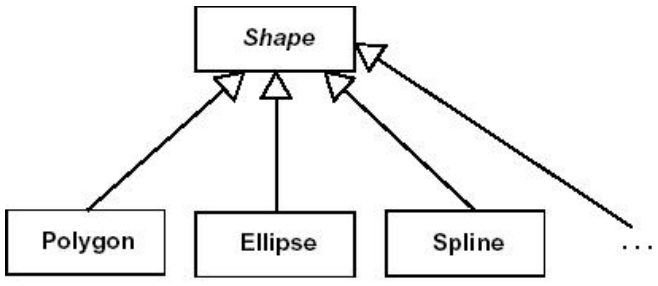
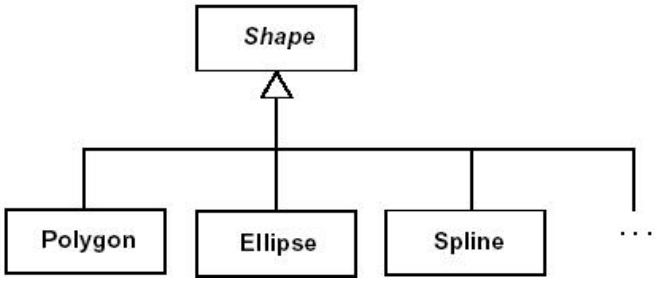
progettare buone gerarchie

generalizzazione

- permette di definire dettagli del modello a vari livelli di astrazione



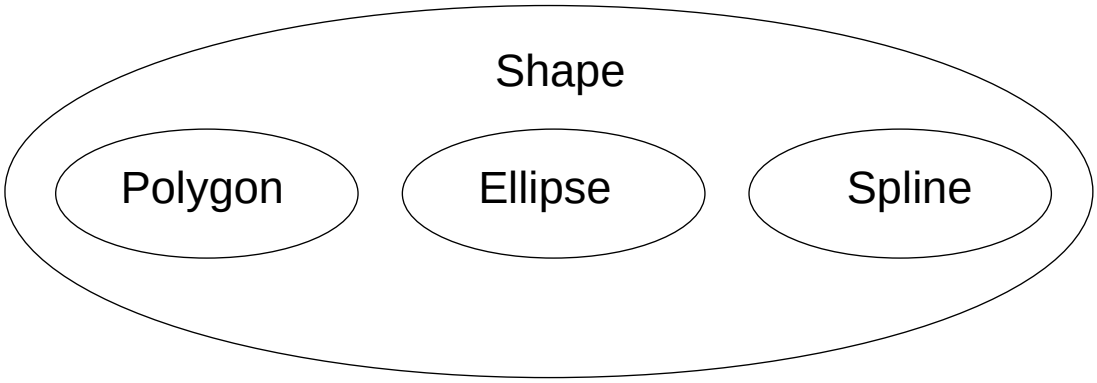
Separate Target Style



Shared Target Style

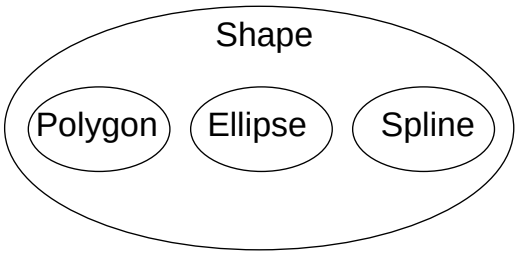
generalizzazione

- le istanze delle classi più specifiche sono istanze anche delle classi generali
- ecco perché tale relazione viene detta anche *is-a* (*è-un*)
- in termini di diagramma di Venn:

