

# LEZIONE 9

## CLASS DIAGRAMS 2.1:

### Ereditarietà

Ingegneria del Software e Progettazione Web  
Università degli Studi di Tor Vergata - Roma







Guglielmo De Angelis  
guglielmo.deangelis@isti.cnr.it

# class diagrams

- struttura statica del sistema:
  - nodi + **relazioni**
- le relazioni di base in un class diagrams
  - semplificando : corrispondono alla definizione delle possibili interazioni tra le classi di un modello
    - una *relazione* tra una classe A ed una classe B significa che A è a conoscenza di B e (in qualche *modo*) può interagirvi
    - il tipo di relazione definisce il *modo* di interazione
    - l'assenza di relazioni nella classe implica l'isolamento di quella classe:
      - la classe é impossibilitata ad interagire con le altre


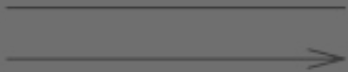

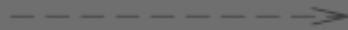


# le relazioni nei class diagrams

**Table 7.3 - Graphic paths included in structure diagrams**

<i>PATH TYPE</i>	<i>NOTATION</i>	<i>REFERENCE</i>
Aggregation		See “AggregationKind (from Kernel)” on page 38.
Association		See “Association (from Kernel)” on page 39.
Composition		See “AggregationKind (from Kernel)” on page 38.
Dependency		See “Dependency (from Dependencies)” on page 62.
Generalization		See “Generalization (from Kernel, PowerTypes)” on page 71.
InterfaceRealization		See “InterfaceRealization (from Interfaces)” on page 89.

# le relazioni nei class diagrams

Table 7.2 Graphic paths included in structure diagrams

<i>PATH TYPE</i>	<i>NOTATION</i>	<i>REFERENCE</i>
Aggregation		See “AggregationKind (from Kernel)” on page 38.
Association		See “Association (from Kernel)” on page 39.
Composition		See “AggregationKind (from Kernel)” on page 38.
Dependency		See “Dependency (from Dependencies)” on page 62.
Generalization		See “Generalization (from Kernel, PowerTypes)” on page 71.
InterfaceRealization		See “InterfaceRealization (from Interfaces)” on page 89.

# generalizzazione

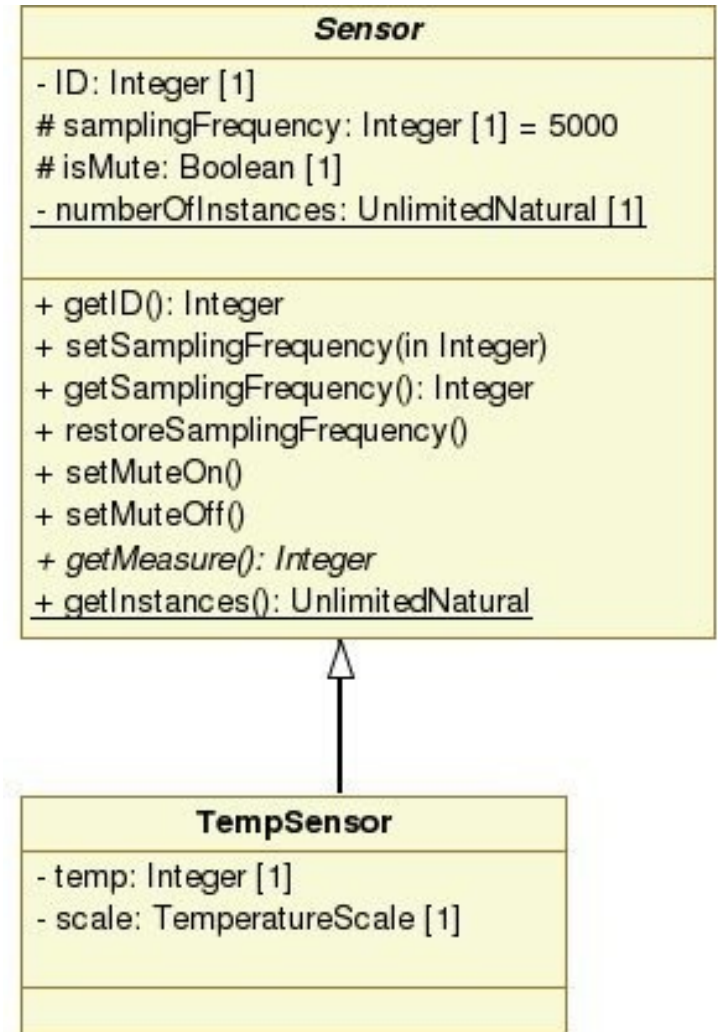
- relazione tra una classe più generale ed una più specifica
  - padre → super-classe
  - figlio → sotto-classe
- una sottoclasse eredita **tutte** le caratteristiche della superclasse
  - is-a-kind-of
  - la sottoclasse contiene più informazione

