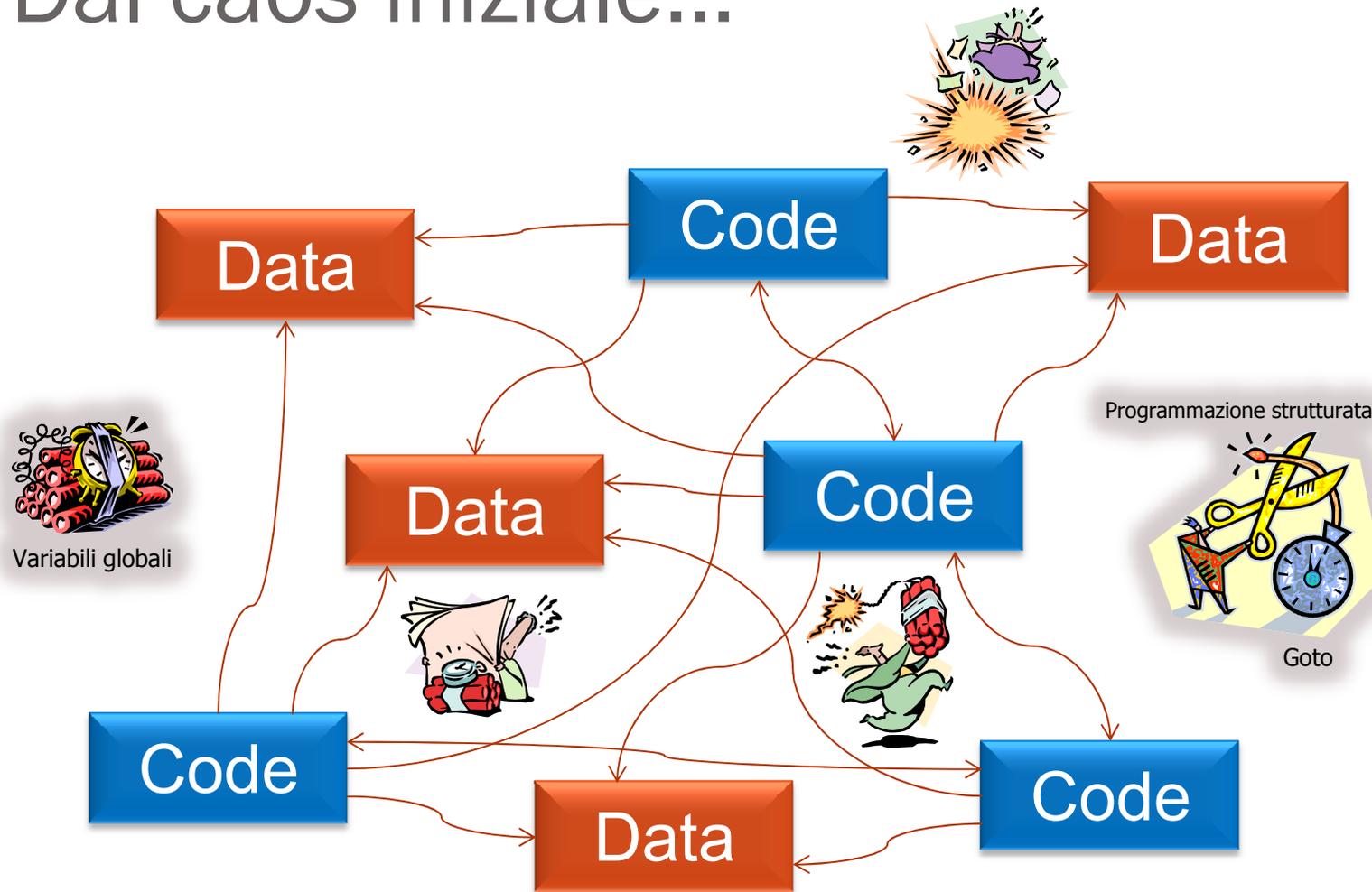


Ingegneria del Software T

Principi e concetti object-oriented

Dal caos iniziale...



Dal caos iniziale...

