Introduzione

Cos'è UML

UML (Unified Modeling Language) è un linguaggio di modellazione grafica per descrivere, specificare, costruire e documentare i componenti di sistemi software. Si basa sul paradigma object-oriented e utilizza notazioni grafiche formalmente definite.

Approcci a UML

- **Come Abbozzo (Sketch)**: approccio più utilizzato
  - **Forward Engineering**: si progettano i diagrammi UML prima del codice (tipico in aziende grandi)
  - **Reverse Engineering**: si costruiscono i diagrammi dal codice esistente (tipico in aziende piccole)
- **Come Progetto**: approccio più ingegneristico
  - **Documento di Definizione Prodotto**: descrive formalmente il sistema con elevato dettaglio, senza lasciare interpretazioni al programmatore
  - **Definizione Interfacce tra Sottosistemi**: i programmatori progettano e sviluppano componenti in autonomia

Principio di Pareto (80/20)

Il 20% dei diagrammi viene usato l'80% delle volte. I diagrammi più utilizzati sono:

- Diagrammi delle Classi (struttura)
- Diagrammi di Attività (comportamento)
- Diagrammi dei Casi d'Uso (comportamento)
- Diagrammi di Sequenza (comportamento)

Ruolo del Diagramma delle Classi

Il **Diagramma delle Classi** è un **diagramma di struttura** che:

- Descrive i **tipi** degli oggetti presenti nel sistema
- Mostra le **relazioni statiche** tra questi tipi
- Viene utilizzato principalmente in fase di **Specifica Tecnica** e **Definizione di Prodotto**

Struttura di una Classe

Rappresentazione Grafica

Una classe è rappresentata da un **rettangolo diviso in tre compartimenti**:

```
┌──────────────────┐
│   NomeClasse     │  ← Obbligatorio (nome della classe)
├──────────────────┤
│   Attributi      │  ← Opzionale (campi/proprietà)
├──────────────────┤
│   Operazioni     │  ← Opzionale (metodi/funzioni)
└──────────────────┘
```

Solo il nome della classe è obbligatorio.

Sintassi degli Attributi

```
visibilità nome: tipo [molteplicità] = valorePredefinito
```

Esempio:

```
- nome: String
+ età: int = 0
# indirizzo: String[0..1]
```

Sintassi delle Operazioni

```
visibilità nome(parametri): tipoRitorno
```

Esempio:

```
+ calcolaStipendio(ore: int): double
- validaCredenziali(user: String, pwd: String): boolean
# aggiorna(): void
```

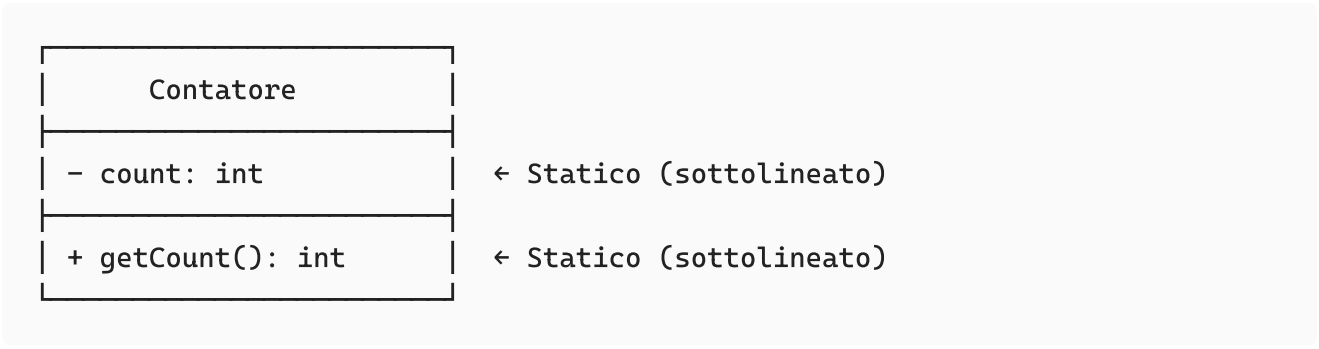
Visibilità

Simbolo	Visibilità	Significato
`+`	**public**	Accessibile da qualsiasi classe
`-`	**private**	Accessibile solo all'interno della classe
`#`	**protected**	Accessibile nella classe e nelle sottoclassi
`~`	**package**	Accessibile solo nel package