# generalizzazione

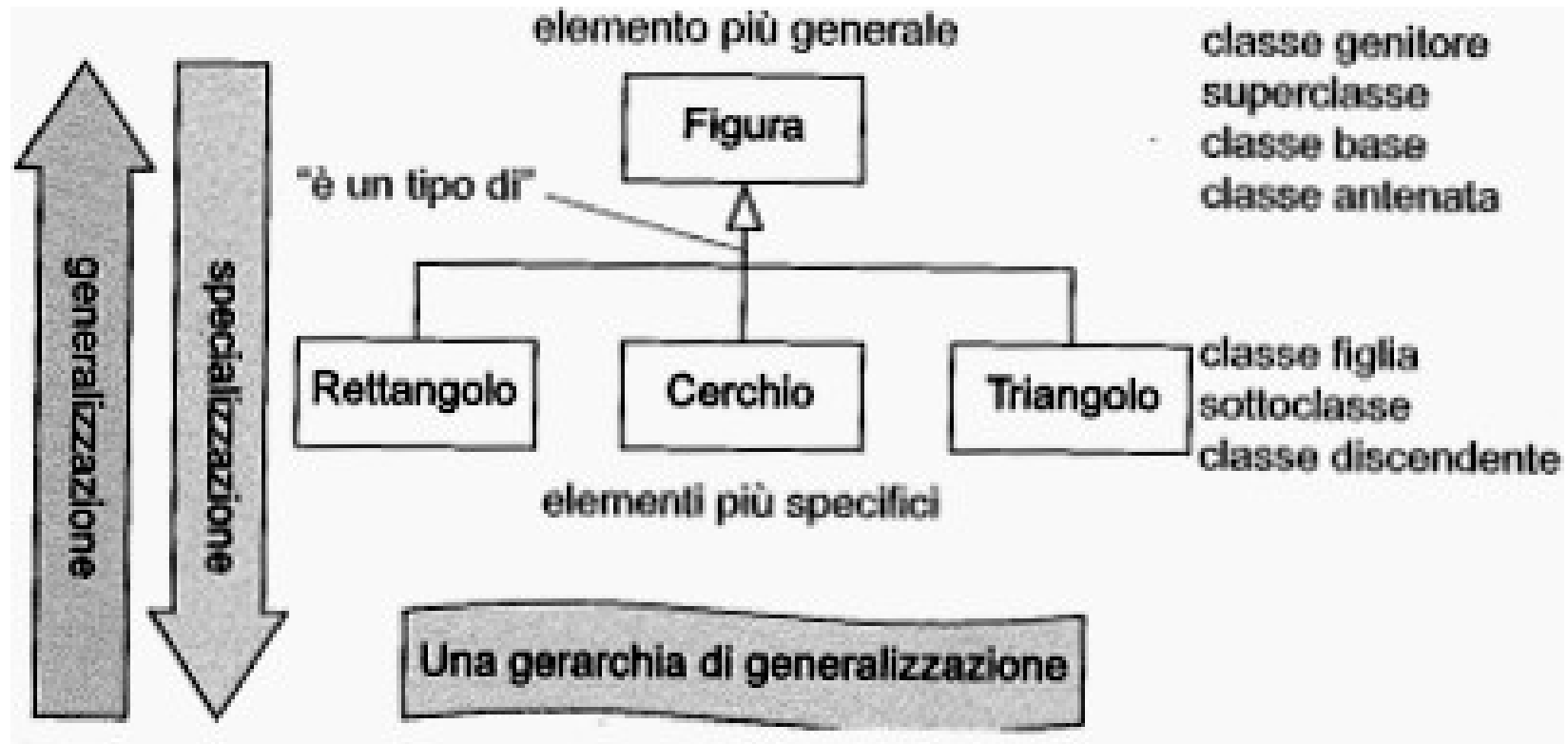
- è una relazione uniforme per tutti gli oggetti, senza eccezioni
  - non è possibile che alcuni oggetti ereditino proprietà da una superclasse mentre altri oggetti della stessa classe non ereditino dalla superclasse
- è una relazione transitiva e determina un ordinamento tra le classi

# generalizzazione

- le sottoclassi possono
  - aggiungere nuove caratteristiche
  - ridefinire operazioni che hanno ereditato
    - rendendo disponibile una nuova **operazione** con la stessa segnatura dell'**operazione** della superclasse che si desidera ridefinire
- in UML è rappresentata da una freccia con triangolo bianco (rivolto verso la superclasse)