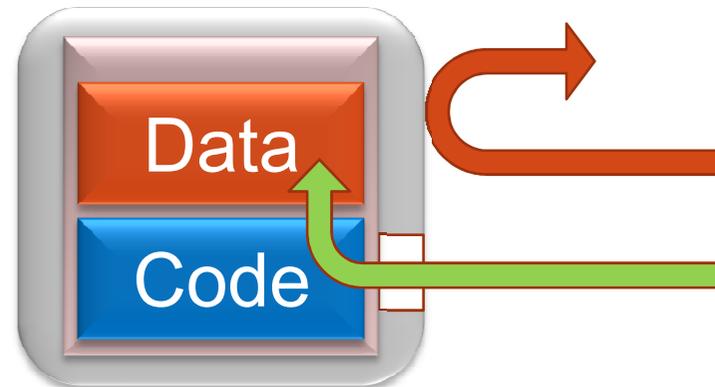
- Fortran (versione iniziale)
 - Caos nel flusso di controllo
 - `IF (espressione logica) GOTO 10`
 - `IF (espressione logica) 10,20`
 - `IF (espressione aritmetica) 10,20,30`
 - Caos nell'accesso ai dati
 - Istruzione `COMMON`:
`REAL V1 (10,10), V2 (10,10)`
`LOGICAL V3`
`INTEGER V4`
`COMMON /NOME/ V1, V2, V3, V4`
- C/C++
 - Uso indiscriminato delle variabili globali

...alla programmazione strutturata

- Nel 1966, Böhm e Jacopini dimostrano che qualsiasi programma che utilizza istruzioni `goto` può essere riscritto senza `goto`, a patto di avere a disposizione tre tipi di strutture di controllo: **sequenza**, **ripetizione** e **alternativa**
- Nel 1968, Dijkstra discute in modo approfondito gli effetti deleteri del `goto` sulla **qualità del software**, e in particolare sulla sua **leggibilità** e **modificabilità**

...alla programmazione basata sugli oggetti

- **ADT** (*Abstract Data Type*)
dati + codice che opera sui dati
interfaccia (visibile) + implementazione (nascosta)
- Lo stato di un oggetto è accessibile solo mediante l'interfaccia del suo ADT
- **Information hiding**
è il principio teorico
- **Incapsulamento**
è la tecnica utilizzata



Tipo di dato astratto

- Per **definire un ADT**, occorre definire
 - un'interfaccia (**interface**):
 - un insieme di **operazioni pubbliche applicabili ai singoli oggetti di quel tipo**
- Per **implementare un ADT**, occorre definire
 - una classe (**class**) che implementa l'interfaccia dell'ADT:
 - un insieme di **attributi privati** (implementazione della struttura dati specifica)
 - un insieme di **metodi pubblici** (implementazione dell'interfaccia) e di **metodi privati che accedono in esclusiva a tali attributi**

Information hiding – Incapsulamento

- Un ADT **nasconde** ai suoi utilizzatori (**clienti**) tutti i dettagli
 - della sua struttura interna e
 - del suo funzionamento interno
- **Obiettivo**
 - Nascondendo le **scelte progettuali** (spesso soggette a cambiamenti), si proteggono le altre parti del programma (i **clienti** dell'ADT) da eventuali cambiamenti di tali scelte
- **Vantaggi**
 - **Minimizzazione delle modifiche** da fare durante le fasi di **sviluppo** e di **manutenzione**
 - Aumento della possibilità di **riutilizzo**
- **Tecnica applicabile a tutti i livelli**
 - Singoli attributi membro di una classe
 - Singoli componenti del sistema
 - ...

