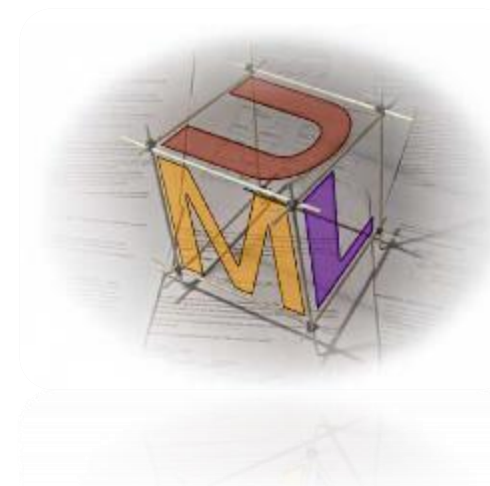


Object Oriented Design

UML – class diagram

cos'è UML

- è un linguaggio di **progettazione**, da non confondere con i linguaggi di **programmazione** (Python, C, C++, Java,...)
- fornisce una serie di **diagrammi** per rappresentare ogni tipo di modellazione
- alcuni ambienti di programmazione sono in grado di convertire diagrammi UML in codice e viceversa

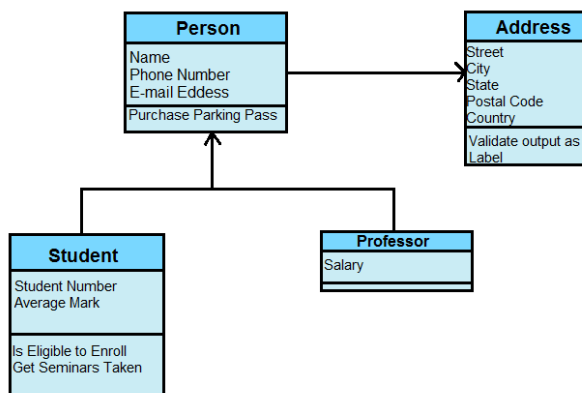


diagrammi UML

- diagramma dei casi d'uso (use case)
- ***diagramma delle classi (class)***
- diagramma di sequenza (sequence)
- diagramma di collaborazione (collaboration)
- diagramma di stato (statechart)
- diagramma delle attività (activity)
- diagramma dei componenti (component)
- diagramma di distribuzione (deployment)

diagramma delle classi

- rappresenta le **classi** (collezioni di oggetti) che compongono il sistema
- di ogni classe rappresenta lo **stato** e il **comportamento** (attributi e operazioni)
- specifica, mediante associazioni, le **relazioni** fra le classi



classe metodi e attributi

SchedaAnagrafica
-nome:String -cognome:String
+getNome():String +setNome(nome:String):void +getCognome():String +setCognome(cognome:String):voi

modificatori

EsempioModificatori
+attributoPubblico #attributoProtected -attributoPrivato
+metodoPubblico #metodoProtected -metodoPrivato

- + ***public***: libero accesso
- # ***protected***: accessibile dalle sottoclassi
- - ***private***: accessibile solo all'interno della classe
- ***static***: accessibili anche senza creare istanze

associazione

- un'***associazione*** rappresenta la possibilità che un oggetto (istanza di una classe) ha di inviare un messaggio a un altro oggetto (istanza di un'altra classe)
- in UML viene rappresentata con una freccia

associazione: esempio



```

public class Automobile {
    private Motore motore;

    public void accendi() {
        motore.inserisciMiscela();
        motore.accendiCandele();
    }
}
  
```

```

public class Motore {
    public void inserisciMiscela() {...};
    public void accendiCandele() {...};
}
  
```

```

class Motore:
    def __init__(self):
        self._miscela = 0
        self._acceso = False

    def inserisci_miscela(self, litri):
        self._miscela += litri

    def accendi_candela(self):
        if self._miscela > 0:
            self._acceso = True

class Automobile:
    def __init__(self, motore):
        self._motore = motore

    def accendi(self):
        self._motore.accendi_candela()
  