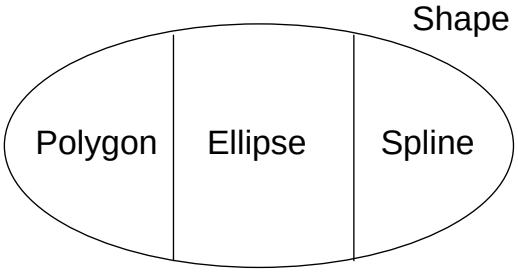


generalizzazioni complete e incomplete

- incomplete

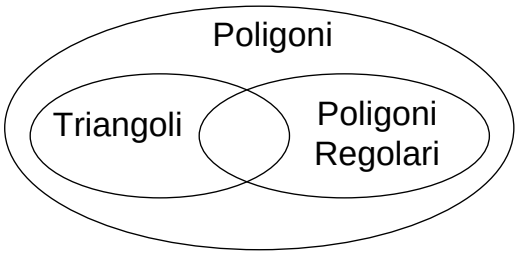


- complete

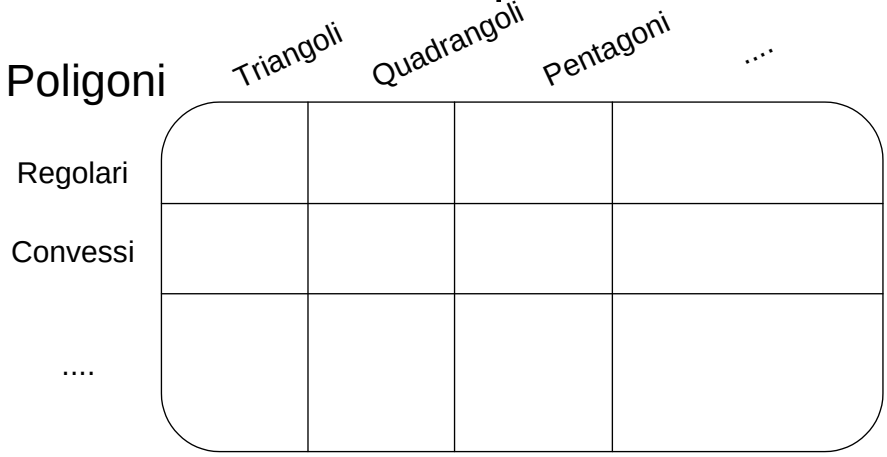


sottoclassi sovrapposte

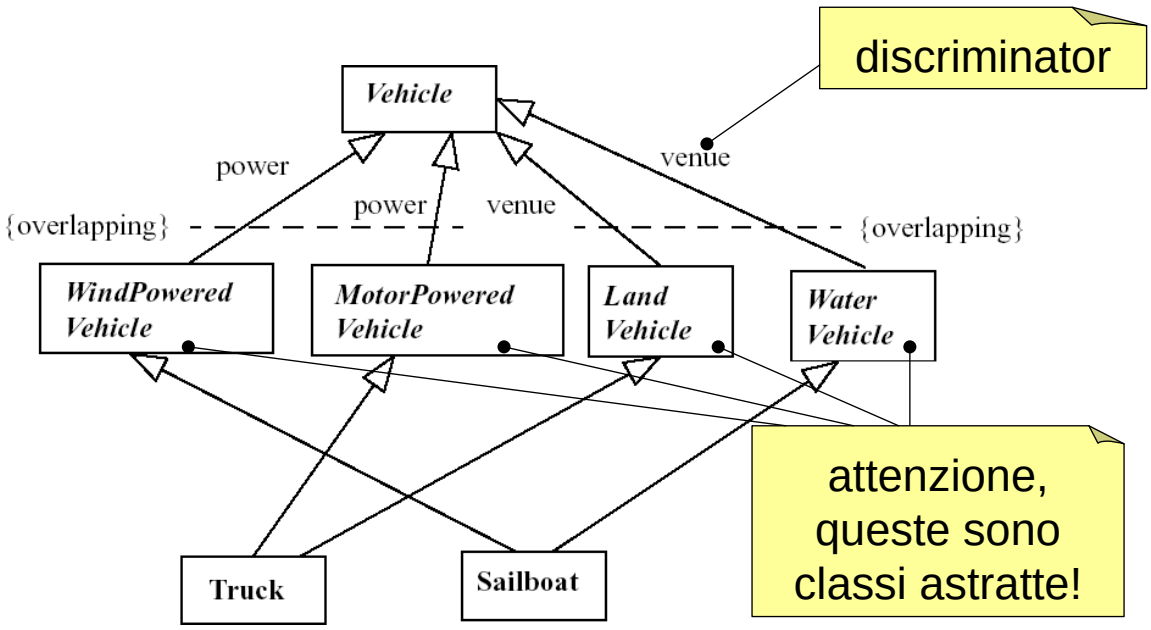
- esempio di classi sovrapposte (overlapping)