Incapsulamento

Singoli attributi membro di una classe

```
public class Timer
{
    private int _count;
    ...
    public int GetCount()
    {
        return _count;
    }
    public void SetCount(int value)
    {
        if(value < 0)
            throw new ArgumentException("value < 0");
        _count = value;
    }
    ...
}
```

```
Timer x = new Timer();
...
x.SetCount(x.GetCount()+1);
```

Incapsulamento

Singoli attributi membro di una classe

```
public class Timer // in C#
{
    private int _count;
    ...
    public int Count
    {
        get
        {
            return _count;
        }
        set
        {
            if(value < 0)
                throw new ArgumentException("value < 0");
            _count = value;
        }
    }
    ...
}
```

Proprietà

```
Timer x = new Timer();
...
x.Count++;
```

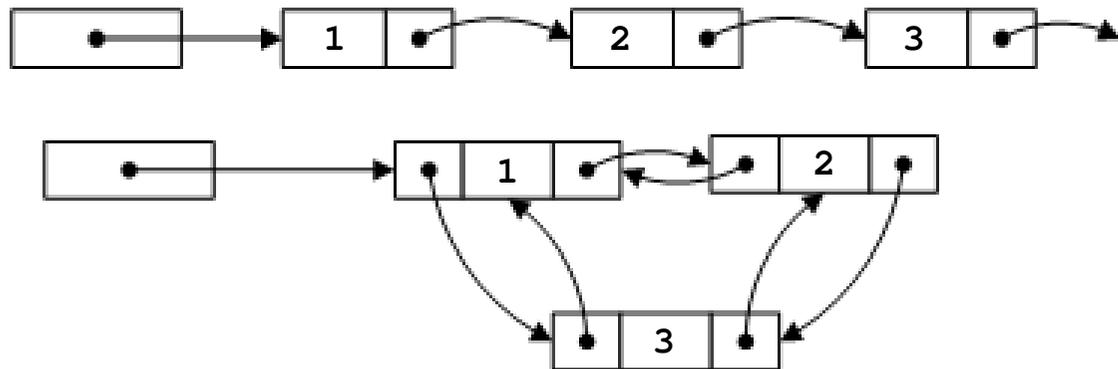
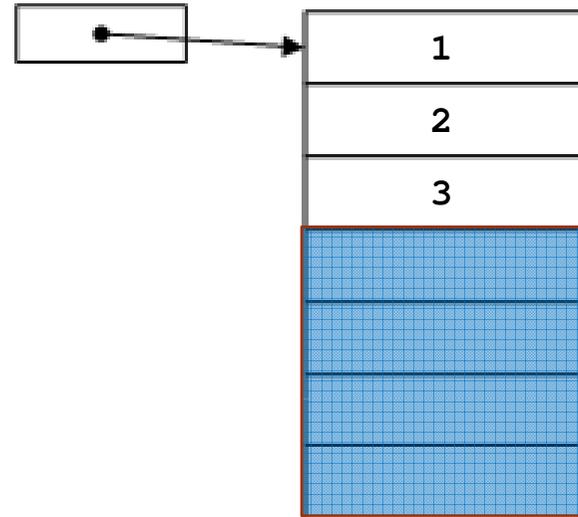
ADT Lista di interi