```

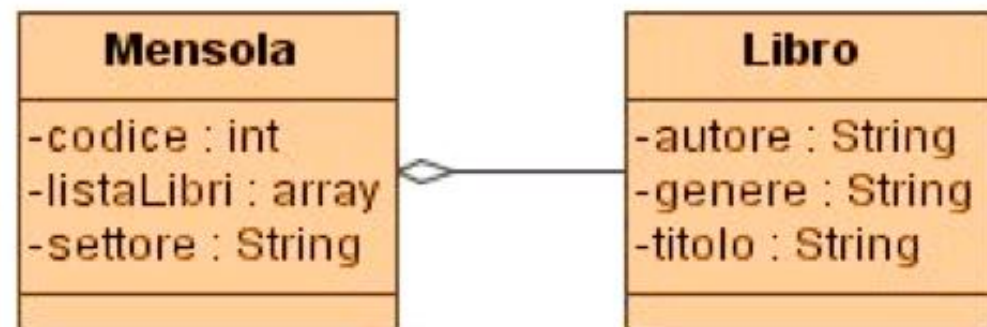

dipendenza

- la **dipendenza** indica che un oggetto di una classe può chiamare i metodi di un oggetto di un'altra classe pur senza possederne un'istanza
- la classe dipendente presuppone l'esistenza della classe da cui dipende
- non vale il viceversa
- in UML la dipendenza viene rappresentata con una freccia tratteggiata
- in java tipicamente l'oggetto dipendente riceve un'istanza dell'oggetto da cui dipende come argomento di una chiamata a metodo

aggregazione

- l'**aggregazione** rappresenta un'associazione uno a molti
- esprime concetto "è parte di " (***part of***), che si ha quando un insieme è relazionata con le sue parti
- in UML l'aggregazione viene rappresentato con una freccia con la punta a diamante

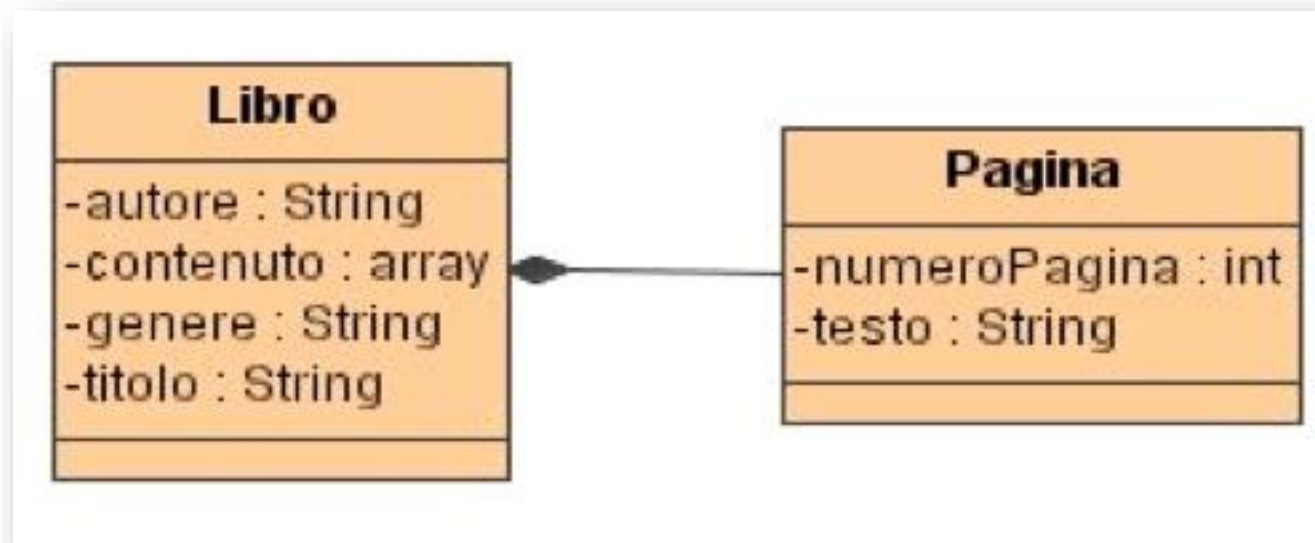
esempio di aggregazione



composizione

- una ***composizione*** è una relazione uno a molti che implica una forma di ***esclusività***
- è un caso particolare di aggregazione in cui:
 - la parte (componente) ***non può esistere da sola***, cioè senza la classe composto
 - una componente appartiene ad un solo composto
- la distruzione dell'oggetto che rappresenta il “tutto” provoca la distruzione a catena delle “parti”
- il diamante si disegna pieno