- più classificazioni indipendenti



generalizzazioni: notazione UML

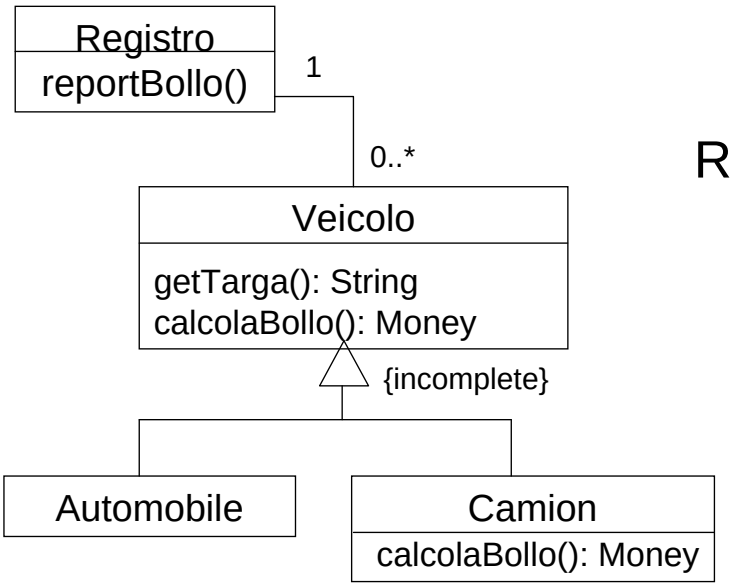


- se nulla è detto le classi si considerano disgiunte
- esercizio: disegna il diagramma di Venn della gerarchia UML mostrata sopra

polimorfismo

- è il più importante aspetto implementativo del concetto di generalizzazione
- istanze di classi derivate possono essere usate come istanze di classi più generali
  - supportato da tutti i linguaggi object oriented
  - alle volte con delle limitazioni
    - es. in C++ solo per oggetti puntati
- estremamente importante per la buona strutturazione dei modelli/progetti/sistemi

polimorfismo: esempio

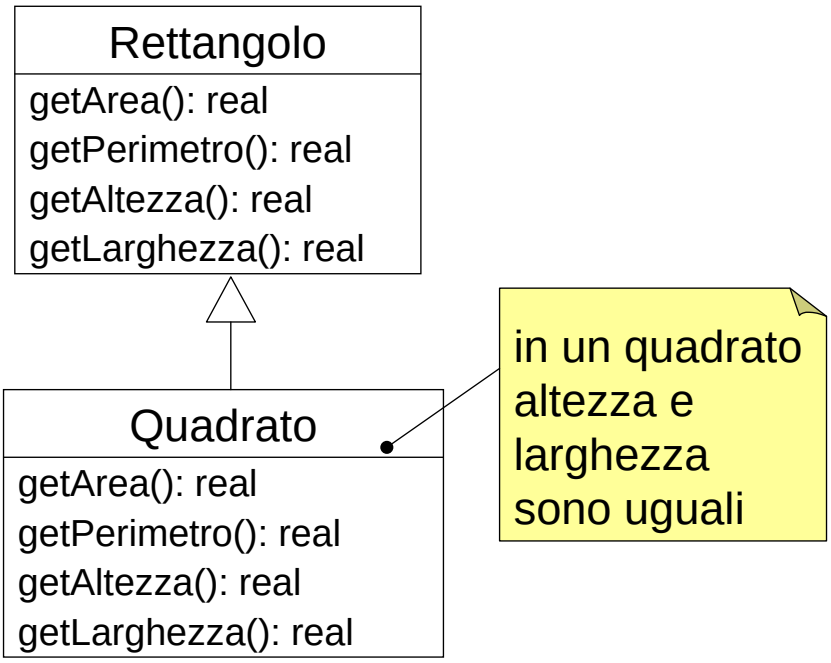



```
Registro::reportBollo()
    ListOfVeicolo g;
    ....// g inizializzata
    forall v elements of g
        print    v.getTarga(),
                 v.calcolaBollo()
```

