Il paradigma Orientato agli Oggetti Modellazione delle Relazioni tra Classi mediante UML (seconda parte)

Programmazione II corso A
Corso di Laurea in ITPS
Università degli Studi di Bari Aldo Moro
a.a. 2022-2023
ultima modifica: 29 novembre 2022

Testo di riferimento: «UML DISTILLED» di Martin Fowler, Pearson

Sommario

- Il linguaggio UML *(già affrontati)*
- Classi ed oggetti in UML *(già affrontati)*
- Relazioni tra classi
  - Associazione
  - Aggregazione
  - Classe associativa
  - Generalizzazione, «insieme generalizzazione» (generalizationset)
  - Dipendenza
  - Implementazione Interfaccia
- Aggregazione/Composizione Vs. ereditarietà
- Classi Astratte
- Interfacce
  - Ereditarietà multipla
- Classi nidificate

Relazione di Generalizzazione

relazione che permette di creare una classe (figlia o sottoclasse) ereditando attributi ed operazioni di un'altra classe (genitore o supeclasse)

- Si possono aggiungere relazioni di generalizzazione per catturare attributi, operazioni e relazioni in un elemento della classe padre e poi riutilizzarle in uno o più elementi della classe figlia.
- la classe figlia eredita gli attributi, le operazioni e le relazioni del genitore;
- La definizione della classe *figlia* prevede solo gli attributi, le operazioni o le relazioni che sono distinte dalla classe *padre*.

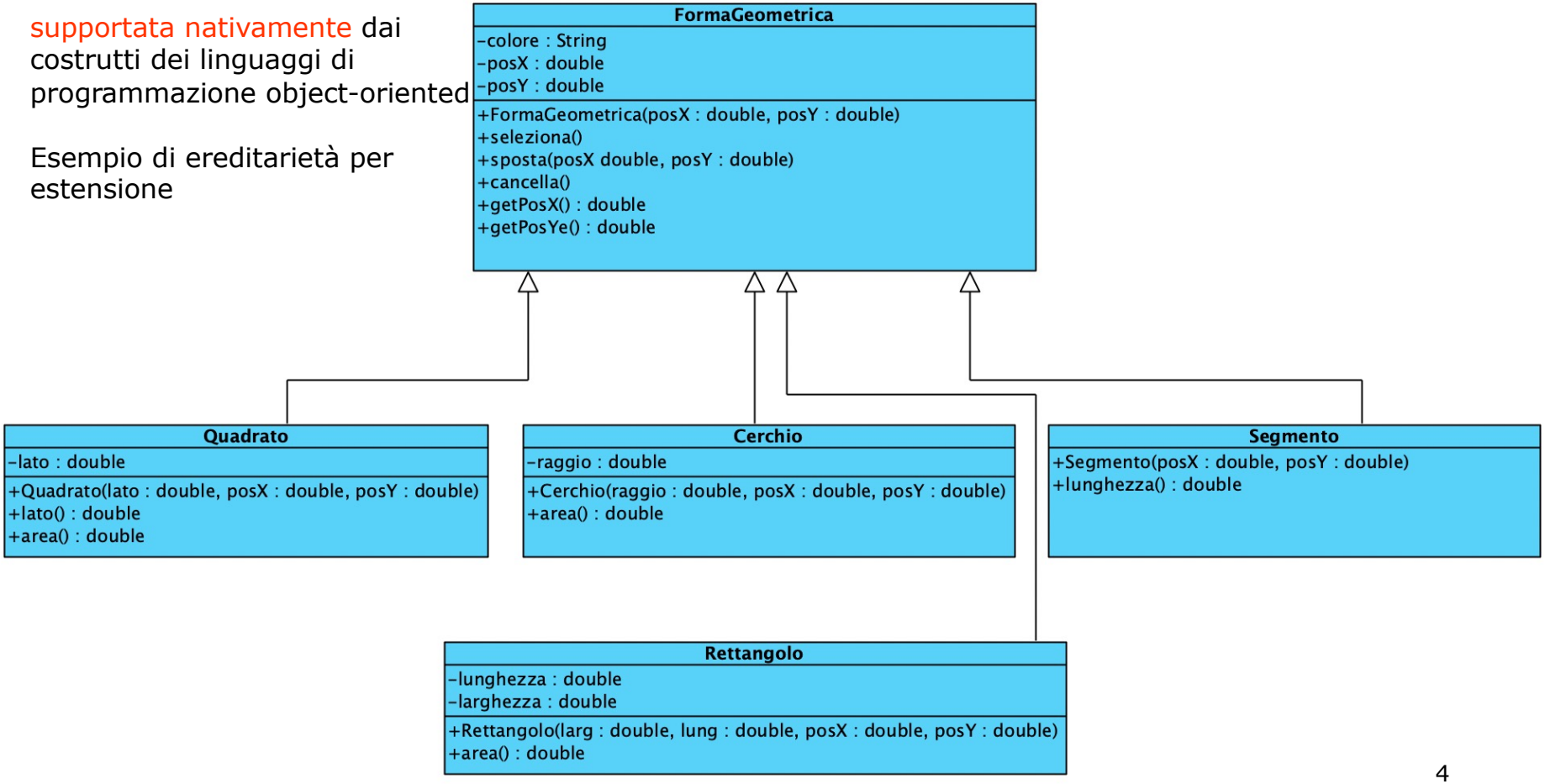
NOTA BENE

ogni istanza di una classe figlia contiene sempre un'istanza della classe padre

Relazione di Generalizzazione: esempio e modellazione in UML

supportata nativamente dai costrutti dei linguaggi di programmazione object-oriented

Esempio di ereditarietà per estensione



Relazione di Generalizzazione: perché usarla?

Per almeno i tre seguenti motivi:

- Supporto del polimorfismo:
  - il polimorfismo aumenta la flessibilità del software.
  - Aggiungendo una nuova sottoclasse ed ereditando automaticamente il comportamento della superclasse.

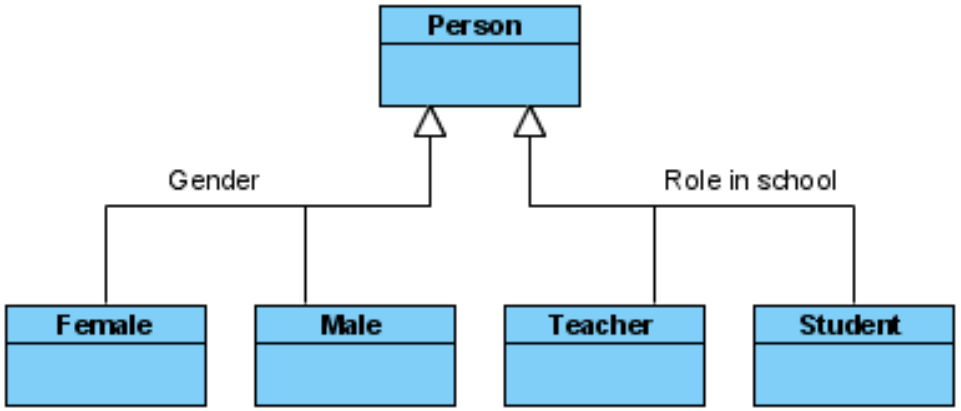
- Strutturare la descrizione degli oggetti:
  - Formare una tassonomia (classificazione), organizzando gli oggetti secondo le loro somiglianze. È molto più profondo che modellare ogni classe individualmente e in isolamento di altre classi simili.

- Rendere possibile il riuso del codice:

  - Il riuso è più produttivo che scrivere ripetutamente codice da zero.

Relazione di GeneralizationSet

Definisce un insieme di relazioni di generalizzazione che descrivono il modo in cui **una superclasse può essere partizionata in specifici sottotipi**;



Esempio: la classe persona potrebbe essere partizionata in 2 sottotipi:

- Gender: sottoclassi «Female» e «Male»
- Role in school: sottoclassi «Teacher» e «Student»

Proprietà della GeneralizationSet

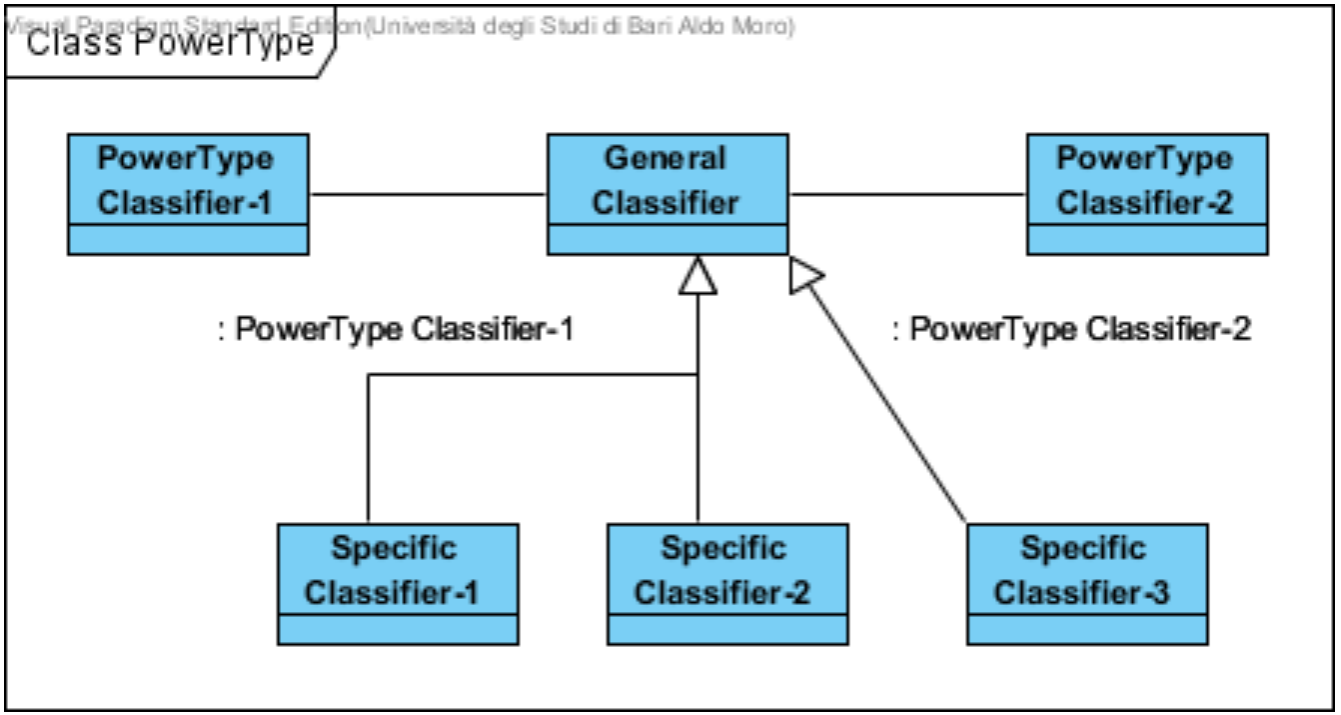
- *IsCovering* (*COMPLETE* o *INCOMPLETE*): specifica se ci siano o meno istanze della superclasse che non siano istanze di almeno uno dei propri classificatori specifici (sottoclassi)
- *IsDisjoint* (*DISJOINT* o *OVERLAPPING*): specifica se una superclasse può essere istanziata da più di una sottoclasse(sovrapposizione delle sottoclassi)
- Le possibili combinazioni delle proprietà sono:

  - {COMPLETE, DISJOINT} – L'INSIEME GENERALIZZAZIONE È COMPLETO E LE SOTTOCLASSI NON HANNO ISTANZE COMUNI
  - {INCOMPLETE, DISJOINT} – L'INSIEME GENERALIZZAZIONE NON È COMPLETO E LE PROPRIE SOTTOCLASSI NON HANNO ISTANZE COMUNI.
  - {COMPLETE, OVERLAPPING} – L'INSIEME GENERALIZZAZIONE È COMPLETO E LE PROPRIE SOTTOCLASSI DIVIDONO ISTANZE COMUNI
  - {INCOMPLETE, OVERLAPPING} – L'INSIEME GENERALIZZAZIONE NON È COMPLETO E LE PROPRIE SOTTOCLASSI DIVIDONO ISTANZE COMUNI.