- **Interfaccia:**

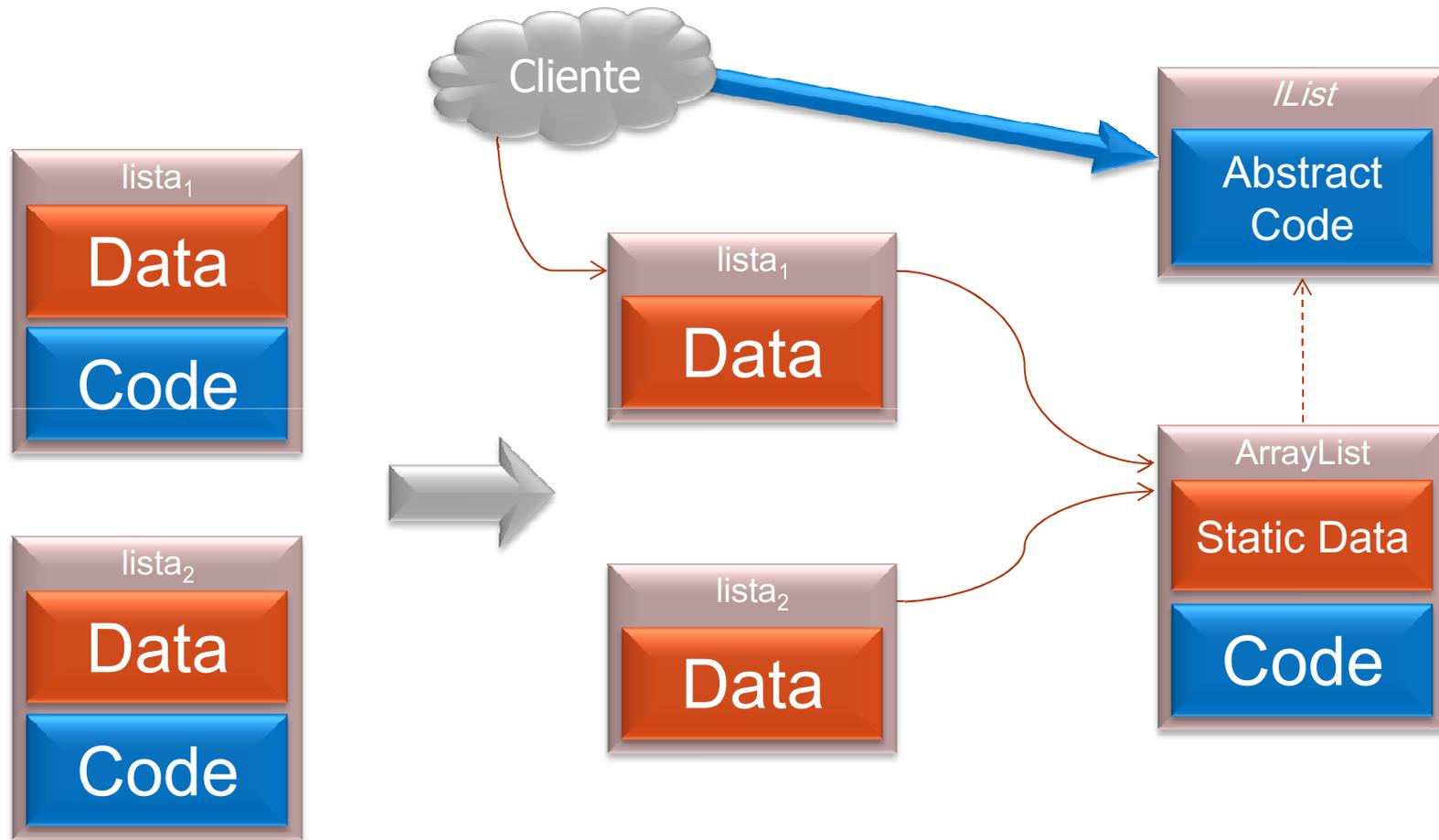
- `Add(int item)`
- `Insert(int index, int item)`
- `Remove(int item)`
- `RemoveAt(int index)`
- ...