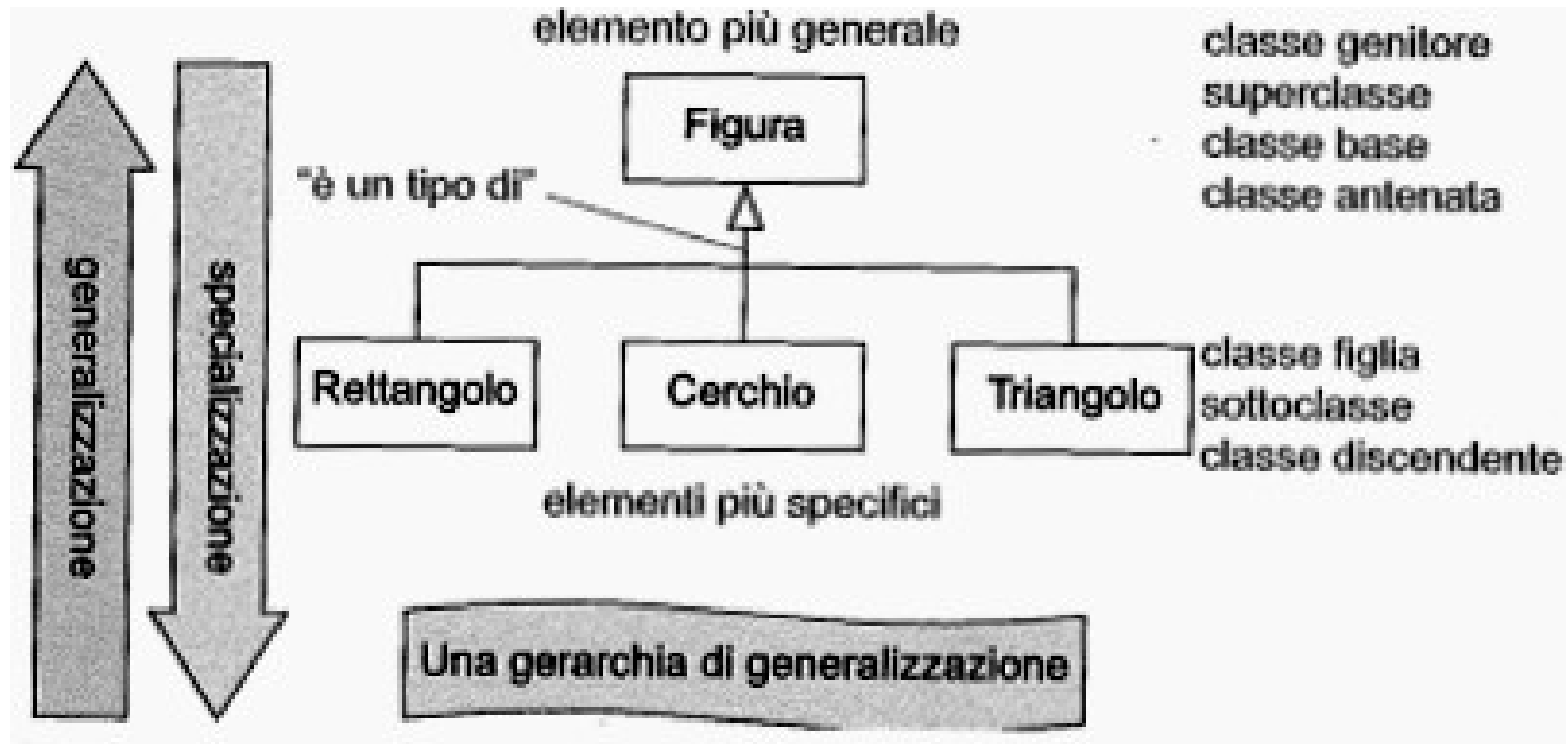


# generalizzazione **VS** specializzazione





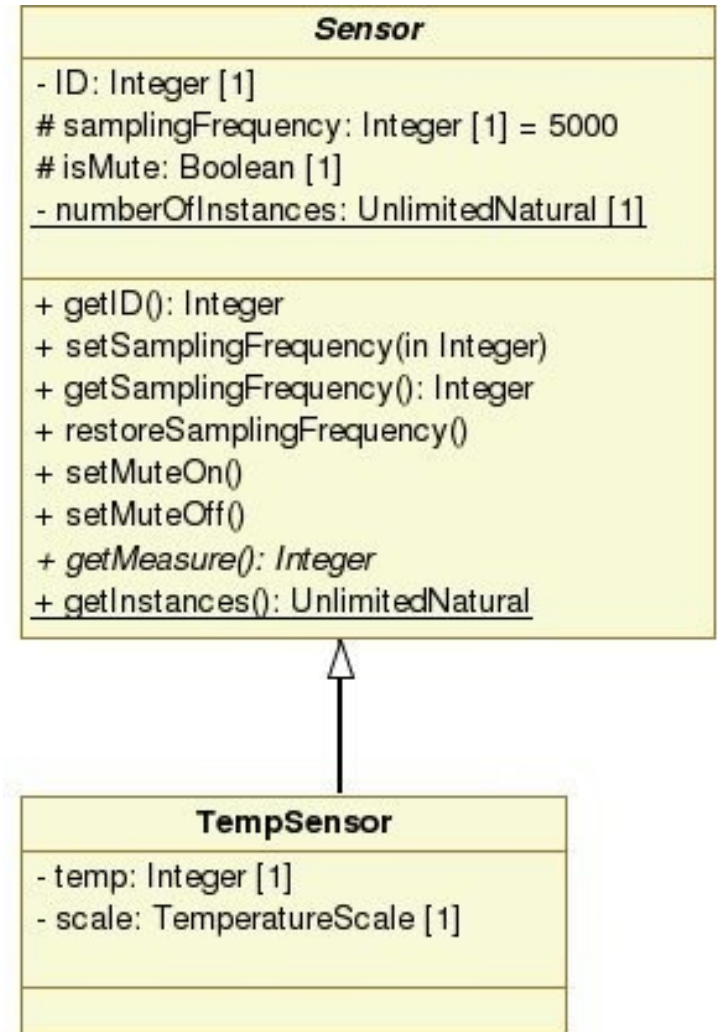
# generalizzazione VS specializzazione



- nella modellazione O.O. si tendono ad utilizzare entrambe le tecniche
- nel processo di design di un sistema software, l'esperienza pratica (in genere) consiglia di individuare e modellare concetti generici quanto prima possibile

# generalizzazione – map in Java

```
public class TempSensor
  extends Sensor {
  private int temp;
  private
    TemperatureScale scale;
  ...
}
```



# generalizzazione && costruttori

- un costruttore è una operazione speciale che ha il compito di istanziare opportunamente un oggetto
- il costruttore di una classe derivata ha la possibilità di accedere (senza restrizioni) i soli campi che la classe definisce
- il costruttore di ogni “classe base” viene sempre chiamato come prima operazione durante il processo di creazione di una classe derivata
  - il processo è ricorsivo su tutta la catena di generalizzazione
  - in questo modo si garantisce la corretta costruzione dell'oggetto
- se la classe derivata non chiama esplicitamente un costruttore della classe padre, viene implicitamente invocato il costruttore di default della classe base
  - se la classe base non ha costruttore di default, viene sollevato un errore di compilazione
- vedere codice di esempio:
  - `it.uniroma2.dicii.ispw.genExample.aboutConstructors`
  - `it.uniroma2.dicii.ispw.genExample.aboutScopes`