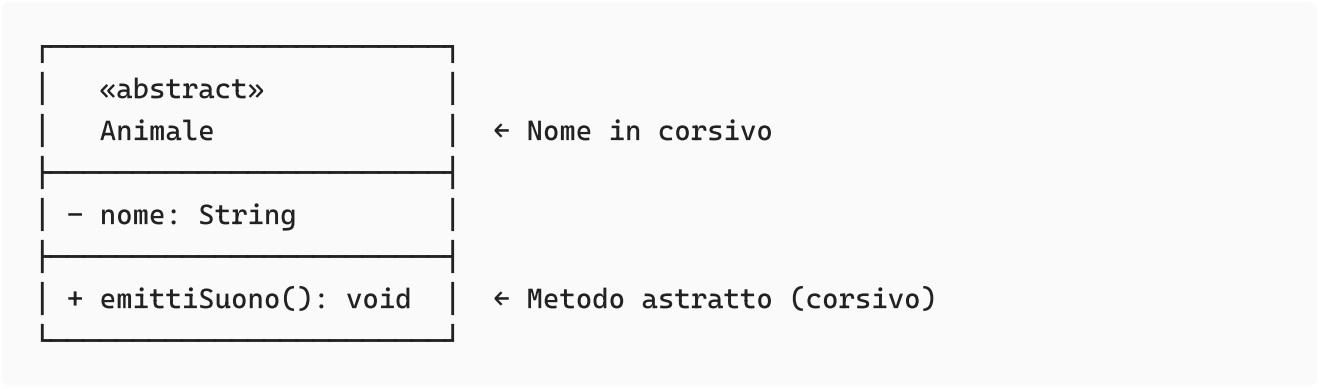
Elementi Speciali

Attributi e Metodi Statici

Si rappresentano **sottolineando** il nome:



Classi Astratte



Alternative per indicare classi astratte:

- Nome della classe in *corsivo*
- Stereotipo `{abstract}` o `«abstract»`

Relazioni tra Classi

1. Dipendenza (Dependency)

Relazione più debole: A usa B temporaneamente.

Notazione: linea tratteggiata con freccia aperta (`- - - - - →`)

Significato nel codice:

```java
public class Servizio {
    public void elabora(Logger logger) {  // Parametro
        logger.log("Elaborazione in corso");
    }
}
```

Quando usarla:

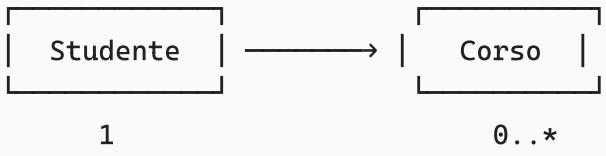
- B è un **parametro** di un metodo di A
- B è una **variabile locale** in A
- A crea istanze temporanee di B

---

2. Associazione (Association)

Relazione has-a: A ha un riferimento persistente a B.

Notazione: linea continua (`———`)



Molteplicità:

- `1` : esattamente uno
- `0..1` : zero o uno (opzionale)
- `0..*` o `*` : zero o più
- `1..*` : uno o più
- `n..m` : da n a m

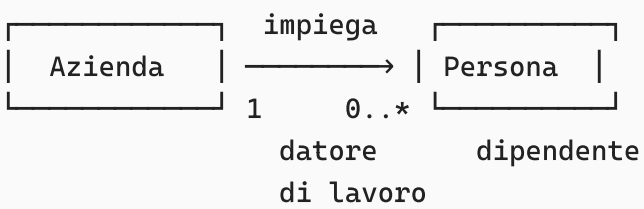
Significato nel codice:

```java
public class Studente {
    private List<Corso> corsi;  // Attributo persistente
}
```

Navigabilità:

- Freccia indica la direzione della navigabilità
- Se non c'è freccia, l'associazione è bidirezionale

Ruoli:



---

3. Aggregazione (Aggregation)

Relazione has-a + whole-part: A contiene B, ma B può esistere indipendentemente.

Notazione: rombo vuoto (`◇———`)

Significato nel codice:

```java
public class Università {
    private List<Studente> studenti;

    public Università(List<Studente> studenti) {
        this.studenti = studenti;  // Reference ricevuto, non creato
    }
}
```

Caratteristiche:

- La parte può esistere indipendentemente dal tutto
- Il contenitore non è responsabile della creazione/distruzione
- Il reference viene **passato dall'esterno** (constructor injection, setter)

---

4. Composizione (Composition)

Relazione has-a + whole-part + ownership: A contiene B e ne controlla il ciclo di vita.