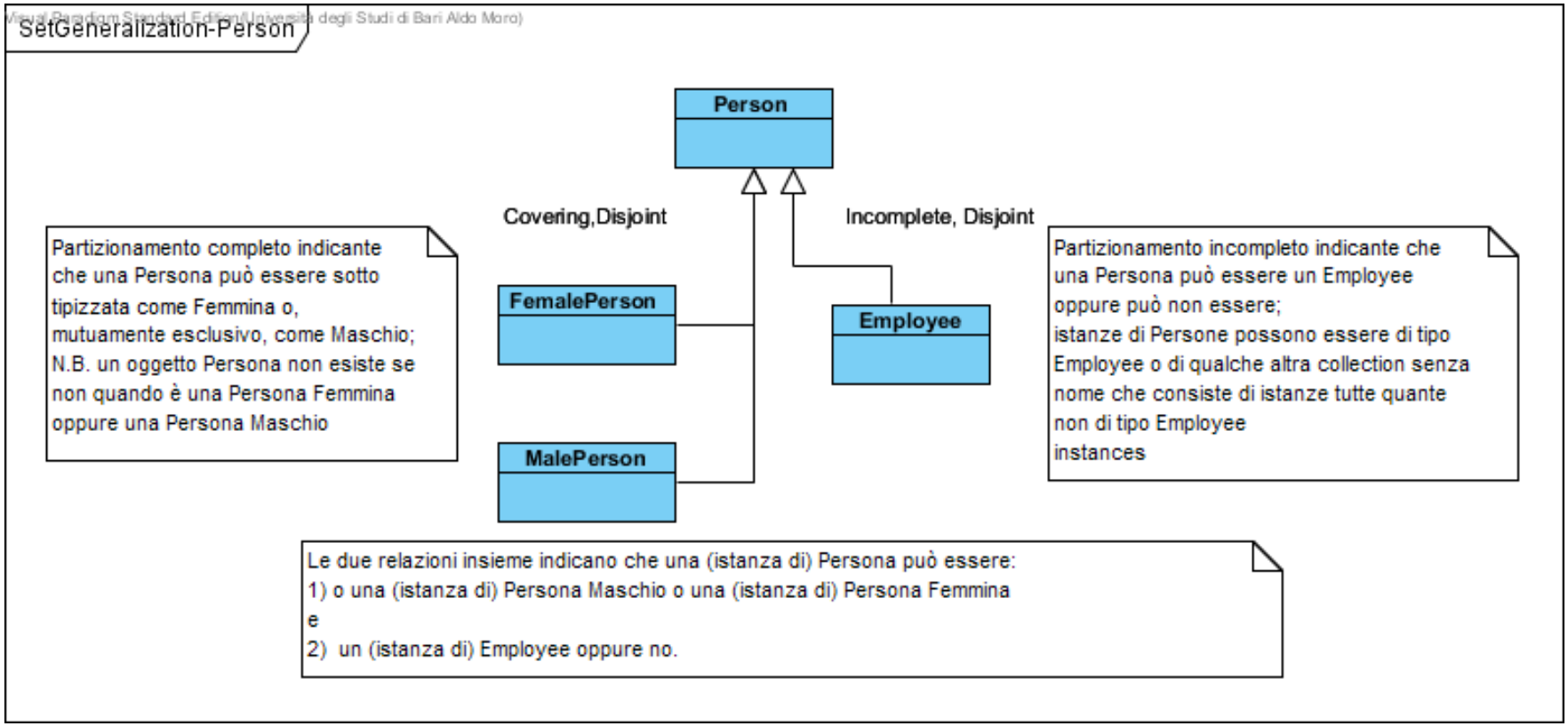
  * DEFAULT È {INCOMPLETE, DISJOINT}

Classe «powertype»

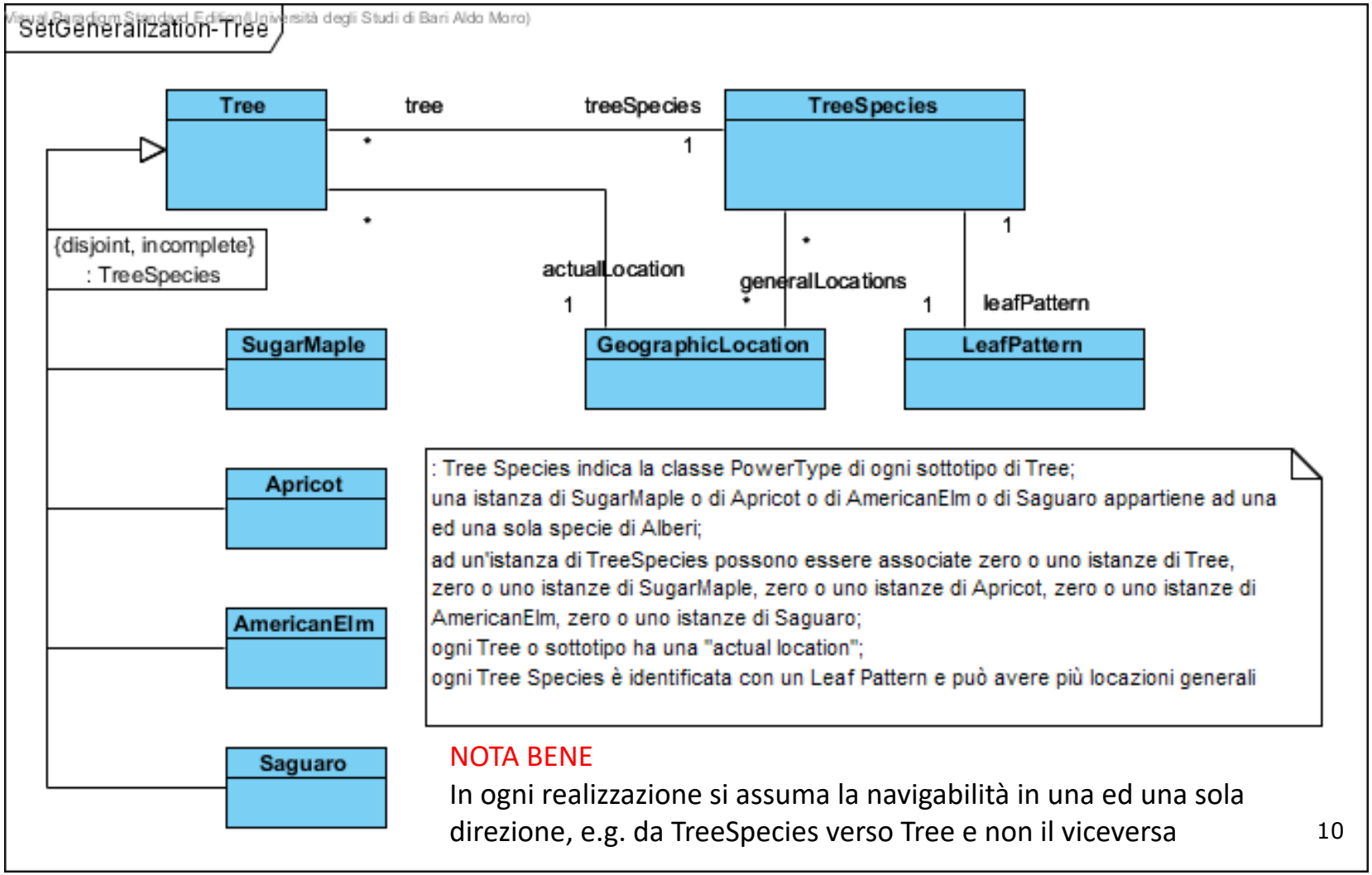
Classe in relazione con uno specifico sottotipo della superclasse



Esempio 1 di Generalizationset



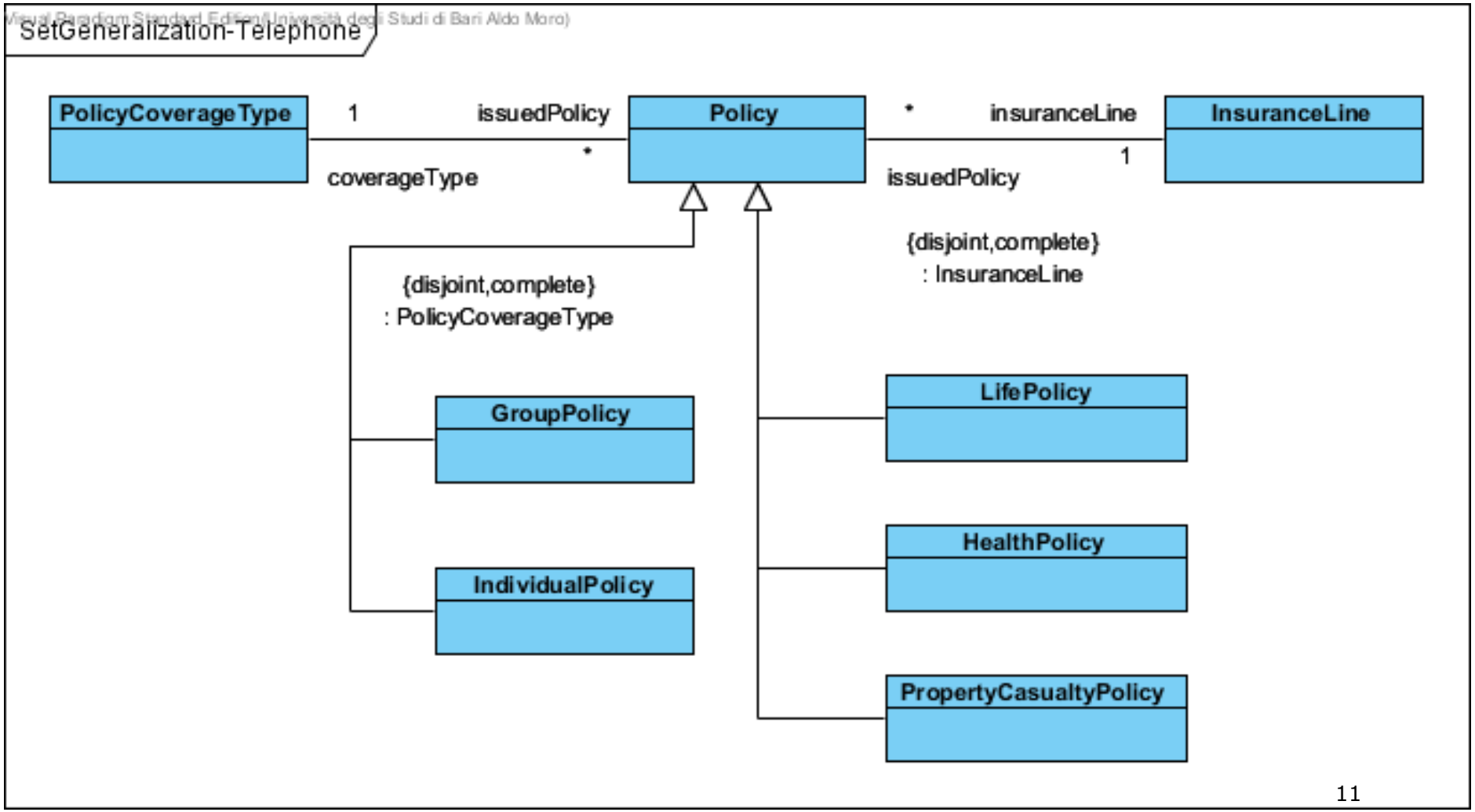
Esempio 2 di Generalizationset



NOTA BENE

In ogni realizzazione si assuma la navigabilità in una ed una sola direzione, e.g. da TreeSpecies verso Tree e non il viceversa