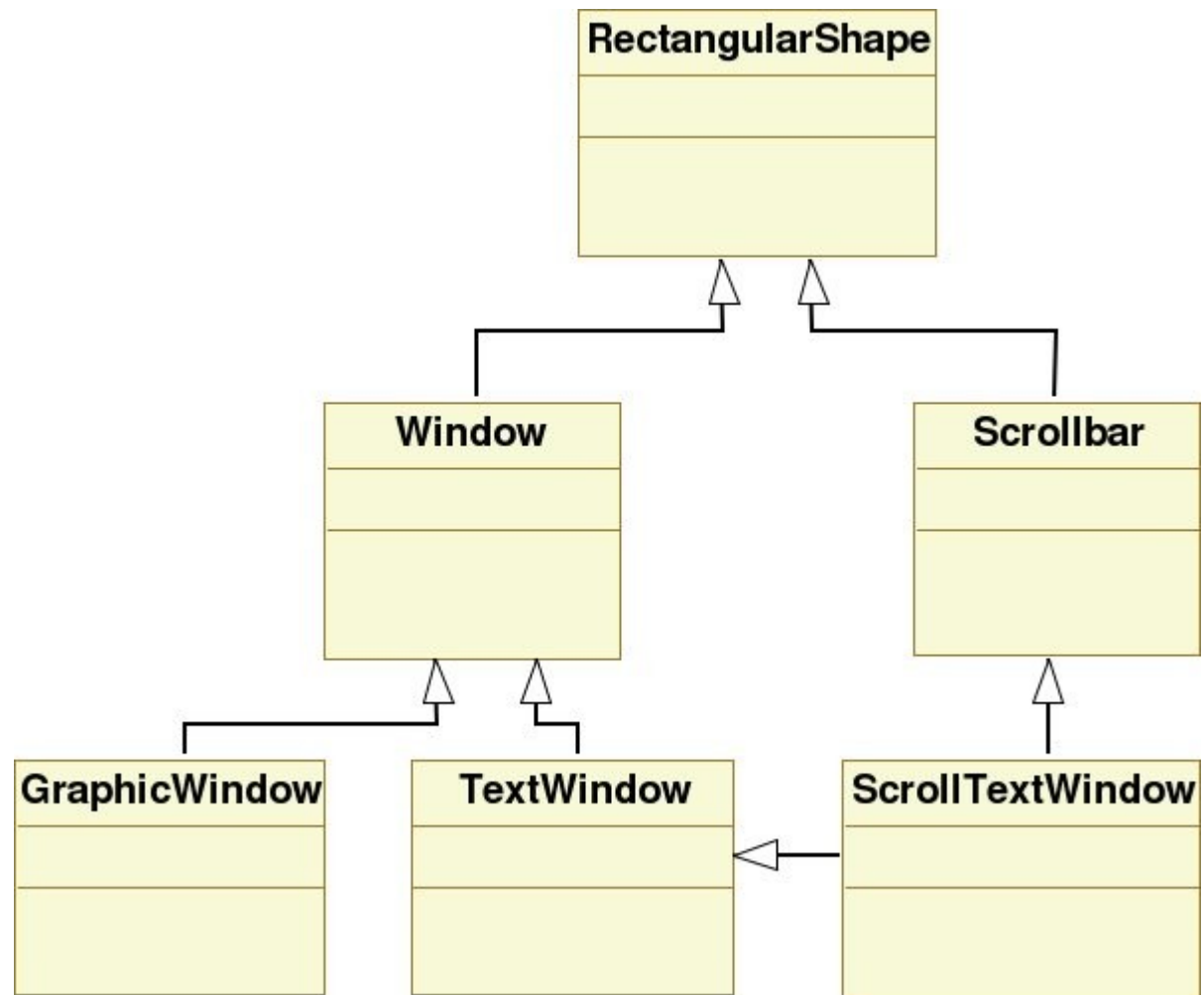
# ereditarietà multipla – 1

- una classe può avere **2 o più** superclassi dirette
- la sottoclasse eredita da **tutte** le superclassi
- non tutti i linguaggi O.O.  
supportano/consentono ereditarietà multipla
  - x** Java → NO
  - ✓** UML → SI
  - ✓** C++ → SI
  - x** Smalltalk → NO