Notazione: rombo pieno (`◆———`)

Significato nel codice:

```
public class Auto {
    private Motore motore;

    public Auto() {
        this.motore = new Motore();  // Creazione e ownership
    }
}
```

Caratteristiche:

- La parte **non può esistere** senza il tutto
- Il contenitore è responsabile della **creazione** e **distruzione**
- Se il contenitore viene distrutto, anche la parte viene distrutta
- Il reference viene **creato internamente** (new)

Differenza Aggregazione vs Composizione:

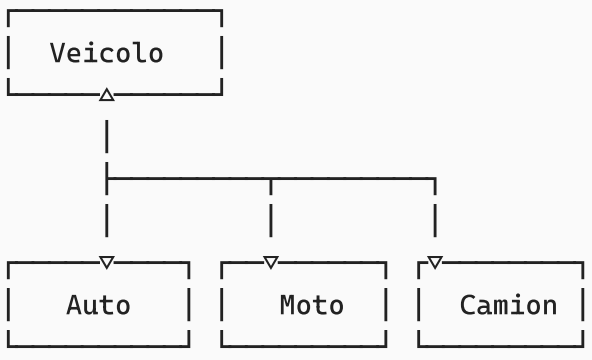
Aspetto	Aggregazione	Composizione
Ciclo di vita	Indipendente	Dipendente
Creazione	Esterna (passata)	Interna (new)
Ownership	NO	SÌ
Distruzione	Parte sopravvive	Parte viene distrutta

---

5. Generalizzazione / Ereditarietà (Inheritance)

Relazione is-a: B è una specializzazione di A.

Notazione: linea continua con triangolo vuoto (`——▷`)



Significato nel codice:

```java
public class Auto extends Veicolo {
    // Auto eredita da Veicolo
}
```

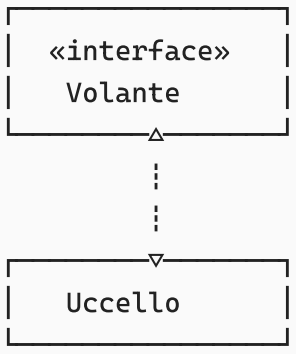
Ereditarietà Multipla: Rappresentata con più frecce che convergono verso il figlio.

---

6. Realizzazione (Realization)

Implementazione di interfaccia: A implementa l'interfaccia B.

Notazione: linea tratteggiata con triangolo vuoto (`- - - - ▷`)



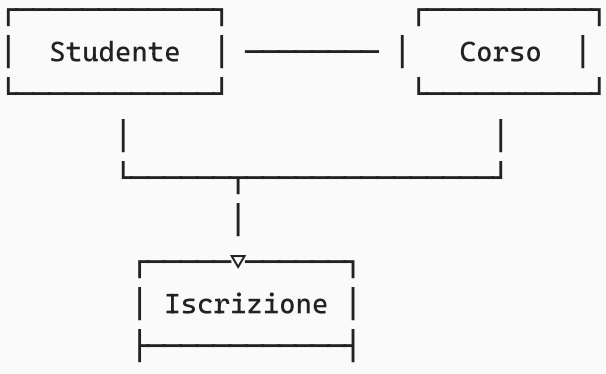
Significato nel codice:

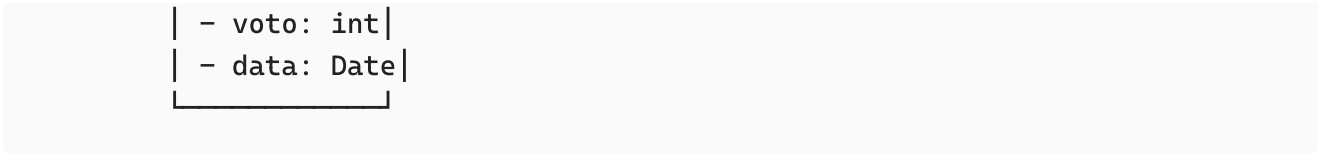
```java
public class Uccello implements Volante {
    @Override
    public void vola() { /* implementazione */ }
}
```

---

7. Classi di Associazione

Quando un'associazione ha **attributi propri**, si usa una **classe di associazione**.





Significato nel codice:

```java
public class Iscrizione {
    private Studente studente;
    private Corso corso;
    private int voto;
    private Date dataIscrizione;
}
```

Elementi Avanzati

Interfacce

Notazione Stereotype (UML 1.x)

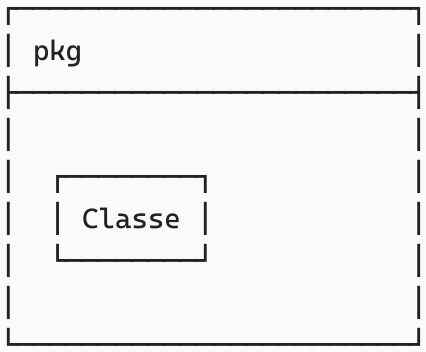
Notazione Ball (UML 2.x)

Il pallino rappresenta l'interfaccia implementata.