- quali parti del modello (o del codice) si devono modificare se...
  - ...si aggiunge un tipo di veicolo?
  - ...si elimina un tipo di veicolo?
  - ...un tipo di veicolo cambia caratteristiche?

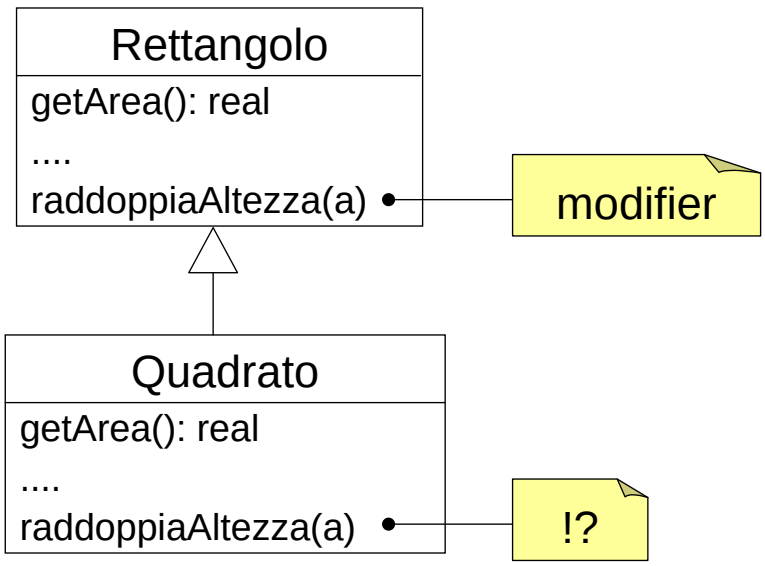
generalizzazione come relazione “è un”

- un quadrato **è un** tipo di rettangolo, usereste la generalizzazione per modellare il fenomeno?



... fate attenzione

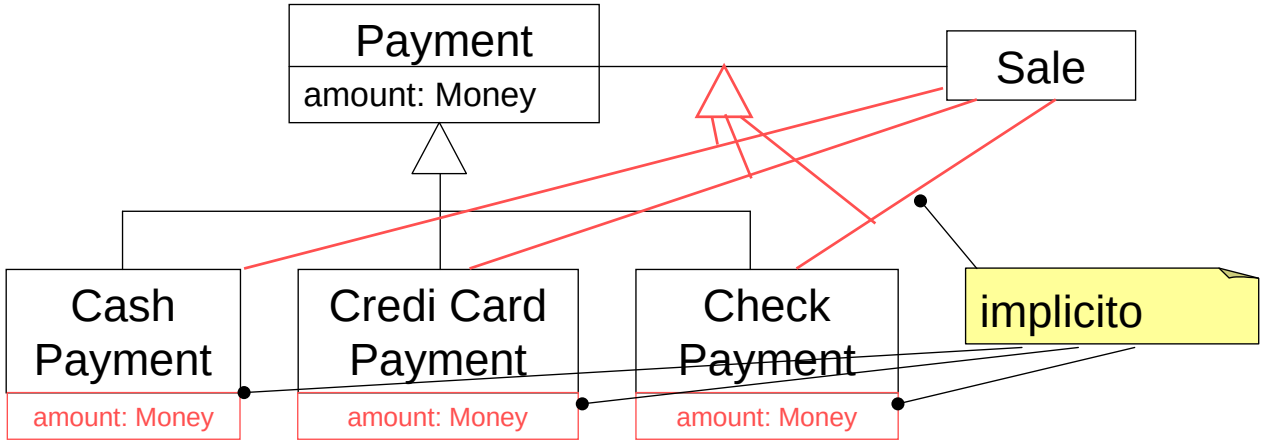
- ed ora?