- **Implementazione:**

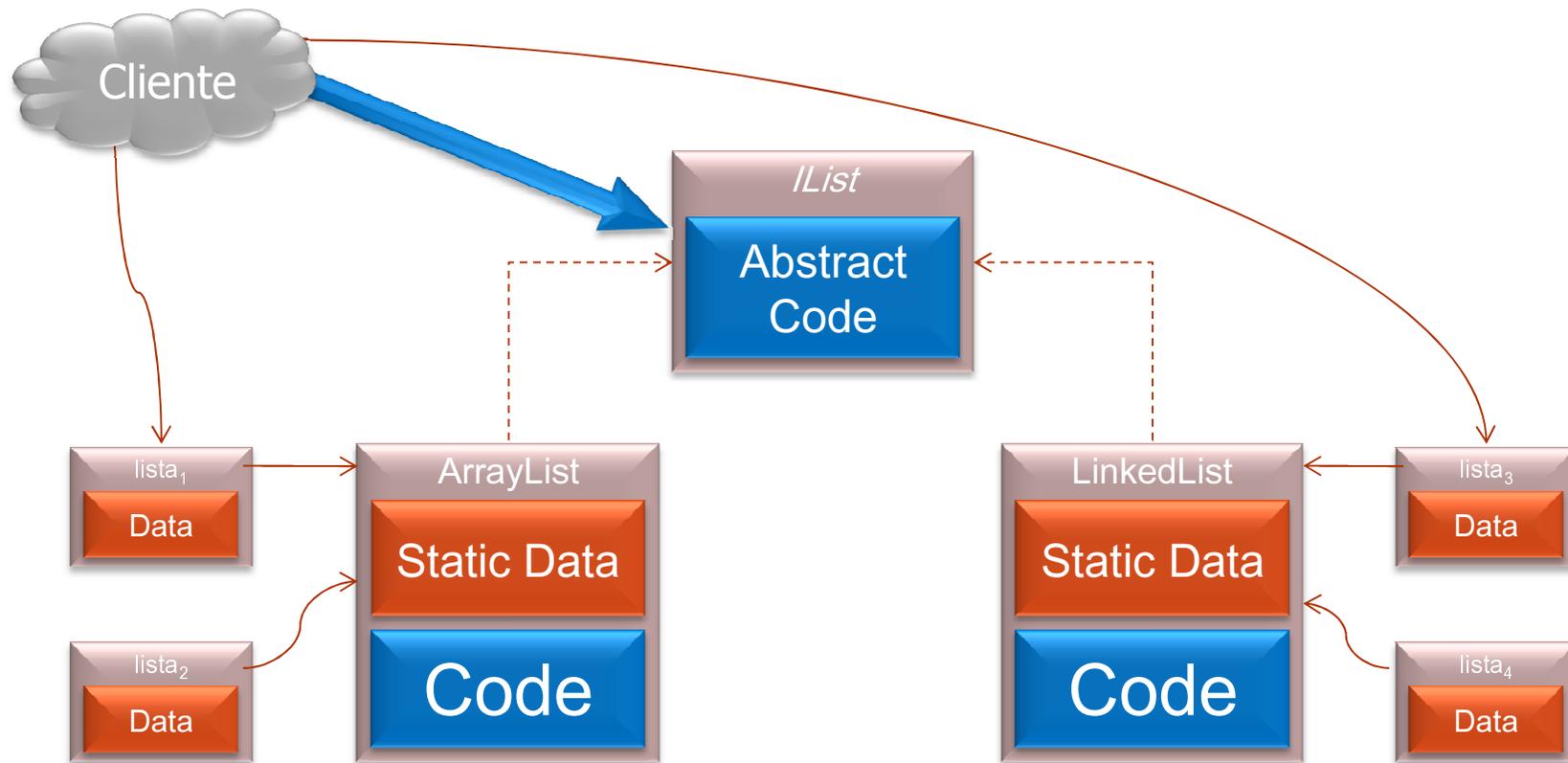
- Array
- Linked list
- ...