# ereditarietà multipla – 2

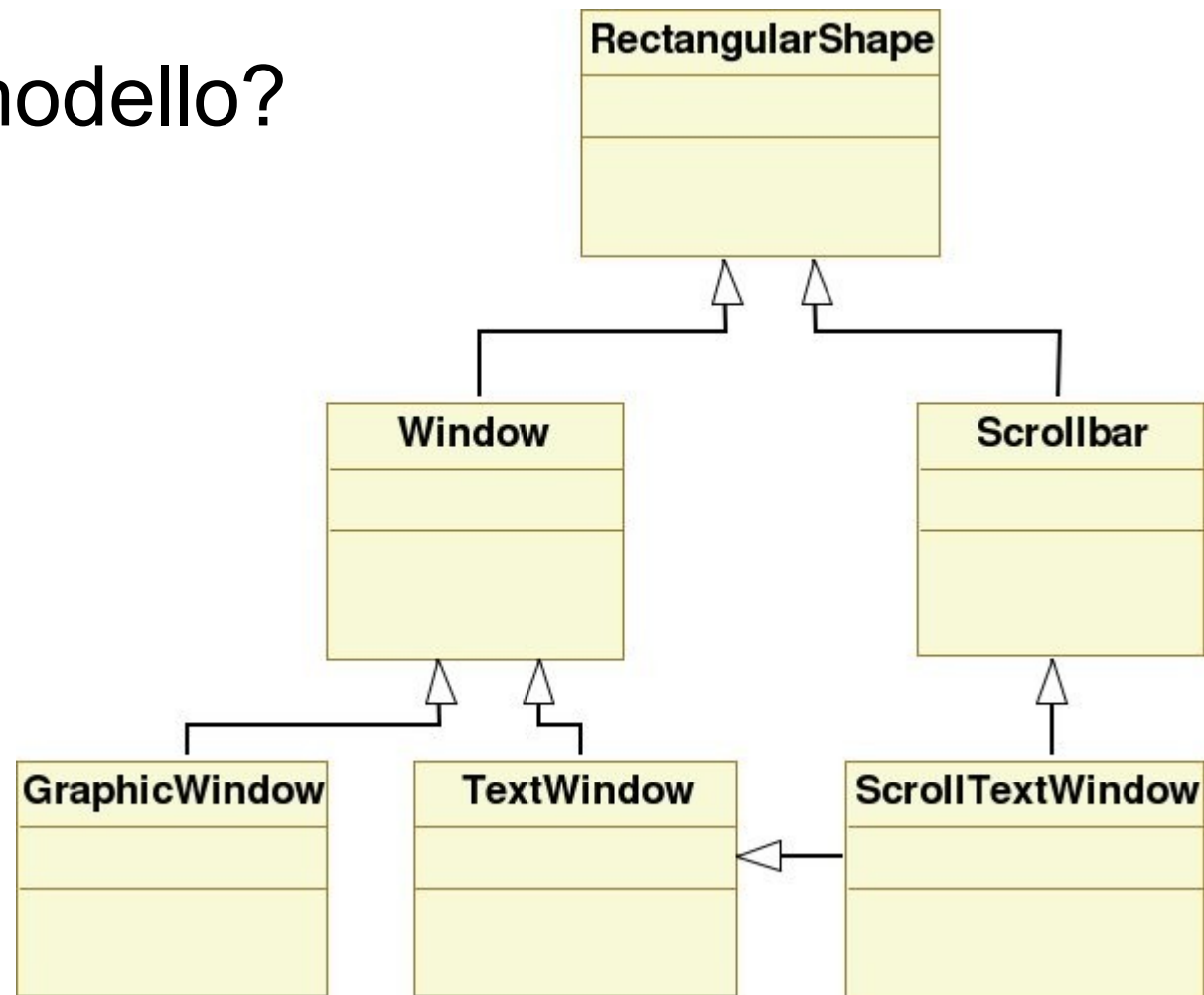
- vantaggi:
  - possibilità di comporre velocemente oggetti molto complessi
  - aggregare funzionalità differenti in un'unica classe
  - soluzione elegante effettivamente utile in molti casi pratici
    - migliore rappresentazione della realtà
    - semplificazione della sintassi
- svantaggi:
  - complicazione notevole del linguaggio che la implementa
  - gestire un linguaggio con ereditarietà multipla può essere complesso e poco chiaro (sia per chi specifica e mantiene il linguaggio sia per chi lo utilizza)
  - rischio elevato di **name clash**
    - se si ereditano membri con lo stesso nome da più di un genitore, avviene un conflitto
    - ci sono due metodi possibili per gestire la situazione
      - gestire il grafo di derivazione
      - applicare (dove possibile) criteri euristici per linearizzare la gerarchia (i.e. ordinamento dei rami “fratelli” nella gerarchia)
      - forzare la risoluzione di conflitti caso per caso

# generalizzazione – quando usarla?



# generalizzazione – quando usarla?

- quanto è buono questo modello?
- è corretto questo modello?

