietario;
    ...
   public Persona getProprietario(){ return proprietario ; }
   public void setProprietario(Persona p){
        if ( proprietario !=null ) proprietario .rimuoviAutomobile(this);
        proprietario =p;
        if ( proprietario !=null ) proprietario .aggiungiAutomobile(this);
   }
    ...
}

class Persona ...{
     ...
    private List<Automobile> auto;
     ...
    public Collection<Automobile> getAutomobili(){return Collections.unmodifiableCollection(auto);}
    void rimuoviAutomobile(Automobile a){ auto.remove(a); } //visibile solo a Automobile
    void aggiungiAutomobile(Automobile a){ auto.add(a); } //visibile solo a Automobile
     ...
}
```

Operazioni

Sintassi

- Le operazioni sono le azioni che la classe esegue (di solito corrispondono ai metodi della classe)
- La sintassi UML completa per le operazioni è la seguente:
  `visibilità nome (lista-parametri): tipo-di-ritorno {stringa-di-proprietà}`
- La visibilità segue le stesse regole sintattiche delle proprietà
- il `nome` è una stringa che indica il nome dell'operazione
- `lista-parametri` è la lista ordinata dei parametri dell'operazione separati da virgole
- `tipo-di-ritorno` specifica il tipo del valore restituito dall'operazione, se esiste
- `stringa-di-proprietà` indica caratteristiche aggiuntive dell'operazione (ad esempio vincoli).

Operazioni

Sintassi della lista dei parametri

- Gli elementi della lista dei parametri seguono una sintassi simile a quella degli attributi:

  `direzione nome: tipo= default`

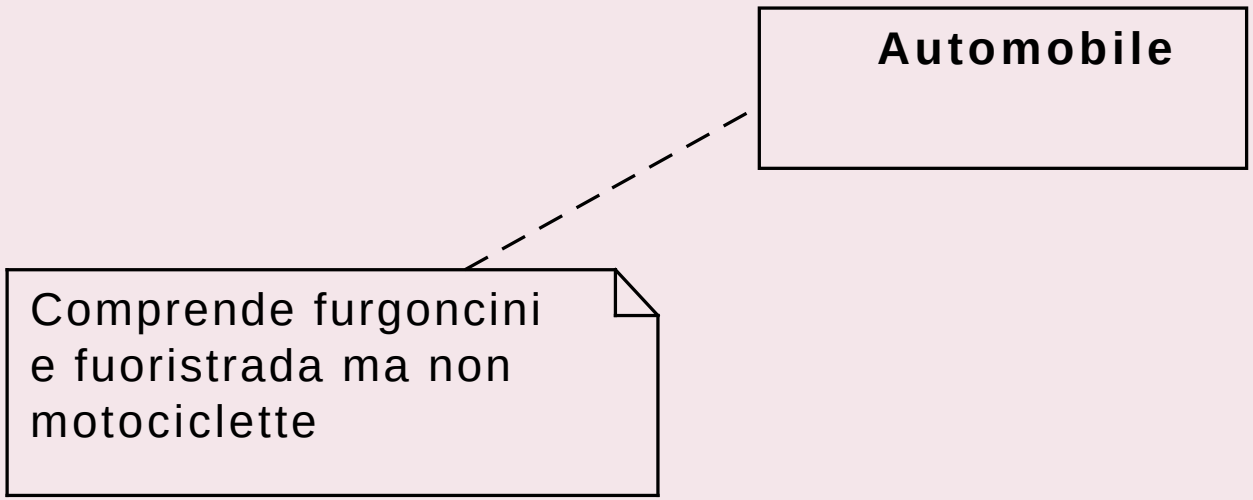
- `nome`, `tipo` e `default` sono gli stessi degli attributi
- La `direzione` indica se il parametro è di input (`in`), output (`out`) o entrambi (`inout`). Se omessa la direzione predefinita è `in`.
- Spesso è utile distinguere tra le operazioni che cambiano lo stato di un sistema e quelle che non lo fanno
- UML definisce **query** quelle che ottengono un valore senza modificare lo stato. Tali operazioni sono etichettate con la stringa di proprietà `{query}`
- UML distingue tra `operazione` e `metodo`: la prima corrisponde alla dichiarazione di una procedura, il secondo al corpo della prima. Spesso i due termini sono utilizzati in modo intercambiabile.