ADT Lista di interi



ADT Lista di interi



Oggetti & classi

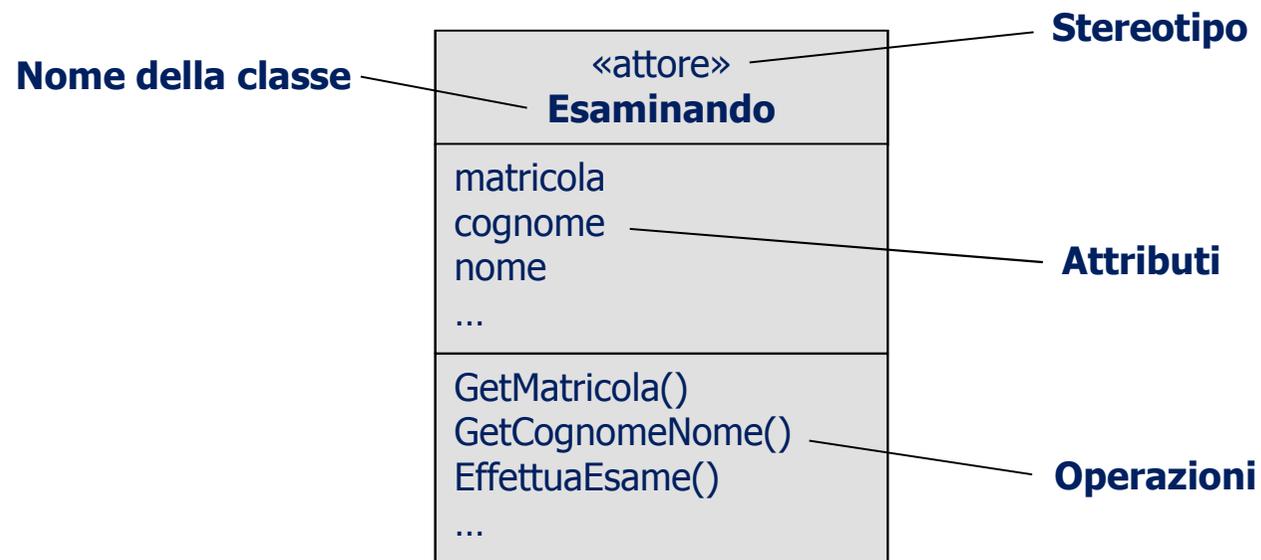
- Ogni **oggetto**:
 - è **identificabile** in modo univoco (ha una sua identità)
 - ha un insieme di **attributi**
 - ha uno **stato** (insieme dei valori associati ai suoi attributi)
 - ha un insieme di **operazioni**
 - che operano sul suo stato
 - che forniscono servizi ad altri oggetti
 - ha un **comportamento**
 - **interagisce** con altri oggetti

Oggetti & classi

- Gli oggetti sono raggruppabili in classi
- Ogni **classe** descrive oggetti con caratteristiche comuni, cioè:
 - con gli stessi attributi
 - con le stesse operazioni (lo stesso comportamento)
- **Compile time**, ogni **classe** definisce l'implementazione di un tipo di dato astratto
- **Run time**, ogni oggetto è un'**istanza** di una classe (traduzione comune anche se impropria del termine *instance*)
- Un'istanza è un particolare oggetto di una determinata classe e quindi di un particolare tipo
- Ogni istanza è separata dalle altre, ma condivide le sue caratteristiche generali con gli altri oggetti della stessa classe