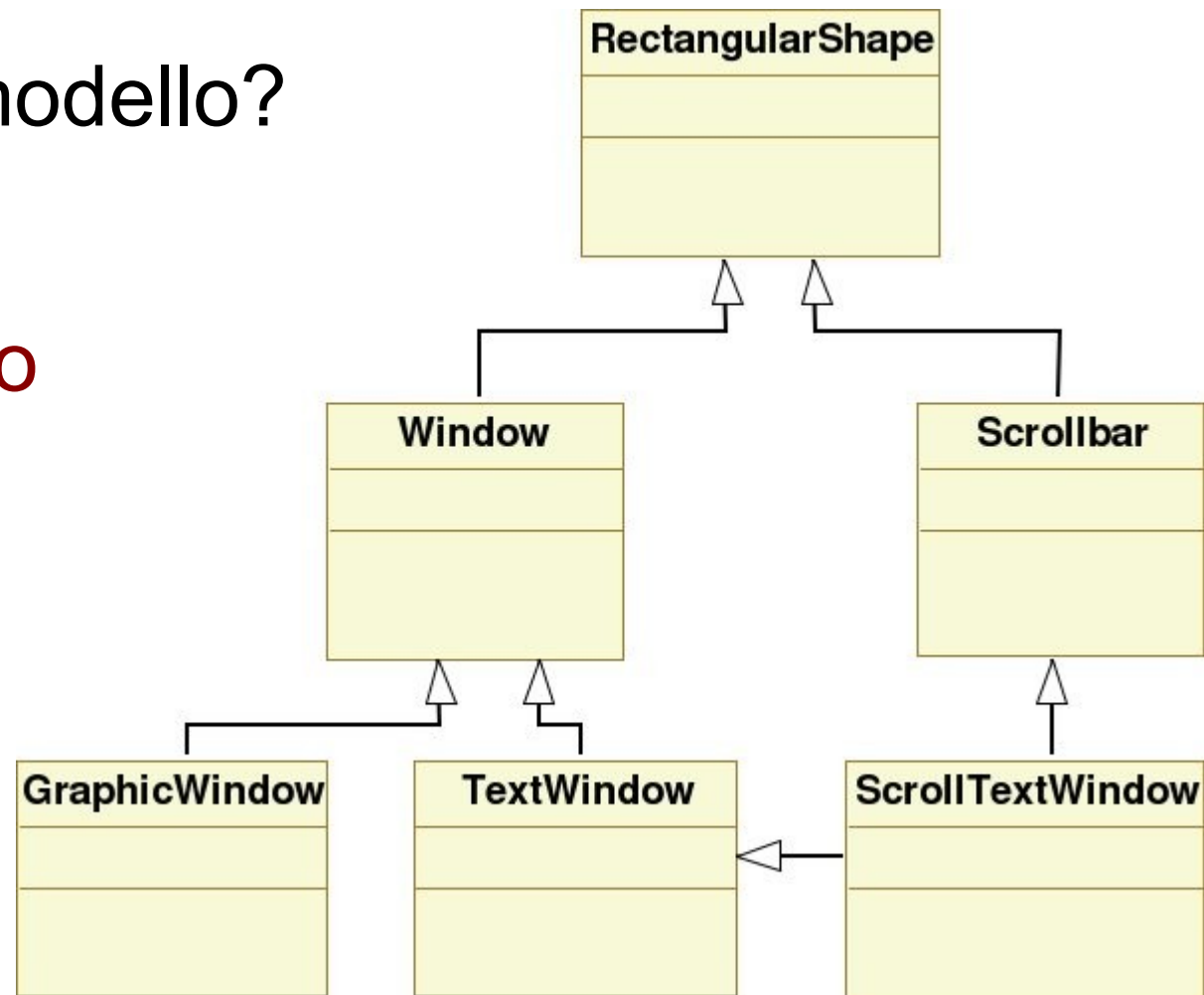


# generalizzazione – quando usarla?

- quanto è buono questo modello?
- è corretto questo modello?

- **dipende dallo scopo del modello:**

- finestra di testo con due scrollbar (orizzontale, verticale) ;





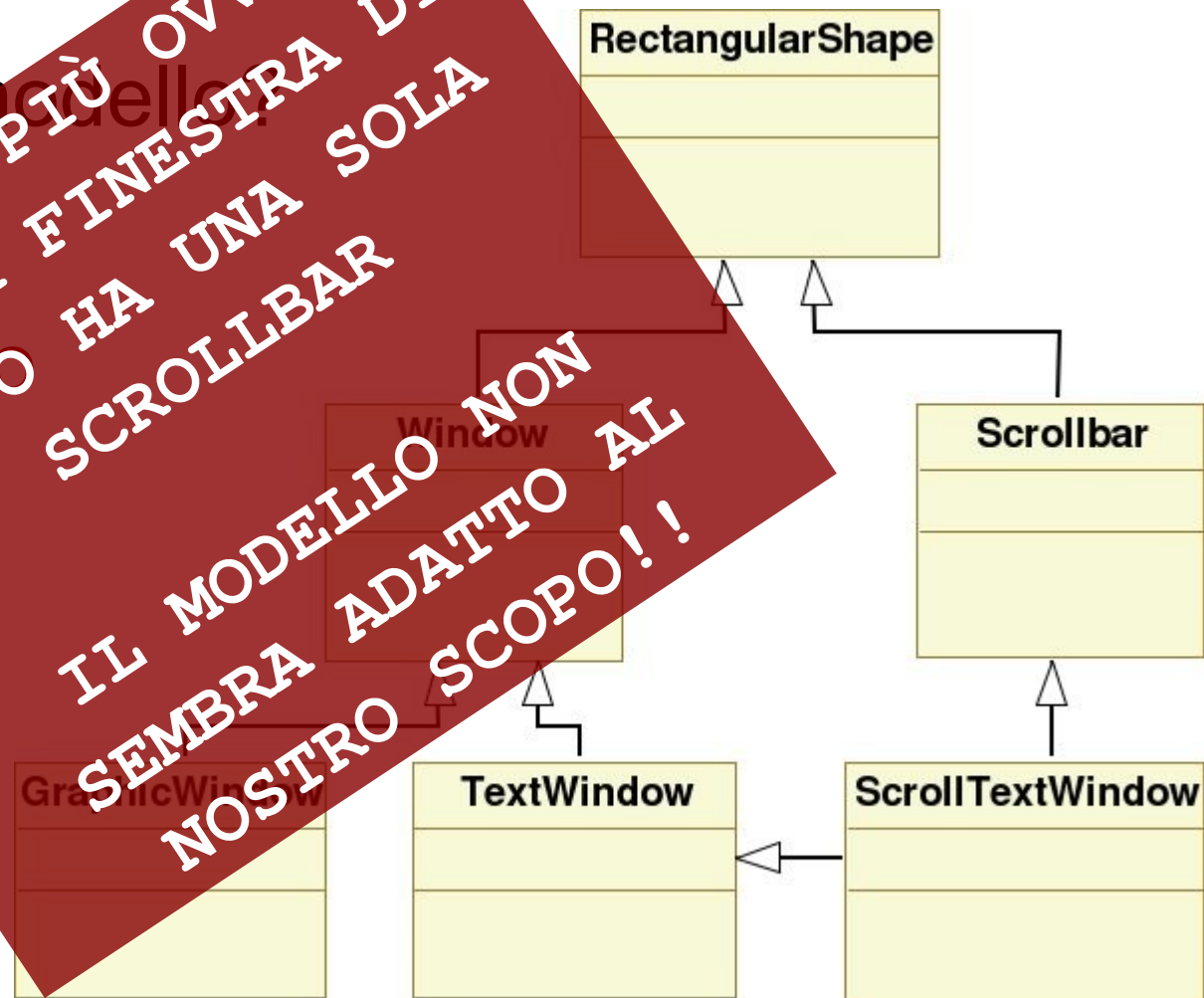
# generalizzazione – quando usarla?

- quanto è buono questo modello?
- è corretto questo modello?