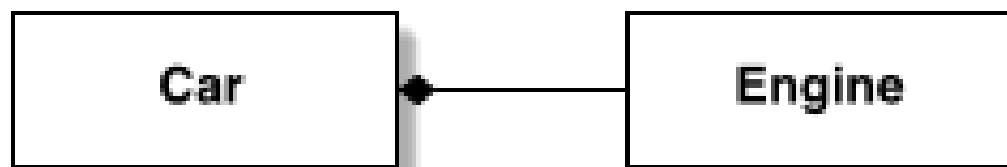
esempio di composizione



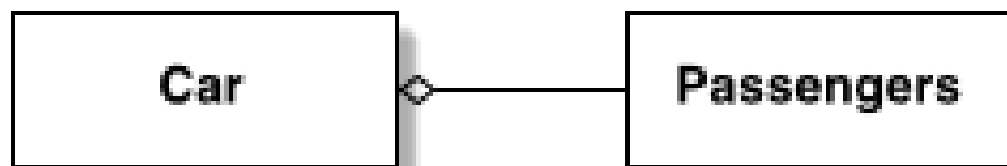
aggregazione / composizione

- per ***distinguere*** l'aggregazione dalla composizione possiamo chiederci che ***destino*** devono avere gli ***oggetti-parte*** nel momento in cui viene distrutto ***l'oggetto-tutto***
- se ***non*** ha senso che gli oggetti-parte ***sopravvivano*** all'oggetto-tutto, allora siamo di fronte a una relazione ***compositiva***
(la cancellazione del rombo pieno che la rappresenta graficamente richiede la cancellazione del bordo e dell'area interna)
- se ha invece senso che gli oggetti-parte ***sopravvivano*** autonomamente all'oggetto-tutto, allora si ha una relazione ***aggregativa***
(la cancellazione del rombo vuoto che la rappresenta graficamente avviene cancellando il bordo, ma non richiede la cancellazione dell'area interna)