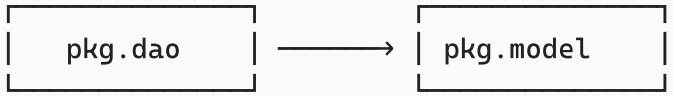
Package

Rappresentano il raggruppamento logico di classi.

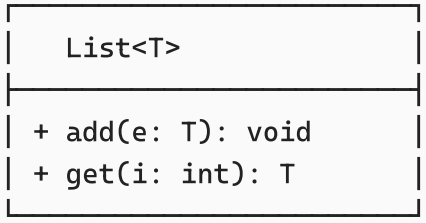
Notazione: cartella



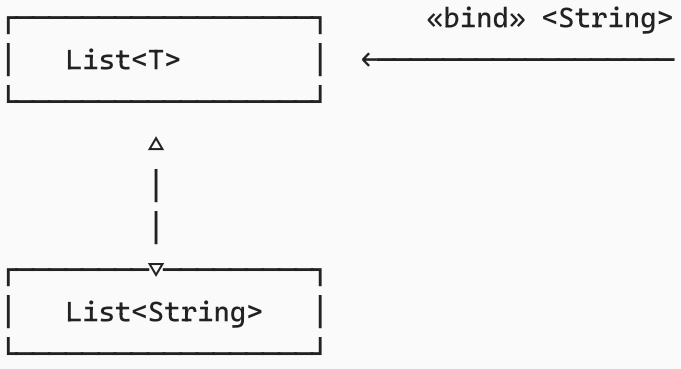
Dipendenze tra Package:



Classi Parametriche (Generics)



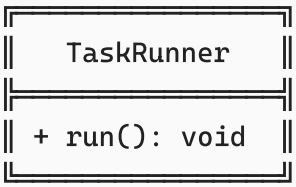
Binding:



Classi Attive

Classi che **eseguono e controllano il proprio thread**.

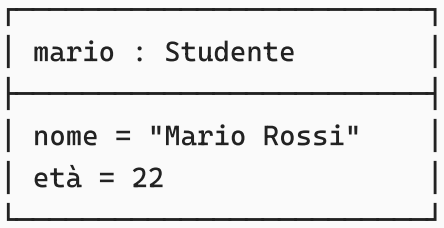
Notazione: rettangolo con bordo doppio



Diagrammi degli Oggetti

Rappresentano **istanze concrete** delle classi in un momento specifico (**snapshot**).

Notazione: `nomeIstanza : NomeClasse`



Caratteristiche:

- Non ci sono metodi, solo attributi con valori concreti
- Possono essere istanze di classi astratte (per scopi esemplificativi)

Best Practices

1. Livello di Astrazione

- **Non mescolare diversi livelli di dettaglio** nello stesso diagramma
- Parti con una **visione ad alto livello** (solo classi principali)
- Approfondisci selettivamente le aree critiche

2. Molteplicità

- **Specifica sempre la molteplicità** nelle associazioni
- Usa `*` invece di `0..*` per brevità quando non c'è ambiguità

- La molteplicità `1` deve essere esplicita

3. Navigabilità

- Mostra la freccia **solo se la navigabilità è unidirezionale**
- Nessuna freccia = associazione bidirezionale
- La navigabilità influenza l'implementazione del codice

4. Nomenclatura

- **Nomi di classi**: PascalCase, sostantivi singolari
- **Nomi di attributi**: camelCase, sostantivi
- **Nomi di metodi**: camelCase, verbi
- **Nomi di interfacce**: aggettivi (es. `Comparable`, `Serializable`)

5. Organizzazione

- Usa **package** per raggruppare classi correlate
- Minimizza le **dipendenze cicliche** tra package
- Mantieni le dipendenze che vanno **dall'alto verso il basso**