Notazione UML

- Una classe si rappresenta come un rettangolo diviso in 1 o 3 sezioni

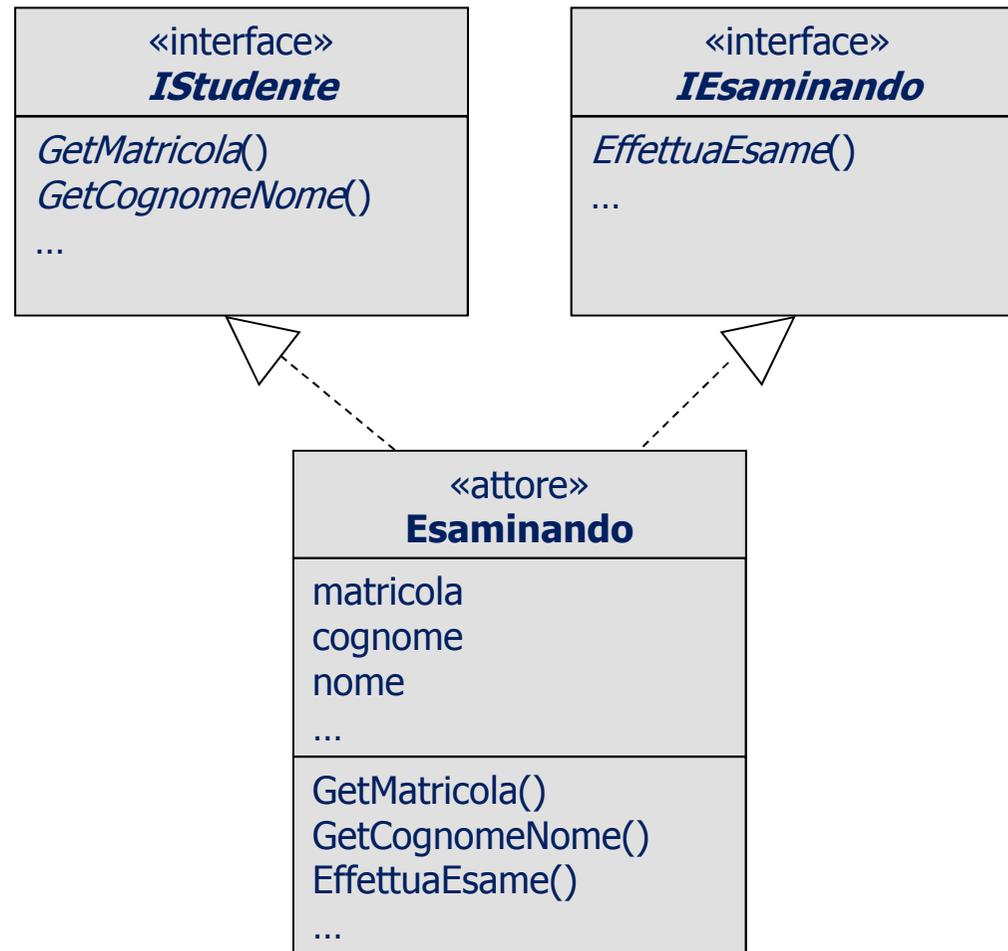


Notazione UML

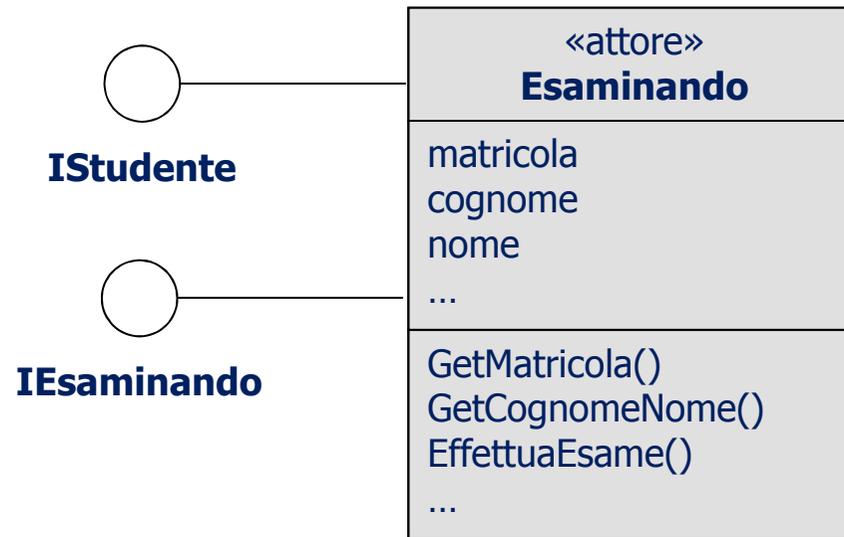


- La prima sezione contiene
 - il **nome della classe**
(in grassetto + in corsivo se astratta)può contenere
 - lo **stereotipo della classe**
(ad esempio, controllore, attore, evento, tabella, ecc.)
 - il **nome del pacchetto**
(*package, namespace*
ad esempio, Quizzer::**Esaminando**)
- La seconda sezione contiene
 - gli **attributi**
- La terza sezione contiene
 - le **operazioni** (in corsivo se astratte)

Notazione UML



Notazione UML



Notazione UML



...alla programmazione orientata agli oggetti

- Le classi possono essere organizzate in una **gerarchia di generalizzazione o di ereditarietà** che mostra la relazione tra classi di oggetti generiche e classi di oggetti più specifiche
- Gli oggetti della **sottoclasse** devono essere in grado di esibire tutti i comportamenti e le proprietà esibiti dagli oggetti appartenenti alla **superclasse**, in modo tale da poter essere "sostituiti" liberamente a questi ultimi (**principio di sostituibilità di Liskov**)
- La sottoclasse può
 - esibire caratteristiche aggiuntive rispetto alla superclasse
 - eseguire in maniera differente alcune delle funzionalità della superclasse, a patto che questa differenza non sia osservabile dall'esterno