composizione vs aggregazione

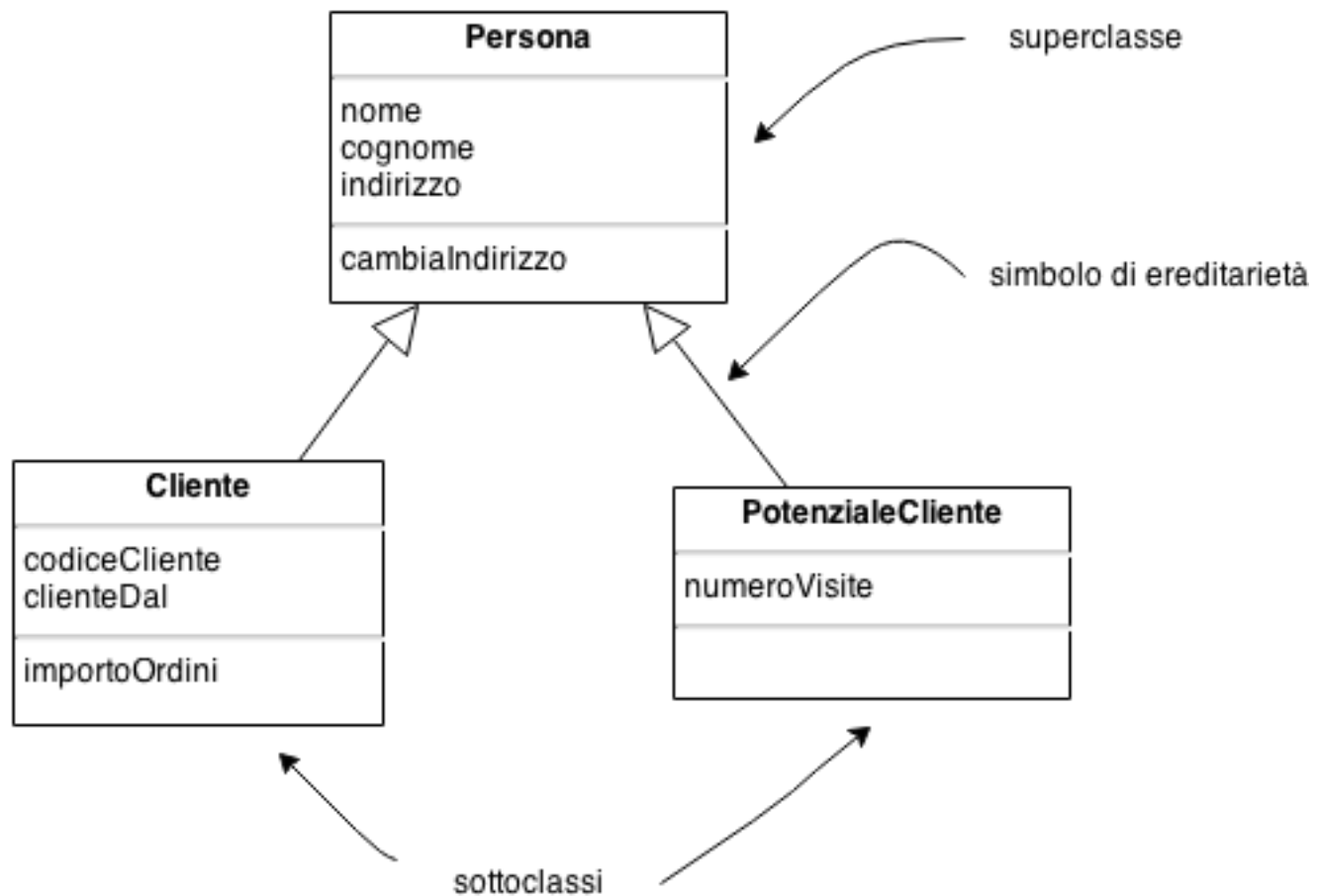


Composition: every car has an engine.



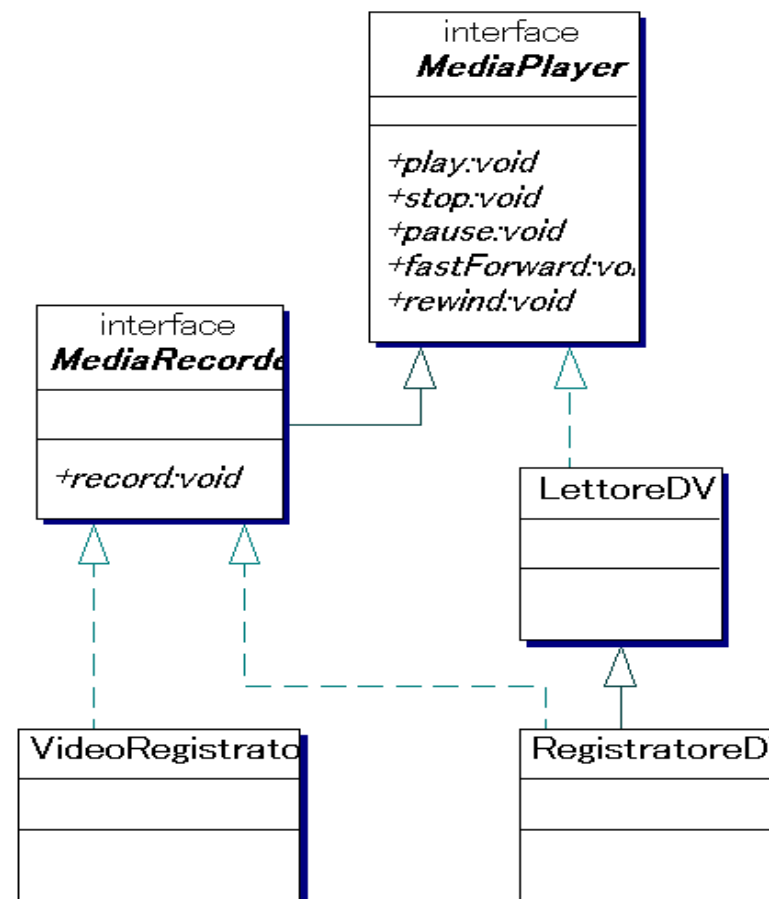
Aggregation: cars may have passengers, they come and go