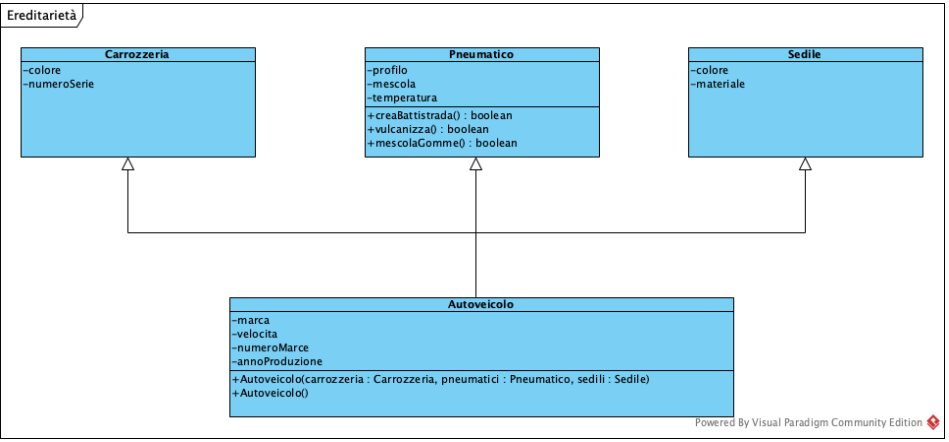
Esempio 3 di Generalizationset

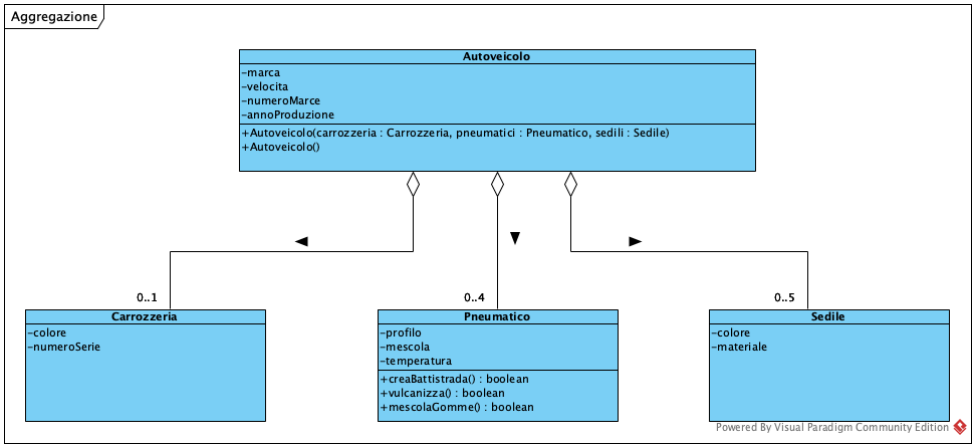


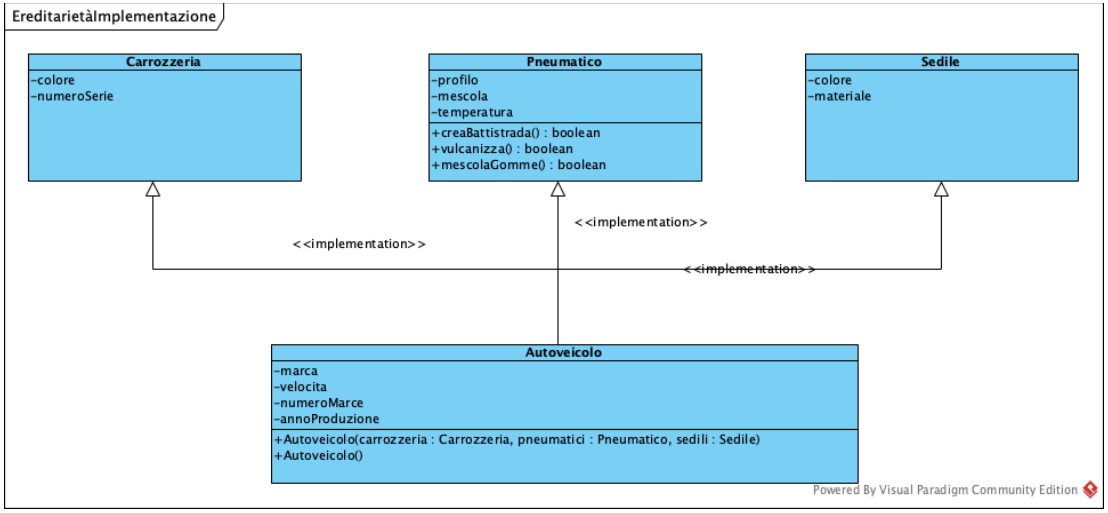
Aggregazione Vs. Ereditarietà

Esempio autoveicolo: Aggregazione o Ereditarietà?

Quale relazione usare tra Autoveicolo, Carrozzeria, Pneumatico e Sedile?







Esempio autoveicolo: Aggregazione o Ereditarietà?

Ricorso all'ereditarietà multipla: fusione dei concetti Autoveicolo e Carrozzeria
Errore concettuale: un autoveicolo è composto da una carrozzeria, da pneumatici, da sedili ma il suo comportamento non corrisponde all'unione di queste differenti parti.
Esempio: mescolaGomme(), creaBattistrada(), vulcanizzazione() operazioni proprie di penumatico sarebbero concettualmente errate per Autoveicolo

Ricorso all'ereditarietà di implementazione: Autoveicolo erediterebbe l'implementazione (attributi e metodi) di Carrozzeria, Pneumatico e Sedile;
Problema: Autoveicolo erediterebbe un solo Pneumatico, un solo Sedile contrariamente ai propri requisiti (più di un pneumatico, più di un sedile)

Soluzione: usare l'Aggregazione in modo che una classe autoveicolo abbia dei riferimenti ad uno o più oggetti di tipo Carrozzeria, Pneumatico e Sedile

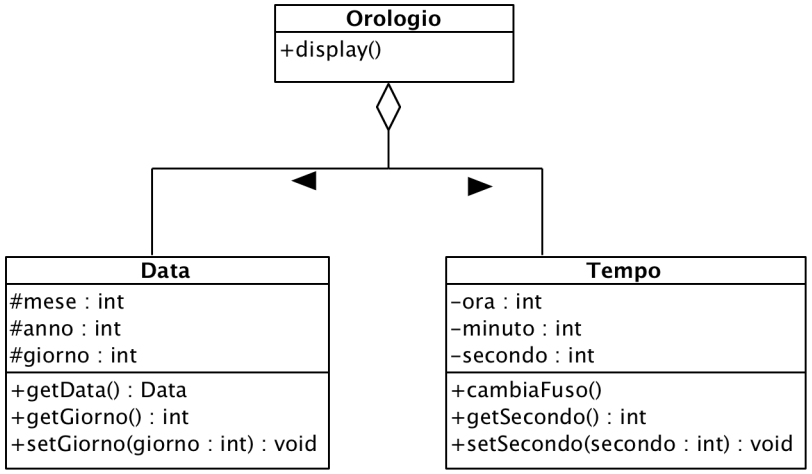
Esempio Orologio: Aggregazione o Ereditarietà?

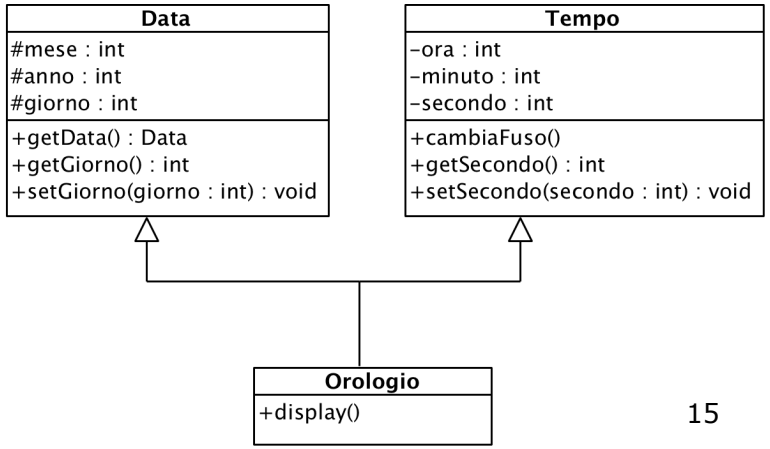
Osservazione: in molti casi la scelta di come associare due classi mediante ereditarietà o aggregazione/composizione non è immediata

Esempio : si supponga di disporre di una classe *Data* e di una classe *Tempo*, che rappresentano e manipolano, rispettivamente, date ed istanti di tempo. Volendo definire una nuova classe *Orologio* per rappresentare sia le date che gli istanti di tempo, si può:

1. derivare *Orologio* da *Data* e da *Tempo*, quando l'ereditarietà multipla è permessa.
2. Comporre *Orologio* con le due classi.

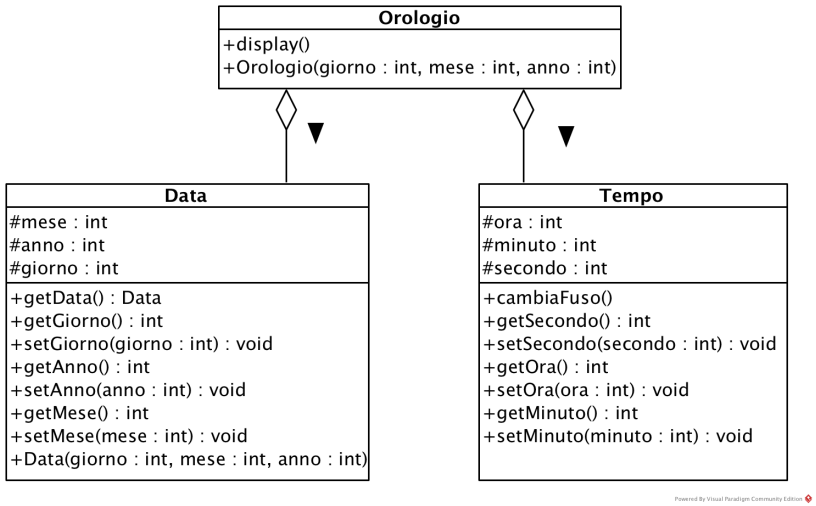
Sono anche possibili soluzioni ibride (ereditarietà da una classe + composizione con l'altra)