- non fidatevi mai dell’interpretazione comune di “è un” per decidere la correttezza di una generalizzazione! specialmente in presenza di metodi modifier.

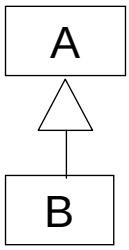
conformance: 100% rule

- tutte le “caratteristiche” delle classi o interfacce più generali sono (devono essere) proprie anche delle più specifiche
  - attributi, associazioni, comportamento



conformità tra interfacce

(o principio di sostituzione di Liskov)



- date due interfacce A e B
- B si dice conforme ad A se
  - B contiene almeno tutti i metodi di A
  - per ciascun metodo M_A di A considero il corrispondente metodo M_B di B, devono valere le seguenti implicazioni logiche
    - pre $M_A \rightarrow$ pre M_B

      cioè ogni volta che è lecito chiamare M_A è lecito anche chiamare M_B
    - post $M_A \leftarrow$ post M_B

      cioè ogni volta che faccio affidamento su post M_A potrei fare affidamento su post M_B poiché tutto ciò che è vero dopo una chiamata a M_A lo è dopo una chiamata a M_B

conformità: esempio 1

- l'interfaccia Rettangolo contiene i metodi
  - getAltezza(): real
    - pre: true (si può sempre chiamare)
    - post: "ritorna l'altezza e lo stato non è cambiato"
  - getLarghezza(): real
    - pre: true (si può sempre chiamare)
    - post: "ritorna la larghezza e lo stato non è cambiato"
- l'interfaccia Quadrato contiene i metodi
  - getAltezza(): real
    - pre: true (si può sempre chiamare)
    - post: "ritorna l'altezza e l'altezza è uguale alla larghezza e lo stato non è cambiato"
  - getLarghezza(): real
    - pre: true (si può sempre chiamare)
    - post: "ritorna la larghezza e la larghezza è uguale all'altezza e lo stato non è cambiato"
- Quadrato è conforme a Rettangolo?

conformità: esempio 2

- l'interfaccia Rettangolo contiene i metodi
  - getAltezza(): real
    - pre: true (si può sempre chiamare)
    - post: "ritorna l'altezza e lo stato non è cambiato"
  - getLarghezza(): real
    - pre: true (si può sempre chiamare)
    - post: "ritorna la larghezza e lo stato non è cambiato"
- l'interfaccia Quadrato contiene il metodo
  - getLato(): real
    - pre: true (si può sempre chiamare)
    - post: "ritorna il lato del quadrato e lo stato non è cambiato"
  - quadrato non ha le operazioni getAltezza() e getLarghezza()
- Quadrato è conforme a Rettangolo?