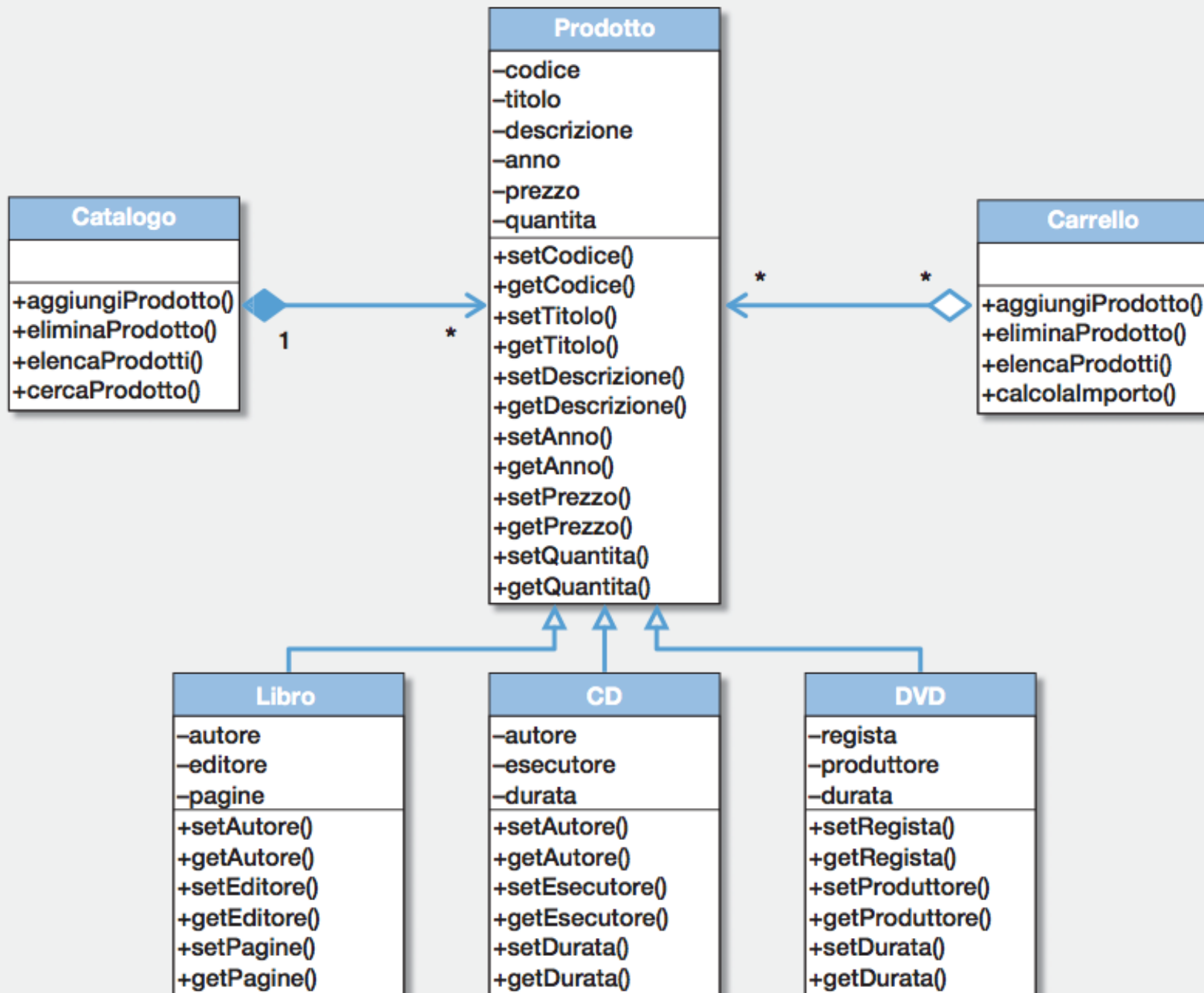
ereditarietà



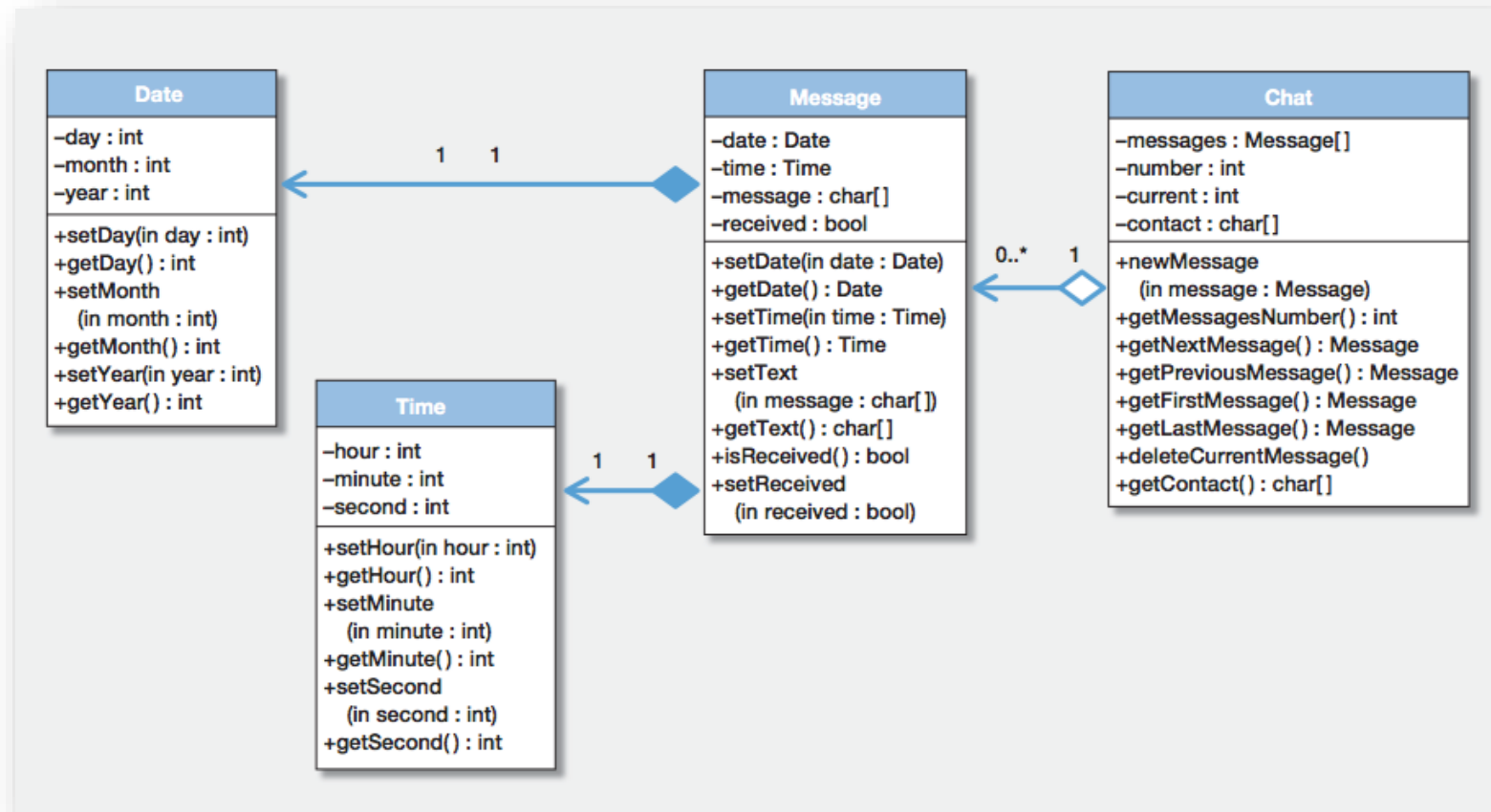
ereditarietà multipla

- in Python e C++ è ammessa l'ereditarietà multipla
- in Java non è ammessa l'ereditarietà multipla
 - le interfacce permettono di ovviare a questo problema: una classe può ereditare da una sola classe ma implementare varie interfacce





esempio



standard UML

- ***molteplicità***
 - 1 esattamente una istanza
 - N esattamente N istanze
 - 1..* una o più istanze
 - 0..* zero o più istanze
 - 1..N una o più istanze (massimo N)
 - 0..N zero o più istanze (massimo N)

esempio “completo”