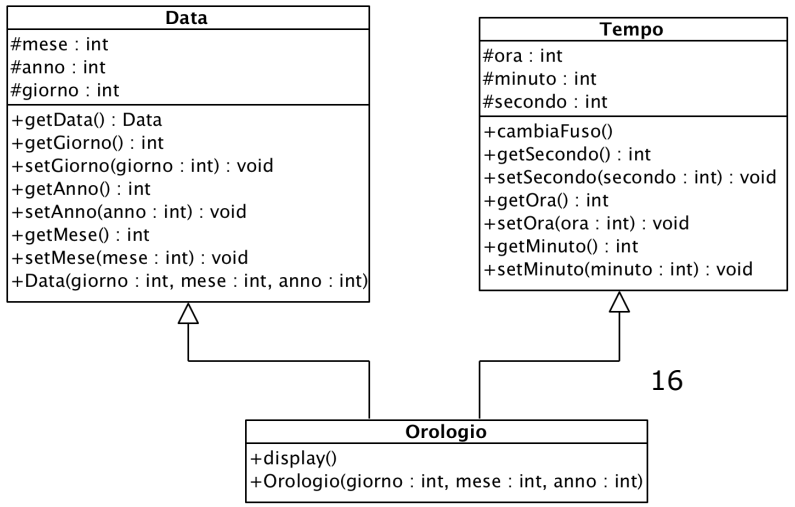




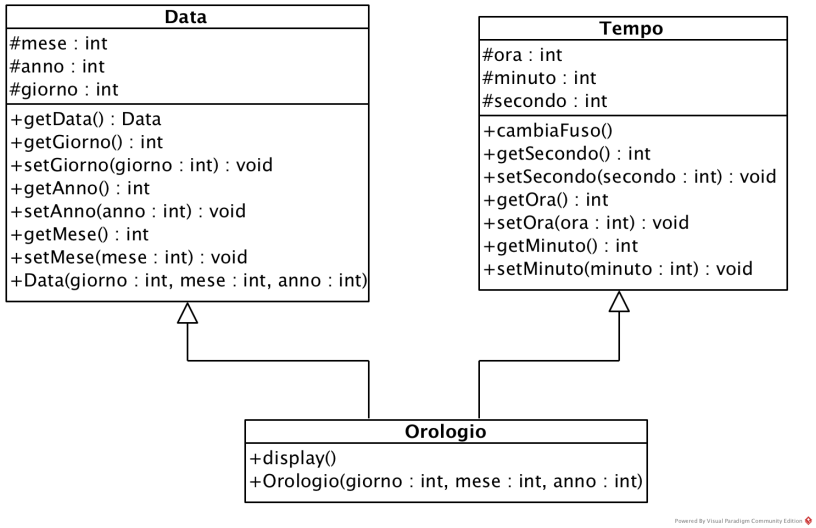
Ereditarietà Vs. Aggregazione: differenze

- Ereditarietà: ogni oggetto della classe Orologio contiene tutti i campi definiti nelle classi Data e Tempo e può accedervi direttamente (se non sono privati)
- Aggregazione: ogni oggetto della classe Orologio contiene due campi di tipo Data e Tempo rispettivamente. Per accedere all'informazione sul giorno occorrerà invocare un metodo (per esempio getData ()) sull'oggetto data istanza di Data
  - qualora Orologio sia dichiarato in un package differente da quello di Data e Tempo
- Polimorfismo di inclusione
  - valido solo se l'ereditarietà corrispondesse a una relazione di generalizzazione (is_a), quindi ogni oggetto di *Orologio* sarebbe utilizzabile come una istanza di *Data* e di *Tempo;*
  - non valido nel caso dell'aggregazione/composizione





Ereditarietà Vs. Aggregazione: scelte di progettazione



Totalmente diverse

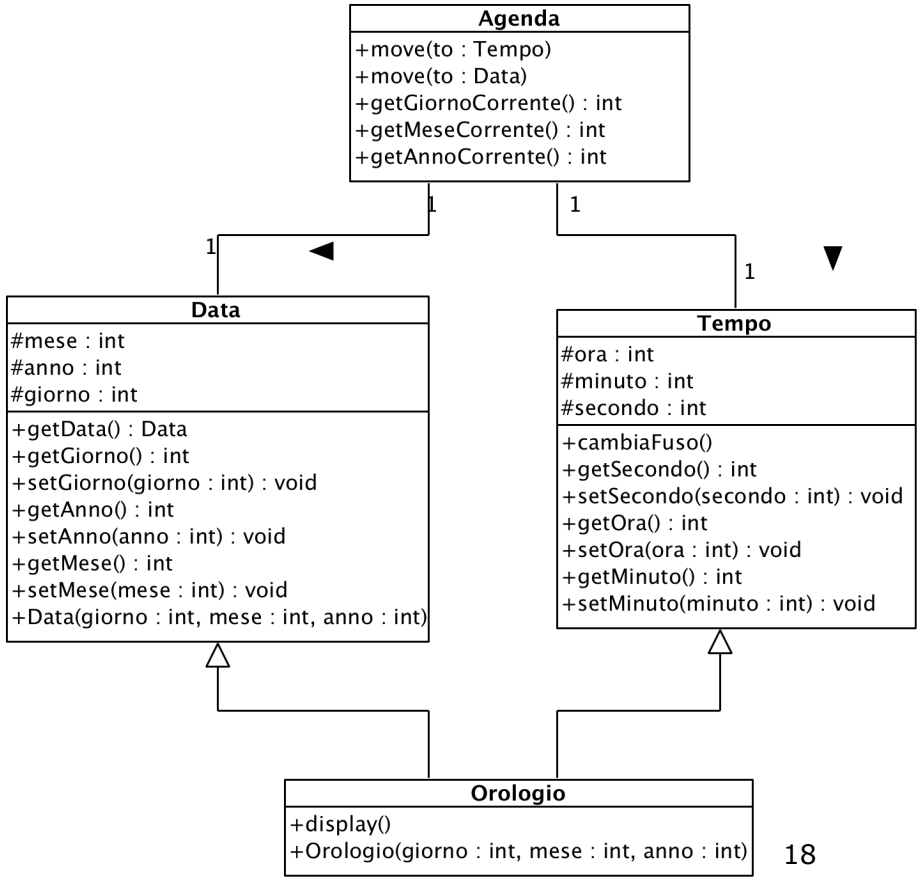
Vantaggio ereditarietà: permette di poter "dimenticare" il fatto che giorno sia definito nella classe Data e di utilizzarlo direttamente come campo di *Orologio* (sempre che sia definito come *protected* e non come *private*).

- Utilità: creazione di gerarchie di classi a più livelli
  - in questo caso non è necessario conoscere a quale livello della gerarchia sia definito un dato campo, ma è possibile utilizzarlo direttamente come se fosse un elemento della classe stessa.

Ereditarietà Vs. Aggregazione: scelte di progettazione

Vantaggio ereditarietà: L'ereditarietà permette di riutilizzare tutti i metodi delle varie classi che operano su oggetti delle classi base (nell'esempio *Data* e *Tempo)*

Esempio: *Agenda* ha due metodi che possono operare anche su istanze di *Orologio*. Il codice dei metodi polimorfi *move()* è riutilizzabile per istanze di *Orologio*.



Quando usare l'Aggregazione rispetto all'Ereditarietà

- (generalmente) quando si vogliono utilizzare i servizi di una classe predefinita ma non esporre la sua interfaccia
- qualora l'ereditarietà di implementazione non dovesse essere permessa da un linguaggio di programmazione

Relazione di Dipendenza

⇨ Definizione

- indica una connessione "semantica" tra due classi in cui una delle due è dipendente dall'altra

⇨ Osservazioni

- una modifica nell'elemento indipendente ha effetti su quello dipendente
- la dipendenza può essere di varia natura, esistono infatti diverse relazioni di dipendenza
- le dipendenze possono esistere tra classi e tra *package* e *package*

Tipi di dipendenza

- USO
  - IL *CLIENT* UTILIZZA ALCUNI DEI SERVIZI RESI DISPONIBILI DAL FORNITORE PER IMPLEMENTARE IL PROPRIO COMPORTAMENTO
- ASTRAZIONE
  - UNA RELAZIONE IN CUI IL FORNITORE È PIÙ ASTRATTO DEL *CLIENT*
    - AD ESEMPIO UNA CLASSE IN UN MODELLO DI ANALISI E LA STESSA CLASSE NEL MODELLO DI PROGETTAZIONE
- PERMESSO
  - IL FORNITORE ASSEGNA AL *CLIENT* "UN QUALCHE TIPO" DI PERMESSO DI ACCESSO AL PROPRIO CONTENUTO

Esempi di dipendenza di tipo *uso* ed *astrazione* saranno proposti nella dispensa SOLID

Classe Astratta

Definizione

- una classe che può non essere completamente specificata
  - una classe è «non completamente specificata» se ha almeno un'operazione per la quale non è stato definito il metodo

Perché le classi astratte? Un esempio

Si supponga di voler creare delle forme geometriche (*Shape*) quali, ad esempio, Cerchio, Rettangolo, Quadrato perché siano visualizzate su un terminale video.

Dall'analisi dei requisiti si ottiene che qualunque forma geometrica

- possiede un'area il cui calcolo, tuttavia, varia da forma geometrica a forma geometrica;
- ha un tipo di tratto con la quale essere resa a video che può assumere uno dei seguenti valori: tratteggiato, continuo, a punti;

e che, inoltre,

- al momento della creazione sia sempre specificato il tipo del tratto;
- Il tipo del tratto possa essere modificato;
- Il tipo del tratto sia un'informazione che una forma geometrica rende disponibile.

Inoltre:

- un Rettangolo è caratterizzato da due dimensioni: lunghezza e larghezza; un rettangolo per poter essere creato deve essere avvalorato rispetto alle proprie dimensioni;
- un Cerchio è caratterizzato da un raggio; un Cerchio per poter essere creato necessita di essere avvalorato rispetto al proprio raggio.

Perché le classi astratte? Un esempio

Considerazioni

- Una forma geometrica, quindi, non sarà di fatto mai creata
- Solo particolari forme geometriche potranno essere create (Rettangolo, Cerchio, …)
- Alcuni operazioni sono comuni a tutte le forme geometriche: tra queste alcune avranno una differente realizzazione in dipendenza della forma geometrica, altre un'unica realizzazione valida per tutte le forme geometriche
- Alcuni attributi ed operazioni di creazioni sono, invece, specifici di ogni forma geometrica