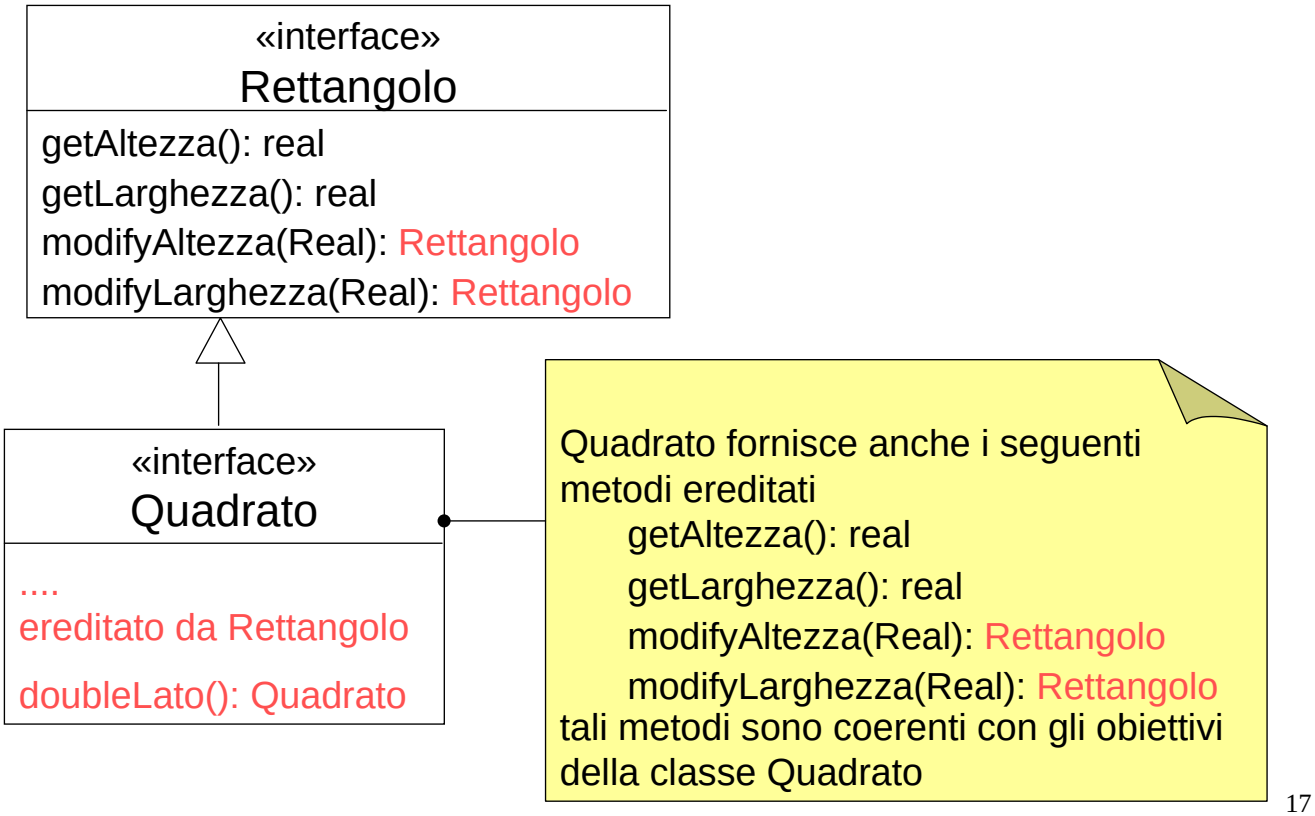
conformità: esempio 3

- l'interfaccia Rettangolo contiene i metodi
  - getAltezza(): real
  - getLarghezza(): real
  - setAltezza(*a*: real)
    - pre: "*a* è un real"
    - post: "l'altezza del rettangolo è *a* e la larghezza non è cambiata"
- l'interfaccia Quadrato contiene i metodi
  - getAltezza(): real
    - pre: true (si può sempre chiamare)
    - post: "ritorna l'altezza e l'altezza è uguale alla larghezza"
  - getLarghezza(): real
    - pre: true (si può sempre chiamare)
    - post: "ritorna la larghezza e la larghezza è uguale all'altezza"
  - quadrato ha la stessa setAltezza()
- Quadrato è conforme a Rettangolo? lo stato di Quadrato è sempre consistente?

esercizio

- progetta le interfacce di quadrato e di rettangolo in modo che
  - si possa leggere l'altezza e la larghezza di rettangoli e quadrati
  - si possano "ottenere" rettangoli per modifica dell'altezza o della larghezza a partire da un dato rettangolo
  - si possano "ottenere" quadrati che abbiano lato doppio rispetto ad un dato quadrato
  - il principio di sostituzione sia soddisfatto

soluzione


«interface»
Rettangolo
getAltezza(): real
getLarghezza(): real
modifyAltezza(Real): Rettangolo
modifyLarghezza(Real): Rettangolo
«interface»
Quadrato
....
ereditato da Rettangolo
doubleLato(): Quadrato
Quadrato fornisce anche i seguenti metodi ereditati
getAltezza(): real
getLarghezza(): real
modifyAltezza(Real): Rettangolo
modifyLarghezza(Real): Rettangolo
tali metodi sono coerenti con gli obiettivi della classe Quadrato