- dipende dallo scopo del modello:

- finestra di testo con due scrollbar (orizzontale, verticale);

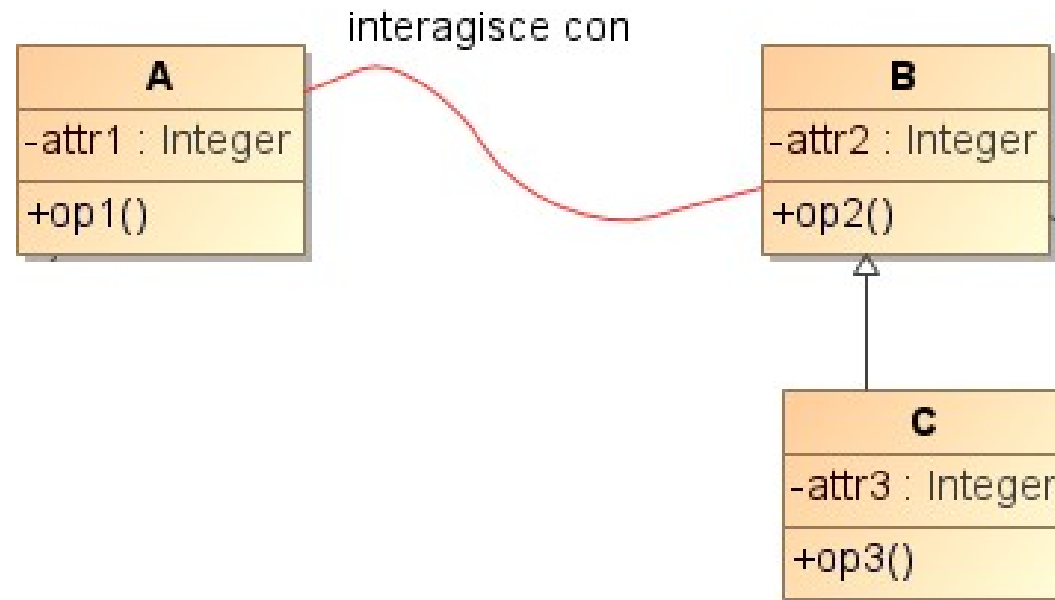
LA COSA PIÙ OVVIA È  
CHE LA FINESTRA DI  
TESTO HA UNA SOLA  
SCROLLBAR  
IL MODELLO NON  
SEMBRA ADATTO AL  
NOSTRO SCOPO!!



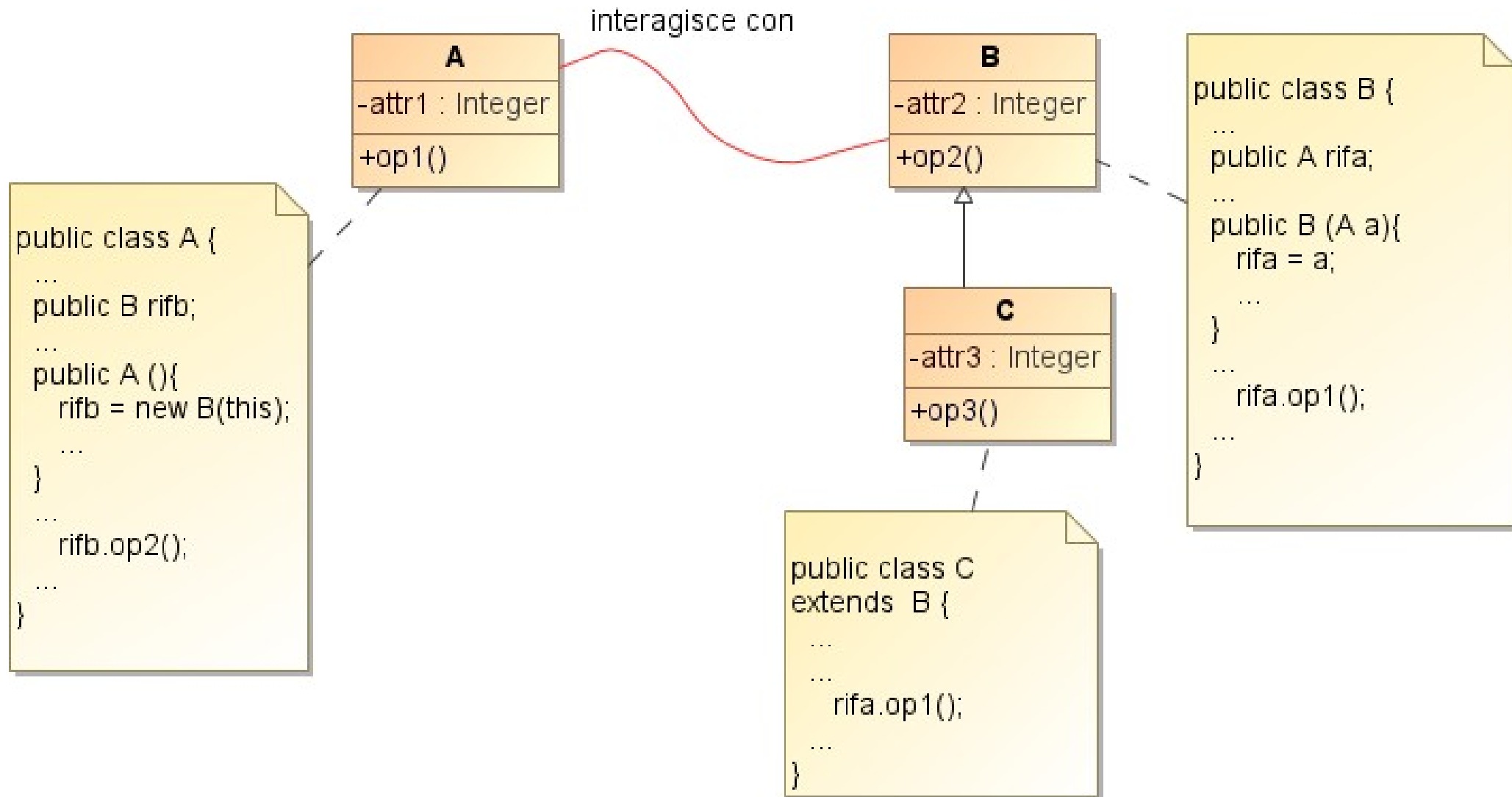
# generalizzazione – esempio di sostituzione