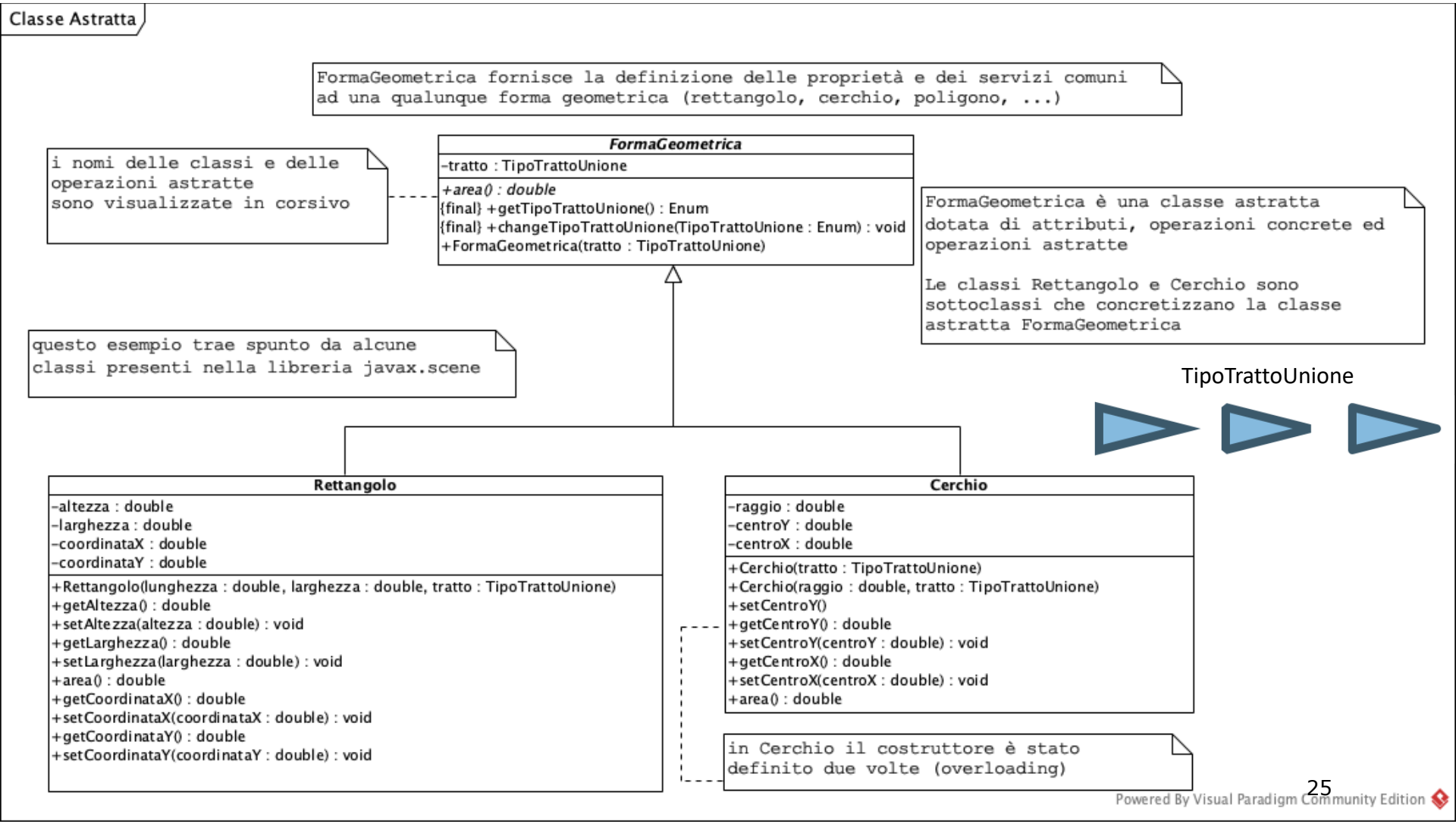
Domanda

- Gli attributi e le operazioni che appartengono alla definizione di una qualunque forma geometrica possono essere definiti separatamente dagli attributi e dalle operazioni specifici delle singole forme geometriche?

Se fosse possibile si otterrebbero i seguenti vantaggi:

1. un'unica manutenzione correttiva, evolutiva e perfettiva delle parti comuni;
2. un Test unico delle parti comuni;
3. un unico modo per il programmatore di riferirsi alle figure geometriche per tutte le parti in comune (polimorfismo di inclusione) con conseguente vantaggio in termini di estendibilità

Classe Forma Geometrica



Pseudo-codice *FormaGeometrica*

```
Classe Astratta FormaGeometrica {
   privato TipoTrattoUnione tratto;

   pubblico TipoTrattoUnione getTratto() {
      restituisci tratto;
   };
   pubblico changeTratto(TipoTrattoUnione tratto){
            puntatoreIstanzaCorrente.tratto = tratto;
   }
   pubblico CreaFormaGeometrica(TipoTrattoUnione tratto) {
            puntatoreIstanzaCorrente.tratto = tratto;
   }
   pubblico astratto TipoReale area();
}
```

Pseudo-codice *Rettangolo*

```
Classe Rettangolo derivada FormaGeometrica {
        privato numeroReale lunghezza, larghezza;

        pubblico CreaRettangolo (TipoTrattoUnione tratto, numeroReale
        lunghezza, numeroReale larghezza)
        {
        // necessario invocare il costruttore esplicito della classe base da cui deriva
        superclasse(tratto);
         puntatoreIstanzaCorrente.lunghezza = lunghezza;
         puntatoreIstanzaCorrente.larghezza= larghezza;
        }
@implementa l'operazione astratta area() per il Rettangolo
        pubblico numeroReale area() {
                restituisci (lunghezza*larghezza)/2;
        }
}
```

Calcolo dell'area di una FormaGeometrica

```
Classe GestioneFormeGeometriche {
        privato costante intero MAX =  19;
        privato  FormaGeometrica forme[MAX];
        privato intero last=0;

        pubblico booleano aggiungiForma(FormaGeometrica forma) {
                booleano risultato = false;
                if (isNotPieno()) {
                        forme[last]= forma; last = last +1;  risultato=true;}
                restituisci risultato;
        }
        privato booleano isNotPieno(){
                restituisci (MAX-last>0)
        }
        pubblico calcolaArea() {
                while (int index=0; index<last; index++)
                        forme[index].area();
        }
}
```

Classe Astratta: proprietà generali

Rappresenta un classe base in una relazione di ereditarietà il cui comportamento non è completamente definito (in teoria)

- non può essere istanziata (di conseguenza si suppone che esista almeno una classe da essa derivata che sia concreta)
- può avere a disposizioni sia attributi (statici e d'istanza) che operazioni (statiche e di istanze)
- Possiede sempre un costruttore che non è mai astratto in quanto non ci sarebbe modo di concretizzarlo in una classe derivata
  - Se un costruttore non è definito Java si serve del costruttore no-args implicito
  - Il costruttore non potrà però essere invocato con *new* (visto che non si possono creare oggetti istanza di questa classe astratta). Verrà perciò invocato univocamente dentro la sottoclasse (come *super()*).
- fattorizza, dichiarandole, operazioni comuni a tutte le sue sottoclassi, pur non definendole

Interfaccia: definizione

Descrizione del comportamento di una popolazione di oggetti senza specificare alcuna implementazione. Un'interfaccia è:

- una collezione di operazioni prive di implementazioni e di attributi di classe con valore costante
- un'entità di programmazione che funge da schema (scheletro, prototipo) di una classe.
- una versione ridotta di una classe astratta, infatti possiede solo metodi pubblici e astratti e, inoltre, non può avere variabili di istanza.

Obbiettivo dell'uso delle interfacce

- Un'interfaccia incapsula le operazioni di una o più classi separandole, dunque, dalla classe che le realizzerà
- L'uso congiunto delle interfacce, della ereditarietà e del polimorfismo permette di realizzare l'astrazione dati

Problema Veicolo

R_1: Un sistema deve tener traccia di differenti tipi di veicoli;

R_2: alcuni veicoli sono già noti prima della realizzazione del sistema; altri lo saranno solo dopo il primo rilascio del sistema.

R_3: ogni veicolo, indipendentemente dal tipo, permetterà sempre di

- *avviare il Motore;*
- *spegnere il Motore;*
- *accelerare;*
- *frenare.*

R_4: ogni veicolo è caratterizzato da:

- marca, anno di produzione, numero di marca, velocità

Esempio Veicolo: Interfaccia

Per un Veicolo, dunque, a prescindere da come sarà realizzato e dal tipo di Veicolo, si potrebbe pensare di definire un contratto costituito dalle operazioni che esso renderà possibile:

- *avviaMotore*, *spegniMotore*, *accelera*, *frena*, …

Considerazioni

- Il contratto di Veicolo, essendo privo di realizzazioni, non sarà di fatto mai creato
- Solo particolari tipi di Veicolo potranno essere creati (AutoVeicolo, AutoCarro, …)
- Le operazioni del contratto sono comuni a tutte i Veicoli; le realizzazioni di tali operazioni dipenderanno dal tipo di Veicolo;

Vantaggio

- un unico modo per il programmatore di riferirsi ai tipi di Veicoli per tutte le operazioni in comune (polimorfismo di inclusione) con conseguente vantaggio in termini di estendibilità

Pseudo-codice: Interfaccia *Veicolo*

```
pubblica Interfaccia Veicolo {
  pubblico booleano avviaMotore();
  pubblico booleano spegniMotore();
  pubblico accelera();
  pubblico frena();
}
```

Capsula che separa le operazioni dalle realizzazioni e dalle classi in cui saranno implementate tali realizzazioni

Pseudo-codice: classe concreta *Autoveicolo*

```
Classe Autoveicolo realizzaInterfaccia Veicolo {
    privato intero annoProduzione, numeroMarce;
    privato Stringa marca;
    privato numeroReale velocita;

   pubblico CreaAutoveicolo(Stringa marca, intero annoProduzione, intero numeroMarce,
numeroReale velocita) { // Todo something }
    @daImplementare
    pubblico booleano avviaMotore() { // Todo something}
    @daImplementare
    pubblico booleano spegniMotore() { // Todo something}
    @daImplementare
    pubblico accelera() { // Todo something}
    @daImplementare
    pubblico frena() { // Todo something}
}
```


La relazione di realizzazione di un'interfaccia servendosi della ereditarietà rende possibile definire una classe concreta a partire da un'interfaccia

Test dell'acceleratore dei Veicoli

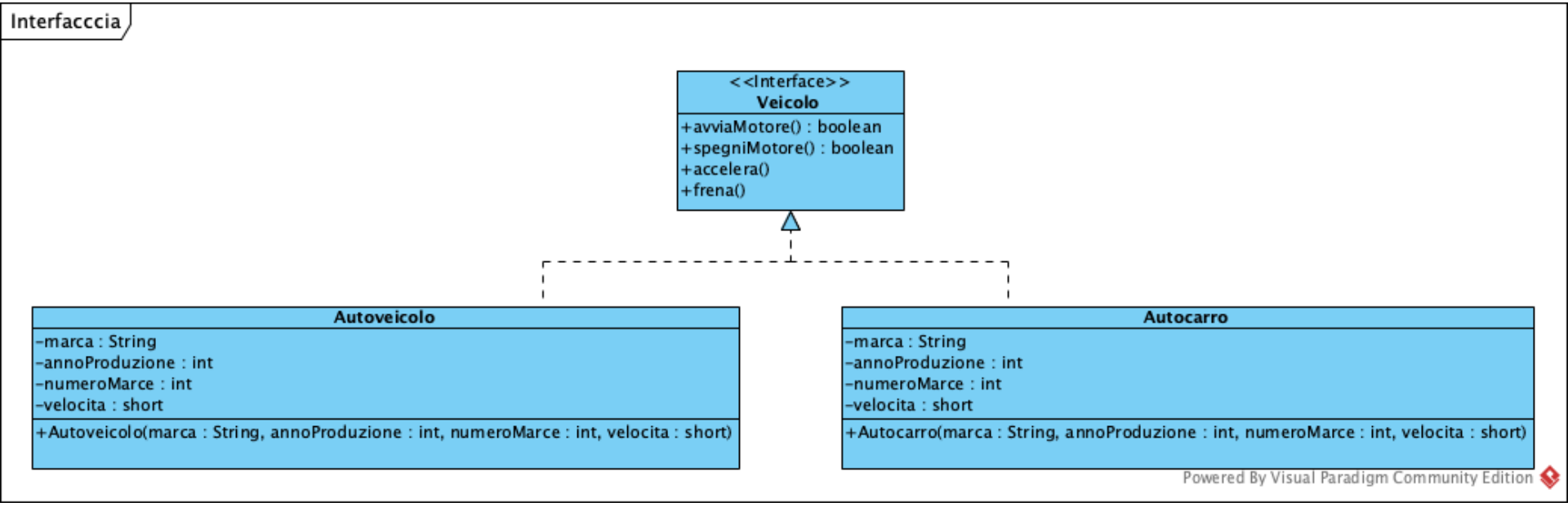
```
Classe GestioneTipi Autoveicoli {
        privato costante intero MAX =  5;
        // all'interfaccia Veicolo corrispondono differenti implementazioni
        privato  Veicolo veicoli[MAX];
        privato intero last=0;

        pubblico booleano aggiungiVeicolo(Veicolo veicolo) {
                booleano risultato = false;
                if (isNotPieno()) {
                        forme[last]= veicolo; last = last +1;  risultato=true;}
                restituisci risultato;
        }
        privato booleano isNotPieno(){  restituisci (MAX-last>0)}
        pubblico testAccelerazione() {
                while (int index=0; index<last; index++)
                        veicoli[index]. accelera();
        }
}
```


Il polimorfismo renderà possible accedere a run-time ad una classe concreta mediante l'interfaccia che essa realizza

Esempio Autoveicolo: modellazione interfaccia e classi concrete

- Consente al programmatore di essere più astratto quando si fa riferimento ad oggetti di tipo Veicolo (compile-time);
- Un'interfaccia richiede al programmatore di creare funzioni specifiche che ci si aspetta in una classe di implementazione quando implementa un'interfaccia.
  - quando viene invocata la funzione veicolo.avviaMotore(), viene effettivamente utilizzata la funzione corretta associata all'oggetto reale. Questo si verifica a "run-time".
  - Autoveicolo ed Autocarro implementeranno ognuna delle operazioni definite in Veicolo;



Proprietà di una Interfaccia

- può avere un qualsiasi numero di operazioni (anche nessuna al limite) per nessuna delle quali è fornita la realizzazione;
- può avere un numero qualunque di attributi che siano tutti di classe, immutabili ed obbligatoriamente inizializzati (trattasi di costanti);
- può avere un qualunque numero di classi che la implementino;
- Il modificatore di accesso di una interfaccia può essere *public* o *package (mai privato o protected);*
- per le costanti e le operazioni l'unico modificatore di accesso consentito è *public*;
- non è mai istanziabile;
- non permette la definizione di costruttori;

Raffinamento della soluzione Veicolo

- La modellazione proposta per Veicolo potrebbe essere migliorata rispetto alla manutenibilità ed alla estendibilità?