6. Relazioni

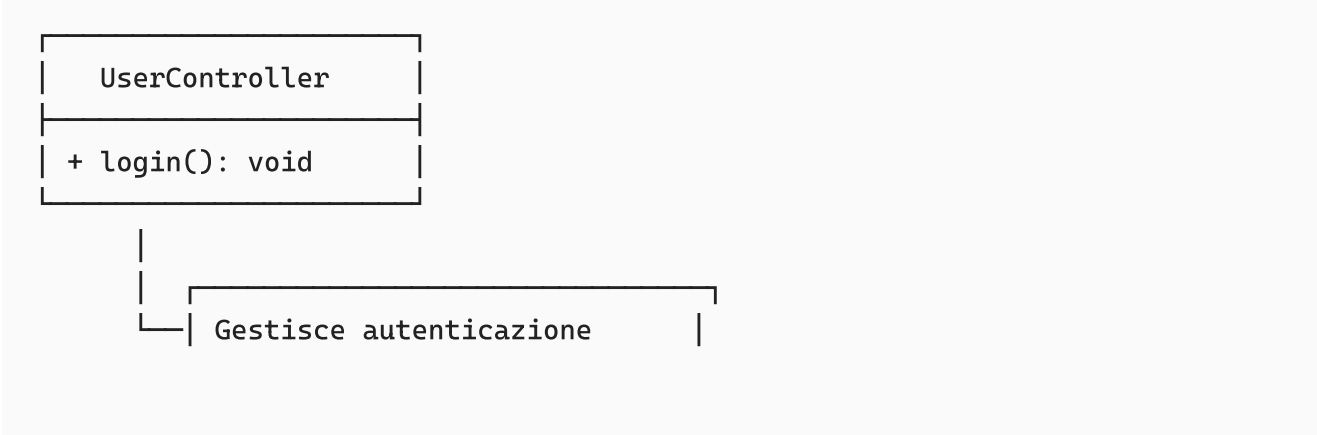
Ordine di preferenza (dalla più debole alla più forte):

1. **Dipendenza** (se possibile)
2. **Associazione**
3. **Aggregazione** (quando il ciclo di vita è indipendente)
4. **Composizione** (quando c'è ownership)
5. **Ereditarietà** (solo per relazioni is-a autentiche)

Principio: "Favorisci la composizione rispetto all'ereditarietà"

7. Responsabilità

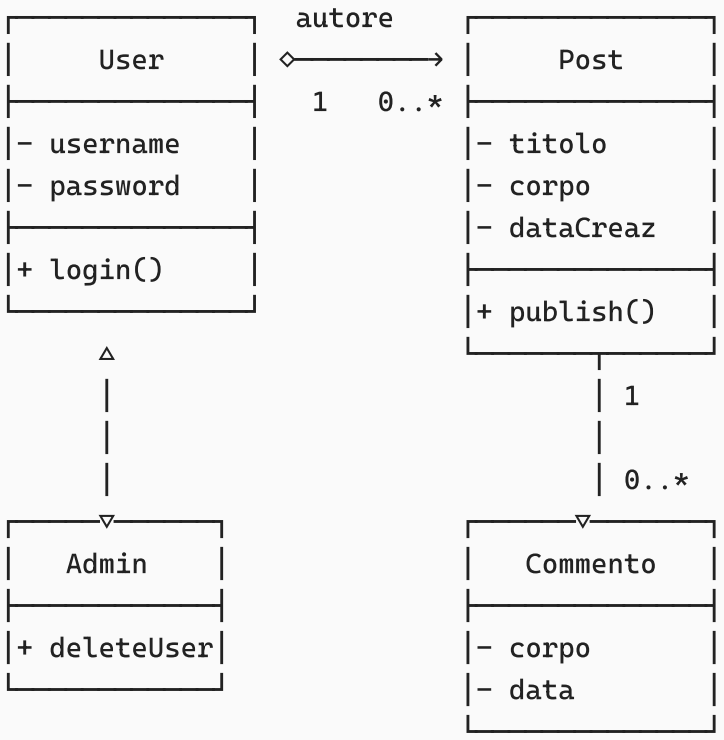
Aggiungi **note di responsabilità** per chiarire il ruolo della classe:



```
          | e autorizzazione utenti       |
          └──────────────────────────────┘
```

Esempi Pratici

Esempio: Sistema Blog



Legenda:

- `User` è **aggregato** da `Post` (autore)
- `Admin` **eredita** da `User`
- `Commento` è in **composizione** con `Post`

Riepilogo delle Notazioni

Relazione	Simbolo	Forza	Ciclo di vita
Dipendenza	`- - - →`	Molto debole	Temporaneo
Associazione	`———→`	Debole	Indipendente
Aggregazione	`◇———→`	Media	Indipendente
Composizione	`◆———→`	Forte	Dipendente
Ereditarietà	`———▷`	Molto forte	N/A

Relazione	Simbolo	Forza	Ciclo di vita
Realizzazione	`- - - ▷`	Media	N/A

Riferimenti

- **OMG Homepage**: www.omg.org
- **UML Homepage**: www.uml.org
- **UML Distilled**, Martin Fowler, Pearson (Addison Wesley), 2004
- **Learning UML 2.0**, Kim Hamilton, Russell Miles, O'Reilly, 2006