DISCUTIAMO INSIEME UN  
ESEMPIO SULLA  
RELAZIONE DI  
GENERALIZZAZIONE

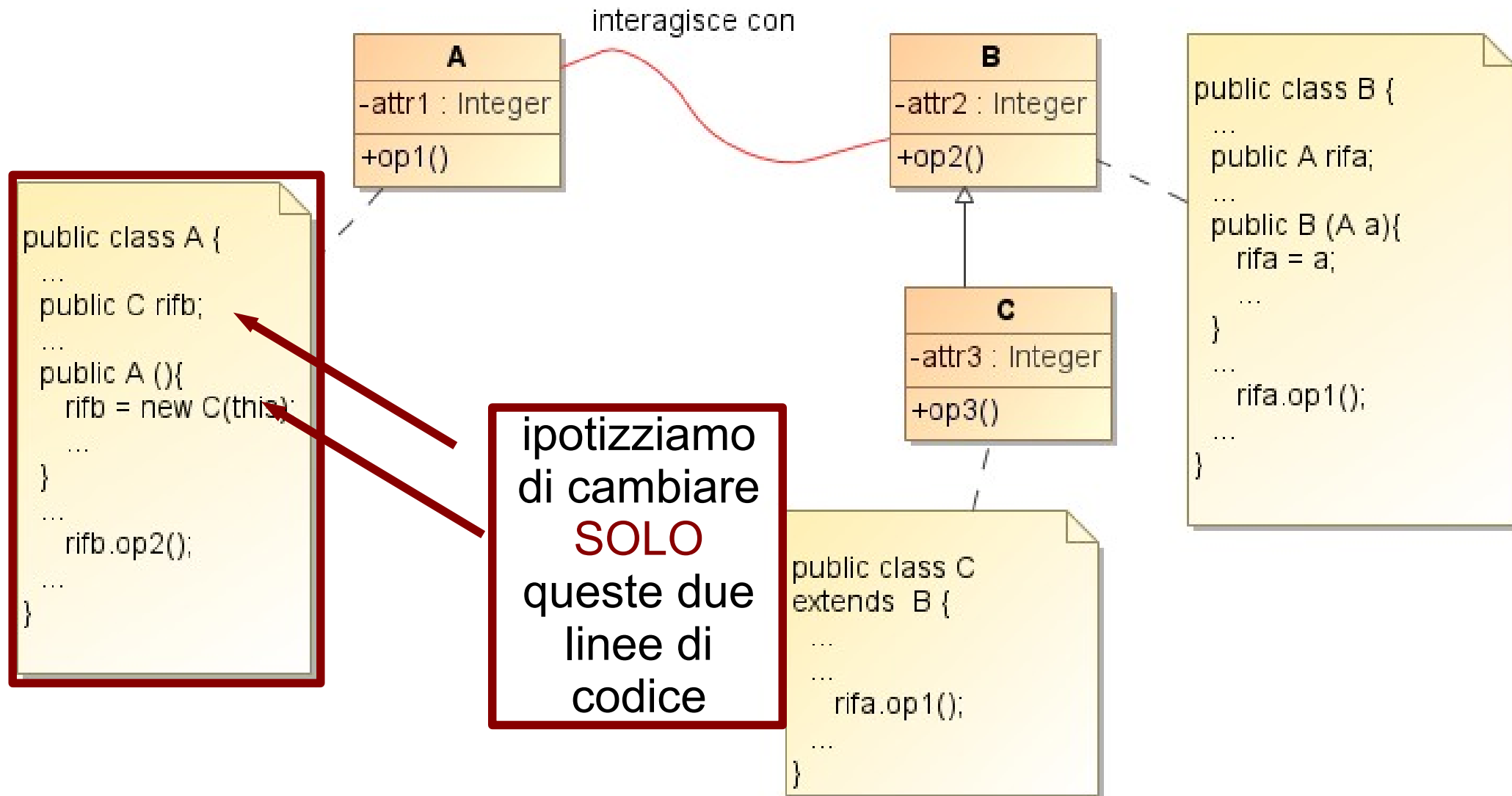
# generalizzazione – esempio di sostituzione



# generalizzazione – esempio di sostituzione



# generalizzazione – esempio di sostituzione



# generalizzazione – soluzione

- PRINCIPIO DI SOSTITUIBILITA' (di Liskov)
  - se  $q(x)$  è una proprietà che si può dimostrare essere valida per oggetti  $x$  di tipo  $T$ , allora  $q(y)$  deve essere valida per oggetti  $y$  di tipo  $S$  dove  $S$  è un sottotipo di  $T$ .

# generalizzazione – soluzione

- PRINCIPIO DI SOSTITUIBILITA' (di Liskov)
  - se  $q(x)$  è una proprietà che si può dimostrare essere valida per oggetti  $x$  di tipo  $T$ , allora  $q(y)$  deve essere valida per oggetti  $y$  di tipo  $S$  dove  $S$  è un sottotipo di  $T$ .
- che riformulato in termini più quotidiani:
  - data una superclasse  $T$  di  $S$ ; in tutti i contesti in cui si usa un'istanza di  $T$ , deve essere possibile utilizzare una qualsiasi istanza di  $S$  (o di una qualunque altra sottoclasse a qualsiasi livello)
- la sottoclasse deve avere la stessa semantica della superclasse

# generalizzazione – quando usarla?

- quanto è buono questo modello?

- questo modello rappresenta quello che?

- dipende dallo scopo del modello.
  - finestra di testo con due scrollbar (orizzontale, verticale) ;

Questo modello è sintatticamente corretto ma SEMANTICAMENTE SBAGLIATO