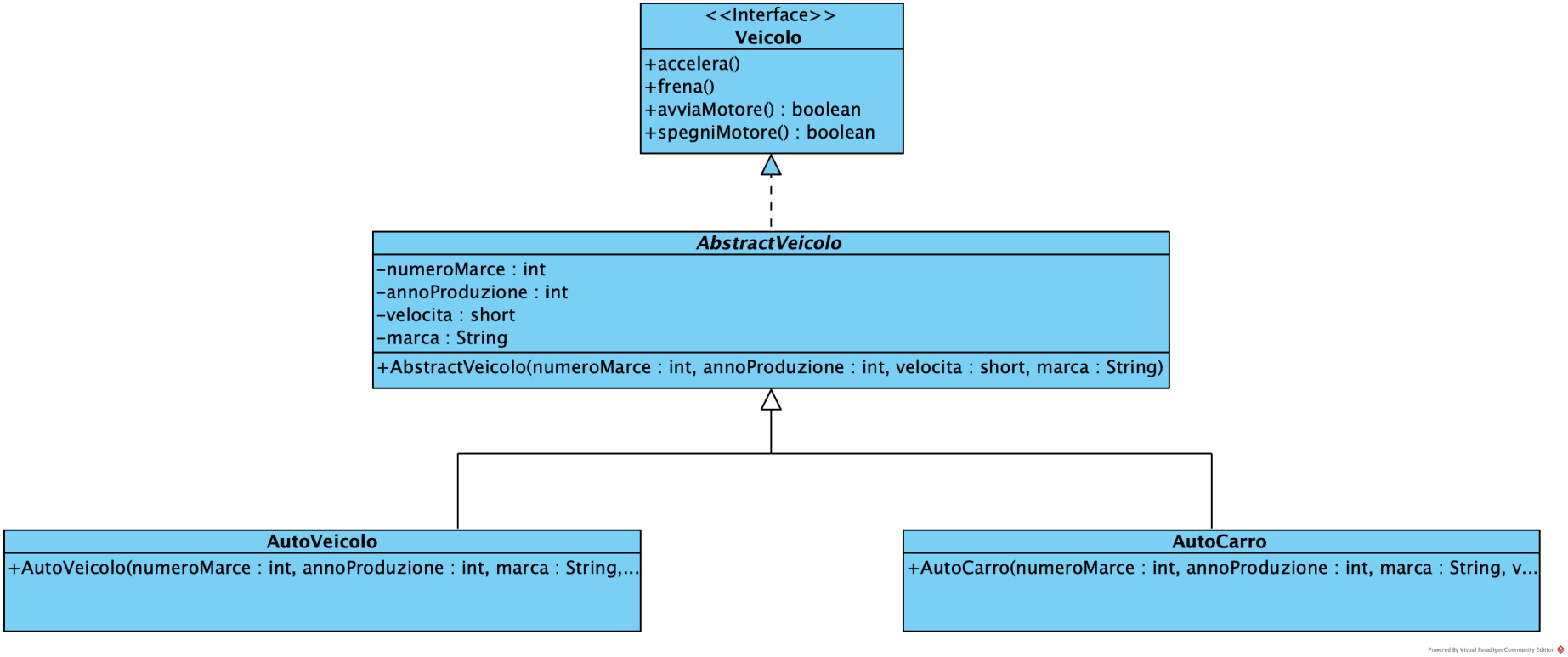
  Idea: introdurre un'ulteriore astrazione che fattorizzi gli elementi comuni non già fattorizzati come ad esempio:

  - attributi di istanza ed operatore di creazione

  Soluzione: definizione di una classe astratta, *AbstractVeicolo*, che gerarchicamente si collochi tra l'interfaccia (*Veicolo*), da un lato, e le classi concrete (*Autocarro, Autoveicolo*), dall'altro.

  - uso congiunto di Interfacce, classi astratte ed ereditarietà

Raffinamento della soluzione Veicolo: modellazione UML



Pseudo-codice Abstract*Veicolo*

Classe Astratta AbstractVeicolo *realizzaInterfaccia* Veicolo {

privato intero annoProduzione, numeroMarce;

privato Stringa marca;

privato numeroReale velocita;

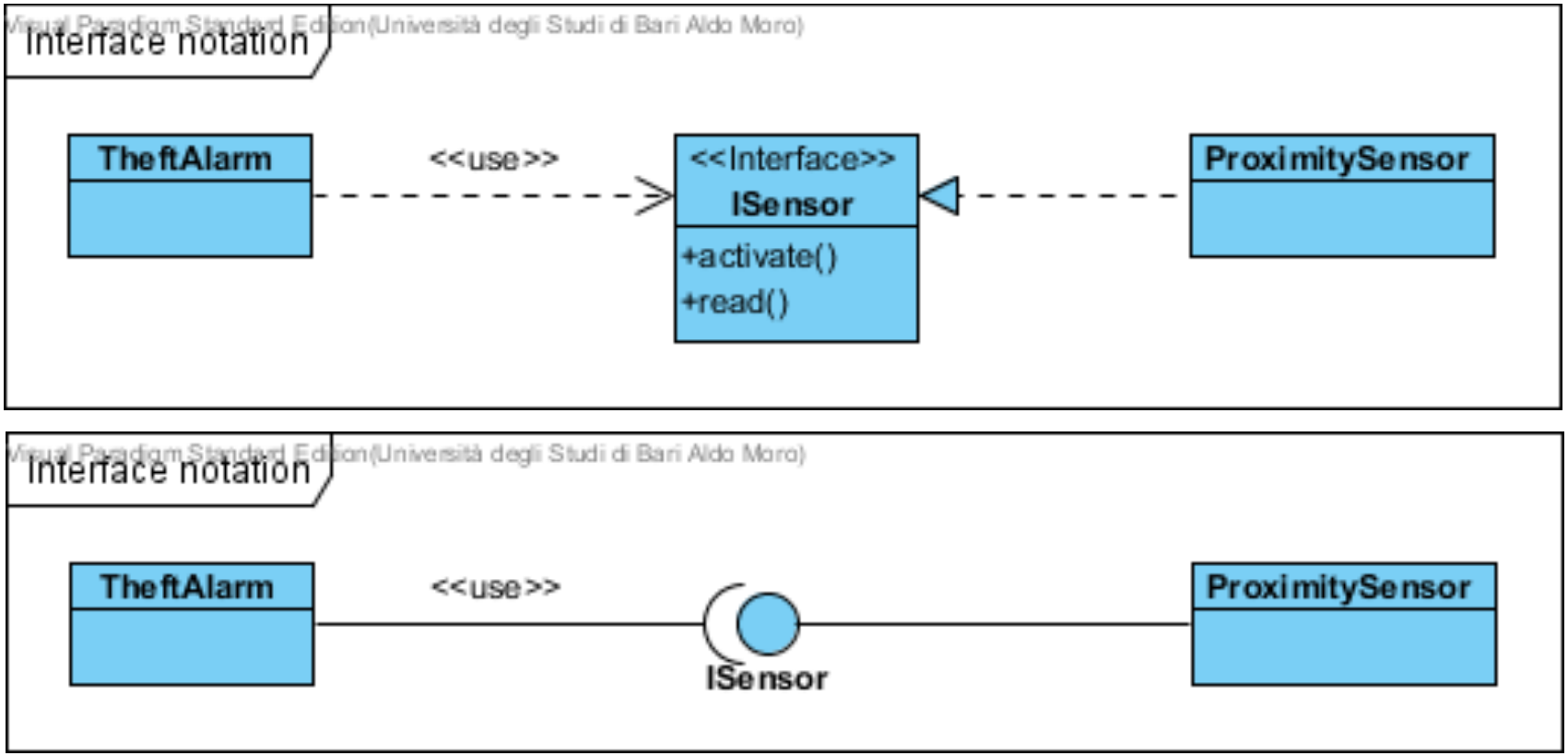
pubblico AbstractVeicolo(*Stringa* marca, *intero* annoProduzione, *intero* numeroMarce, *numeroReale* velocita) { // Todo something }

}

Pseudo-codice *Autoveicolo*

```
Classe Autoveicolo derivaDa AbstractVeicolo {
  pubblico CreaAutoveicolo(Stringa marca, intero annoProduzione, intero numeroMarce,
numeroReale velocita)
  {
        // necessario invocare il costruttore esplicito della classe base da cui deriva
        superclasse(marca,annoProduzione, numeroMarce,velocita);
  }
  @daImplementare
  pubblico booleano avviaMotore() { // Todo something}
  @daImplementare
  pubblico booleano spegniMotore() { // Todo something}
  @daImplementare
  pubblico accelera() { // Todo something}
  @daImplementare
  pubblico frena() { // Todo something}
}
```