Generalizzazione

- La relazione di generalizzazione può essere interpretata in vari modi
- Dal punto di vista concettuale esprime la relazione che esiste tra concetti più generici e concetti più specifici
- Dal punto di vista software l'interpretazione più comune è quella che implica il meccanismo dell'ereditarietà
- il principio fondamentale da rispettare quando si usa l'ereditarietà è il principio di **sostituibilità** (Liskov):
  Ovunque ci si attende un'istanza dell'entità più generica deve essere possibile utilizzare, senza alterare il corretto funzionamento del sistema, un'istanza di quella più specifica

Note e commenti

- Le note sono commenti aggiuntivi che possono apparire in qualunque tipo di diagramma UML
- Possono essere collegate con una linea tratteggiata agli elementi cui fanno riferimento o essere disegnate isolate
- Per inserire un commento interno di un elemento del diagramma si scrive il testo del commento dopo due trattini −−



Dipendenza

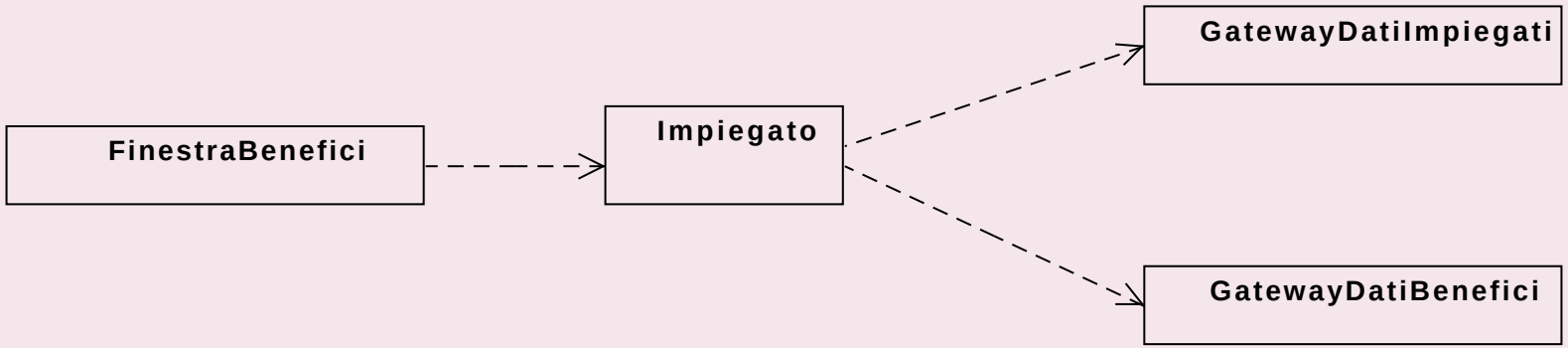
- Tra due elementi di un diagramma esiste una relazione di **dipendenza** se una modifica alla definizione di uno (fornitore o *supplier*) può comportare un cambiamento all'altro (il *client*)
- Nel caso delle classi la dipendenza può essere dovuta a varie cause: una classe invoca i metodi di un'altra, la usa come tipo di un suo campo o come tipo di un parametro di un suo metodo
- UML consente di indicare dipendenze tra ogni tipo di elemento
- La dipendenza è una relazione unidirezionale, si indica con una linea tratteggiata terminante con una freccia che ne specifica la direzione
- UML prevede vari tipi di dipendenza, ognuna con una particolare semantica e varie parole chiave (vedi slide seguente)
- È importante sottolineare che la dipendenza non è una relazione transitiva

Dipendenza

Parola chiave	Significato
≪call≫	La sorgente invoca un'operazione della classe destinazione
≪create≫	La sorgente crea istanze della classe destinazione
≪derive≫	La sorgente è derivata dalla classe destinazione
≪instantiate≫	La sorgente è un'istanza della classe destinazione. (In questo caso se la sorgente è una classe, ed è istanza della destinazione, quest'ultima deve essere una meta-classe)
≪permit≫	La classe destinazione permette alla sorgente di accedere ai suoi campi privati
≪realize≫	La sorgente è un'implementazione di una specifica o di un'interfaccia definita dalla destinazione (si veda in seguito)
≪refine≫	Il raffinamento indica una relazione tra livelli semantici differenti; la classe sorgente ad esempio potrebbe essere una classe di progettazione, la destinazione una classe di analisi
≪substitute≫	La sorgente è sostituibile alla destinazione (si veda in seguito)
≪trace≫	Usata per tenere traccia di cose come i requisiti o di come i cambiamenti a una parte di modello si collegano ad altre sue parti
≪use≫	La sorgente richiede la destinazione per la sua implementazione

Dipendenza

Esempio



- La classe `FinestraBenefici` fa parte dell'interfaccia utente (fa parte del livello detto di **presentazione**)
- Essa dipende da `Impiegato` che è un oggetto del **dominio** il quale incapsula il comportamento fondamentale del sistema
- Si può cambiare la classe `FinestraBenefici` senza che le modifiche abbiano effetto sulla classe `Impiegato` o su altre classi del dominio
- Se cambia una delle classi `GatewayDatiImpiegati` o `GatewayDatiBenefici` può essere necessario cambiare `Impiegato`