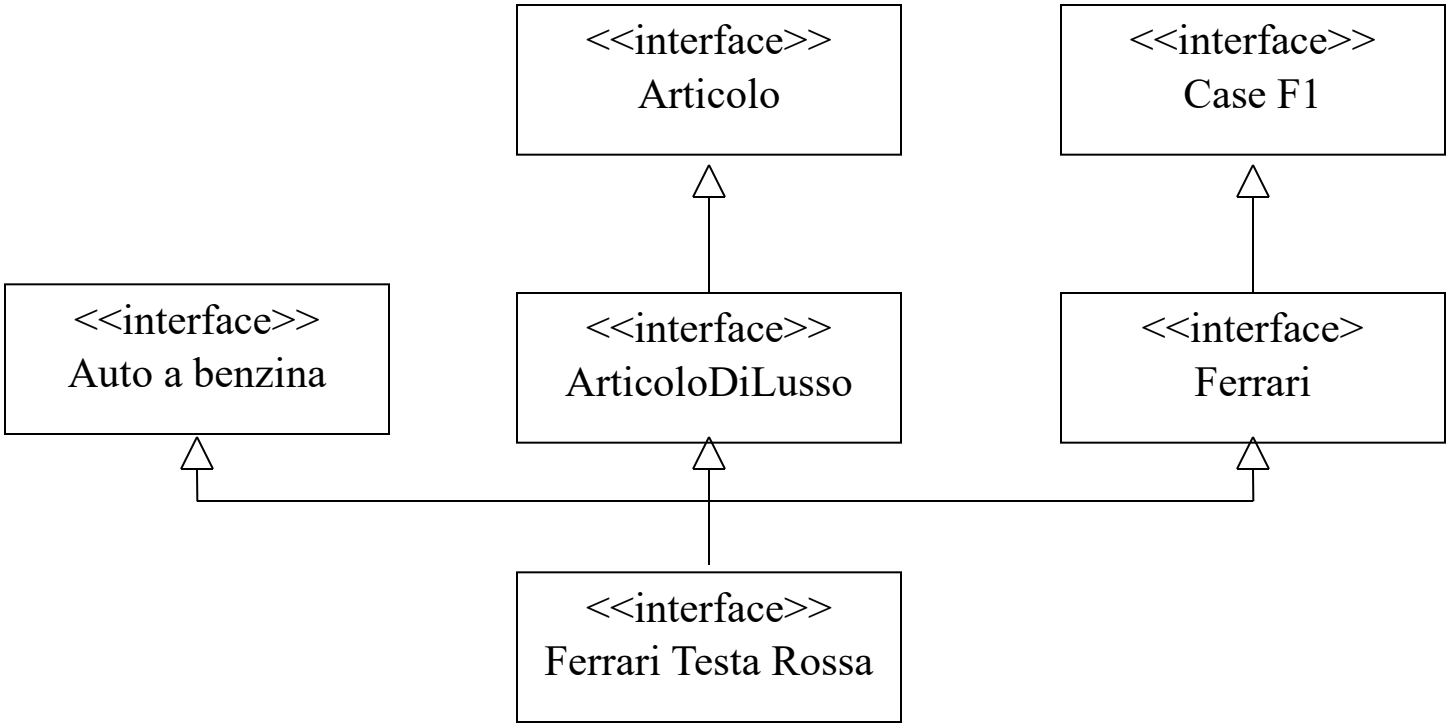
Interfaccia: notazioni UML



- *ProximitySensor* realizza l'interfaccia *ISensor*
- *TheftAlarm* usa l'interfaccia *ISensor* per accedere ad un'istanza di *ProximitySensor*
- *ISensor* funge da capsula (barriera) tra *ProximitySensor* e le classi che hanno bisogno di interagire con istanze di *ProximitySensor*

Ereditarietà multipla tra interfacce

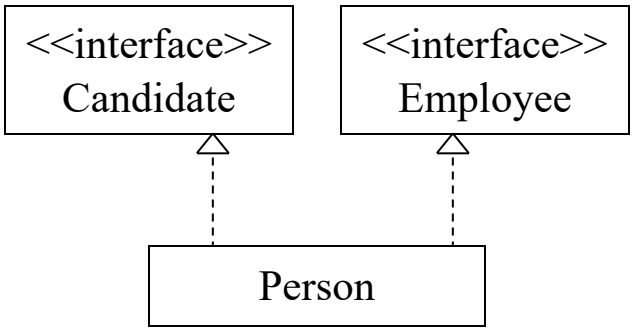
- Un interfaccia può ereditare (estendere in Java) una o più interfacce.
- la relazione di ereditarietà è naturalmente una relazione di generalizzazione "is_a" (non ci sono implementazioni)



Ereditarietà multipla tra interfacce e classi concrete

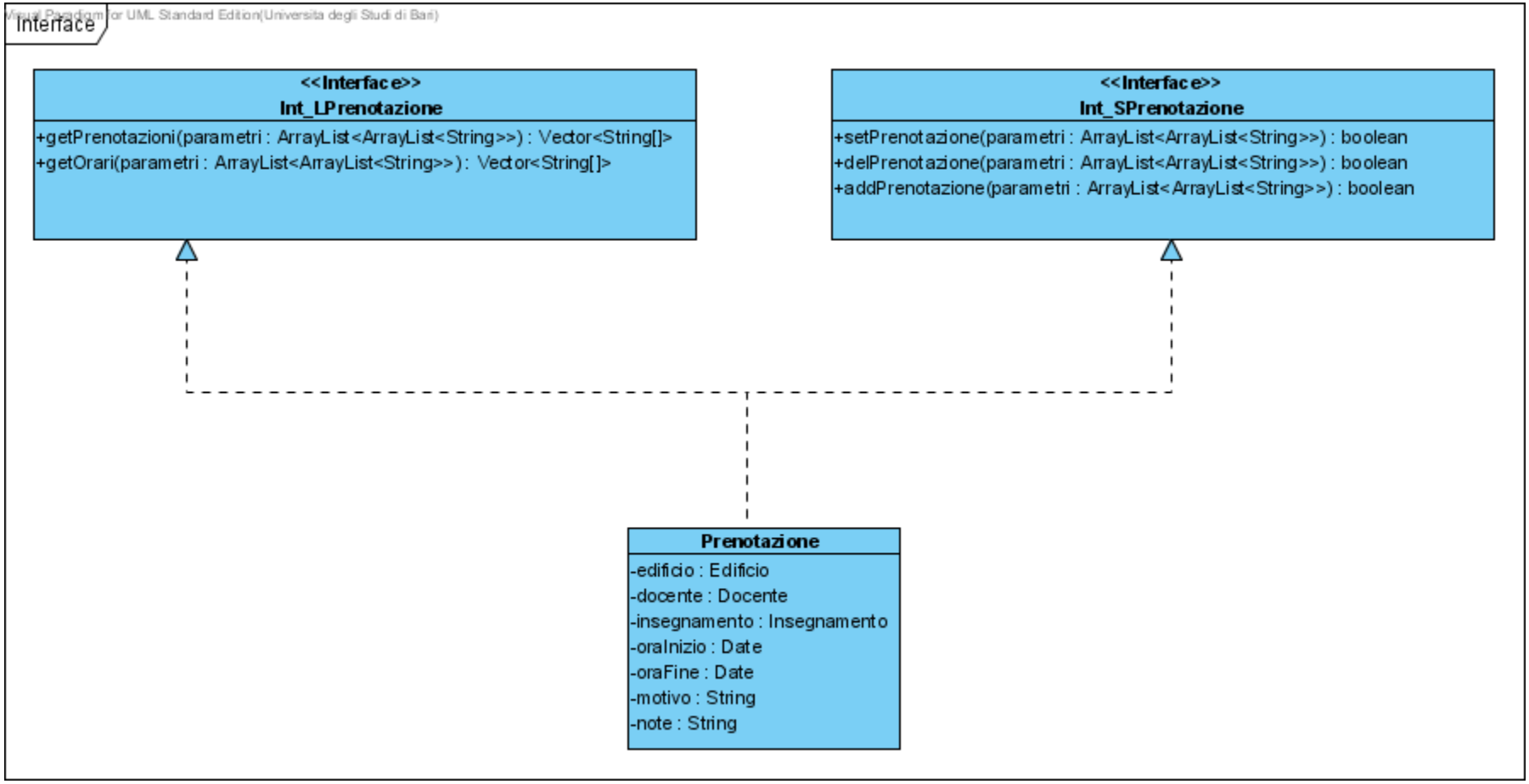
Poiché non possono sorgere problemi di conflitto di realizzazione, è permesso a una classe di realizzare più interfacce, per di più non correlate da una relazione di generalizzazione.

Questa modellazione risolve il problema dell'impossibilità di avere oggetti di più di un tipo



In questo modo tutto il codice scritto per Candidate e Employee può essere utilizzato su oggetti di classe Person.

Ereditarietà multipla tra interfacce e classi concrete



Una classe può implementare più di un'interfaccia; ogni singola interfaccia che realizza, di fatto, costituisce un contratto con il quale comunicare

Pseudo-codice *Prenotazione*

```
Classe Prenotazione realizzaInterfaccia Int_LPrenotazione, Int_Sprenotazione, {
    // dichiazione degli attributi di istanza di Prenotazione

// nel costruttore, per brevità, sono omessi i parametri corrispondenti agli attributi di istanza da inizializzare
    pubblico CreaPrenotazione(….) { // Todo something }

    @daImplementare
    pubblico booleano getPrenotazioni(ListaParametri parametri) { // Todo something}
    @daImplementare
    pubblico booleano getOrari(ListaParametri parametri) { // Todo something}
    @daImplementare
    pubblico booleano setPrenotazioni(ListaParametri parametri) { // Todo something}
    @daImplementare
    pubblico booleano delPrenotazioni(ListaParametri parametri) { // Todo something}
    @daImplementare
    pubblico booleano addPrenotazioni(ListaParametri parametri) { // Todo something}
}
```

Pseudo-codice *Prenotazione*

```
Classe Cliente GestionePrenotazione {
 //…
     public doSomethingPrenotazione() {
          Lprenotazione  prenotLettura instantiationOf Prenotazione;
          prenotLettura.getPrenotazione(listaParametriFiltro);
          // illegale; errore a compile-time
          prenotLettura.setPrenotazione(listaParametriFiltro);

          Sprenotazione  prenotazione instantiationOf Prenotazione;
          prenotazione.setPrenotazione(listaParametriFiltro);
          // illegale; errore a compile-time
          prenotazione.getPrenotazione(listaParametriFiltro);
          }
}
```

Problema: se una parte del sistema volesse sia poter avere l'elenco delle prenotazioni che effettuare le prenotazioni?

Ereditarietà multipla tra interfacce

```
pubblica Interfaccia AllPrenotazione derivaDa Lprenotazione, Sprenotazione {
        // nothing
}
Classe Prenotazione realizzaInterfaccia Int_LPrenotazione, Int_Sprenotazione, AllPrenotazione{
        // omissis
}

Classe ClienteGestionePrenotazione {
 //…
    public doSomethingPrenotazione() {
        Lprenotazione prenotazioneL instantiationOf Prenotazione();
        Sprenotazione prenotazioneS instantiationOf  Prenotazione();
        Allprenotazione prenotazioneAll instantiationOf Prenotazione();
        Allprenotazione prenotazioneAll instantiationOf Prenotazione(«lezione»);

    }
}
```

Problema: Le interfacce multiple riducono la dipendenza tra la parte del sistema che richiede delle operazioni e la parte del sistema che le offre

Interfacce: considerazioni sulla ereditarietà multipla

- l'ereditarietà multipla su interfacce non pone problemi di conflitto di realizzazione poiché non si considerano le implementazioni delle operazioni
  - questo distingue le classi astratte (per le quali l'ereditarietà multipla può porre dei problemi riguardo ai metodi implementati) dalle interfacce.

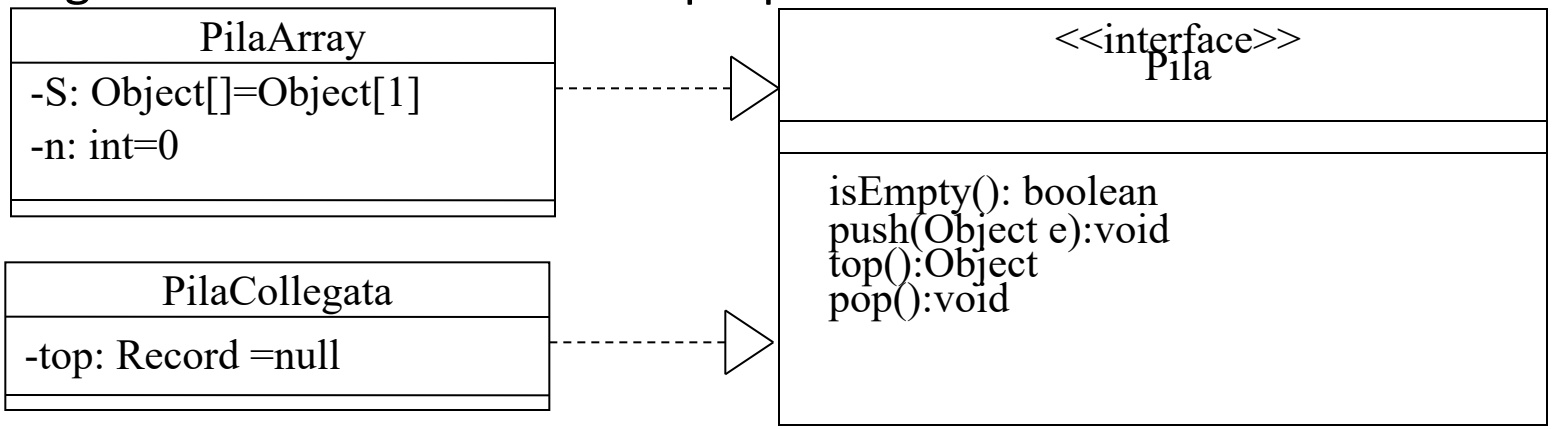
Java e l'ereditarietà

- Java supporta solo l'ereditarietà singola sulle classi, astratte o concrete che siano (in quanto specificano anche le implementazioni) mentre supporta l'ereditarietà multipla sulle interfacce

Realizzazioni multiple di una interfaccia

Più classi possono implementare la stessa interfaccia

- Esempi: dati astratti quali pile, code, liste, alberi, grafi hanno ognuna un'unica interfaccia per le differenti realizzazioni
  - le implementazioni variano per efficienza
  - ogni struttura di dati ha una propria interfaccia



Il codice applicativo che intende fare uso di una pila dichiarerà delle variabili di tipo *Pila*, svincolandosi dalla specifica realizzazione. Questa sarà specificata solo al momento della inizializzazione della variabile. Si garantisce così una forte invarianza ai cambiamenti delle realizzazioni di una pila.

Interfacce Vs Classi Astratte

Un'interfaccia è diversa da una classe astratta sotto due aspetti generali

- Teorico
  - una classe combina progetto e implementazione degli oggetti (ciò vale anche per una classe astratta, la cui implementazione è tuttavia solo parziale); l'interfaccia è puro progetto
- Pratico
  - una classe può essere estesa, garantendo così l'ereditarietà singola di Java. Grazie alle interfacce si può invece avere una sorta di ereditarietà multipla

Classe Leaf

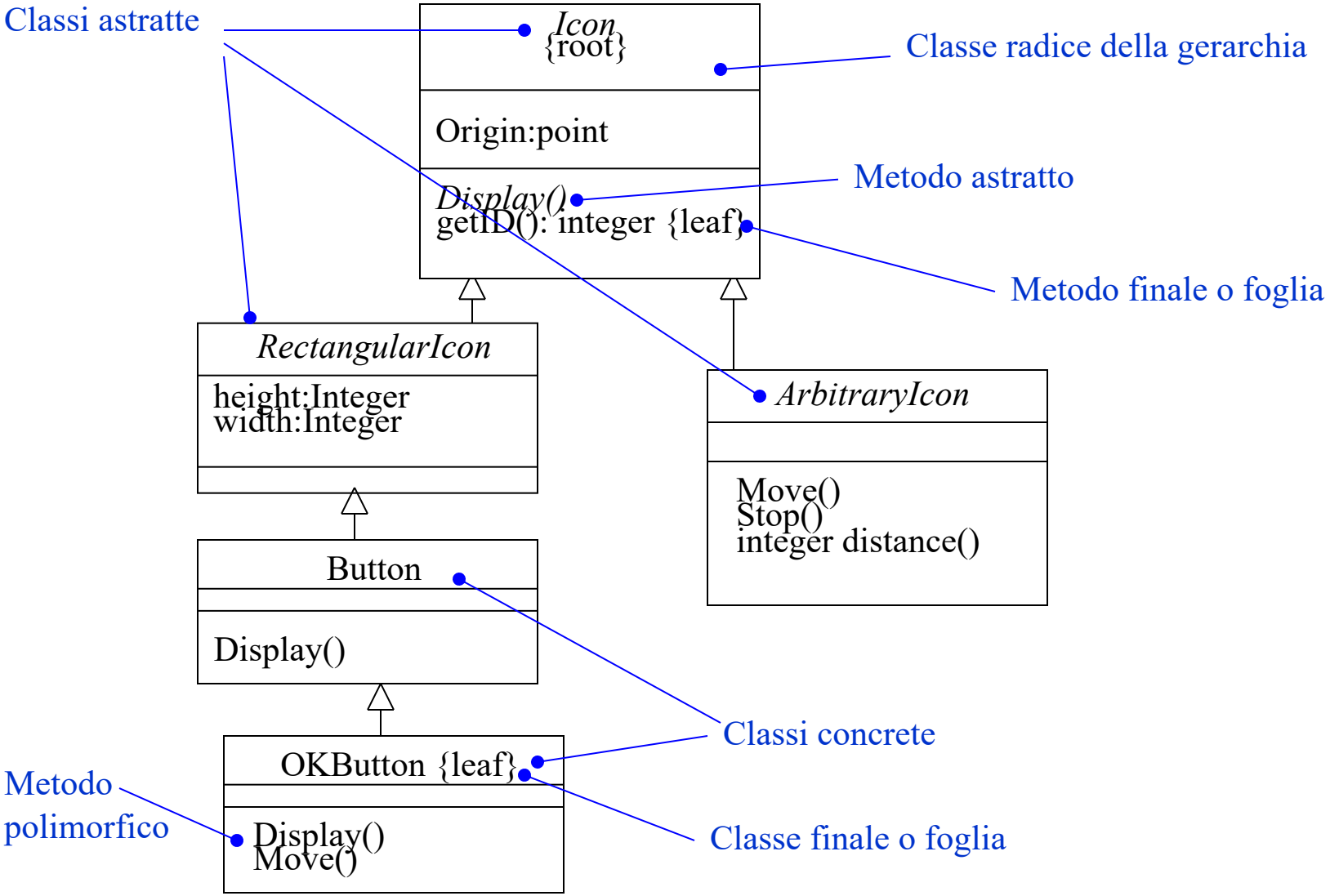
Definizione

- una classe è detta *leaf*, quando non può essere ulteriormente specializzata e, quindi, non può essere modificata

Quando dichiarare una classe come leaf?

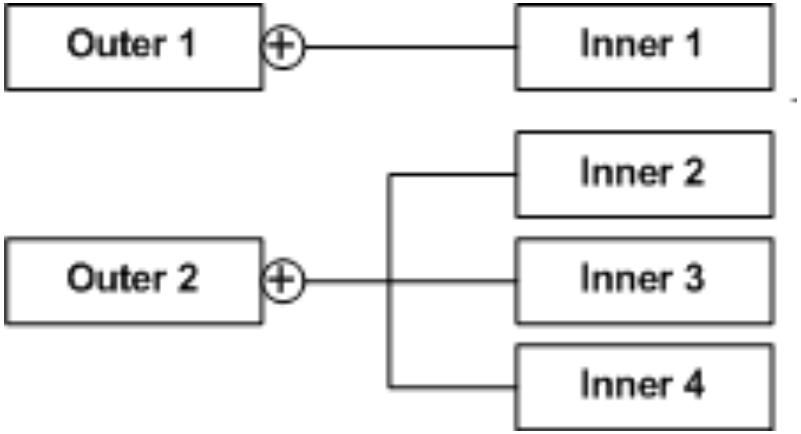
1. Quando il comportamento della classe dev'essere ben stabilito per ragioni di affidabilità
2. Quando si hanno particolare esigenze di ottimizzazione del codice.
   - la dichiarazione di una classe foglia permette anche la generazione di codice ottimizzato in quanto facilita l'espansione in linea del codice (impossibile nel caso di metodi sovrascrivibili nelle sottoclassi).

Classe Leaf: esempio



Classi interne: notazione UML

UML non ha una notazione standard per le classi interne.
Allen Holub suggerisce la seguente notazione:



http://www.holub.com/goodies/uml/index.html

Esercizio: Forme geometriche 1/2

Un sistema deve trattare un insieme di forme geometriche. Tali forme geometriche sono: cerchio, quadrato, segmento, rettangolo.

- Ogni forma geometrica è composta da un insieme continuo di punti e da un colore con il quale questi punti sono visualizzati;
- Una forma geometrica è caratterizzata dalle seguenti operazioni:
  - Selezione della forma geometrica
  - Cancellazione della forma geometrica
  - Spostamento sul piano cartesiano della forma geometrica in una posizione random
  - restituzione del colore della forma geometrica
- Ogni punto è specificato da una coppia di coordinate cartesiane ovverosia coordinataX, coordinataY e dalle rispettive operazioni che ne permettono di ottenere il valore

Esercizio: Forme geometriche 2/2

Le forme di tipo *Quadrato* sono caratterizzate da un *lato* e da operazioni quali:

- Calcolo dell'area
- Restituzione del lato del quadrato

Le forme di tipo *Cerchio* sono caratterizzate da un *raggio* e da operazioni quali:

- Calcolo dell'area
- Restituzione del raggio

Le forme di tipo *Segmento* sono caratterizzate da una coppia di punti e da operazioni quali:

- Traccia una retta tra la coppia di punti
- Calcolo della lunghezza del segmento

Le forme di tipo *Rettangolo* sono caratterizzate da una base e da una altezza e da operazioni quali:

- calcolo dell'area;
- restituzione della lunghezza; restituzione della larghezza

Bibliografia …

1. m.fowler, “uml distilled”, terza edizione, pearson addison wesley, 2004
2. l.a.maciaszek, “sviluppo di sistemi informativi con uml”, pearson addison wesley, 2002
3. http://www.uml.org/ (specifiche omg)
4. http://www.analisi-disegno.com/uml/uml.htm
5. l.vetti tagliati, “uml e ingegneria del software”, scaricabile gratuitamente, previa registrazione, a partire dall’indirizzo: http://www.mokabyte.it/umlbook/index.htm
6. association for computing machinery, oopsla'87 conference proceedings, special issue of sigplan notices, vol. 22, no. 12, dicembre 1987
7. i.jacobson, "object-oriented development in an industrial environment", oopsla'87 conference proceedings, special issue of sigplan notices, vol. 22, no. 12, dicembre 1987, pp. 183-191
8. d.coleman, p.arnold, s.bodoff, c.dollin, h.gilchrist, f.hayes, and p.jeremaes, "object-oriented development: the fusion method", prentice hall, englewood cliffs, new jersey, 1994
9. g.booch, "object-oriented analysis and design - with applications", second edition, benjamin/cummings, menlo park, california, 1994

… Bibliografia …

10. s.cook, j.daniels, "designing object systems - object oriented modelling with syntropy", prentice hall, englewood cliffs, new jersey, 1994
11. g.booch,j.rumbaugh, "unified method: user guide, version 0.8", rational software corporation, santa clara, california, 1995
12. i.jacobson, m.ericsson, a.jacobson , “the object advantage : business process reengineering with object technology ”, addison-wesley object technology series, 1995
13. i.jacobson, m.christerson, p.jonsson, and g.övergaard, "object-oriented software engineering: a use case driven approach", addison-wesley, reading, massachusetts, 1992
14. r.lee, w.m.tepfenhart “uml and c++: a practical guide to object-oriented development”, prentice hall, 1997
15. g.booch, i.jacobson and j. rumbaugh, “the unified modeling language reference manual", addison wesley, 1999
16. meilir page-jones, “progettazione a oggetti con uml”, apogeo, 2002