...alla programmazione orientata agli oggetti

- **Ereditarietà**

- **Attributi e operazioni comuni**
devono essere specificati una volta sola
- **Attributi e operazioni specifici**
vengono aggiunti e/o ridefiniti

- **Obiettivo**

- Semplificare la **definizione** e la **realizzazione**
di tipi di dato simili
- Permette di esprimere esplicitamente le caratteristiche comuni,
sino dalle prime attività dell'analisi

Ereditarietà (*inheritance*)

- **Model inheritance**

- **Subtype inheritance**
- **Extension inheritance**
- Restriction inheritance
- View inheritance

reflecting "is-a" relations between abstractions in the model

- **Software inheritance**

- Reification inheritance
- Structure inheritance
- **Implementation inheritance**
- Facility inheritance
 - Constant inheritance
 - Machine inheritance

expressing relations within the software itself rather than in the model

- **Variation inheritance**

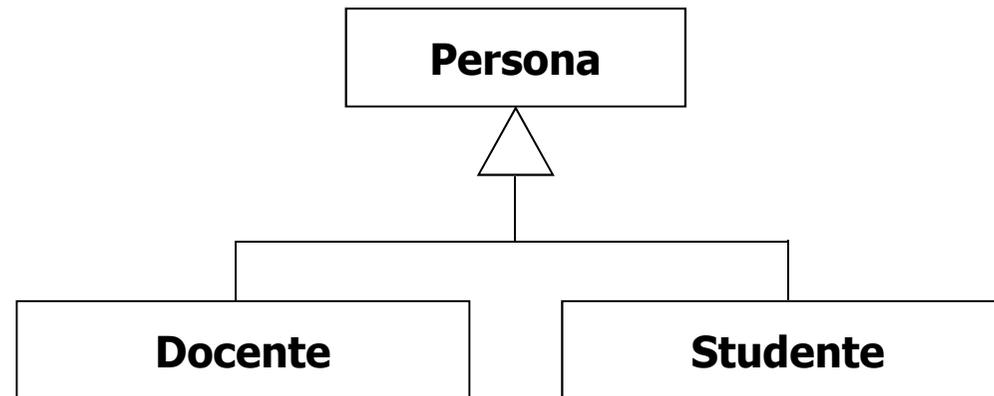
- Functional variation inheritance
- Type variation inheritance
- Uneffecting inheritance

a special case that may pertain either to the software or to the model

Ereditarietà

- Ereditarietà di interfaccia o *subtyping*
Ereditarietà di estensione
 - meccanismi di compatibilità fra tipi:
una sottoclasse è un sottotipo compatibile con tutti i tipi definiti lungo la sua catena ereditaria (**relazione IsA**)
 - consentono il **polimorfismo per inclusione**
- Ereditarietà di realizzazione (o di implementazione) o *subclassing*
 - meccanismo di riuso:
si riutilizza il codice definito nelle superclassi
 - ammessa in C++, non ammessa in Java e .NET

Ereditarietà (di interfaccia o di estensione)



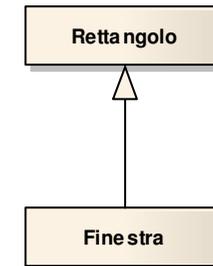
- Un Docente è una Persona
 - un Docente può essere utilizzato come una Persona
- Uno Studente è una Persona
 - uno Studente può essere utilizzato come una Persona
- Non è detto che una Persona sia un Docente o uno Studente
- E se una Persona è sia un Docente, sia uno Studente?

Ereditarietà di realizzazione

- Spesso una classe ha bisogno di utilizzare i **servizi** di un'altra classe
- Ad esempio, la classe **Finestra** ha bisogno di utilizzare la classe **Rettangolo** per
 - memorizzare posizione e dimensione
 - fare calcoli di sovrapposizione con altre finestre
 - ...
- Potrei definire **Finestra** come sottoclasse di **Rettangolo** (ma una **Finestra** NON è un **Rettangolo**)
 - Finestra eredita e quindi ha accesso diretto a dati e operazioni (public e protected) di Rettangolo
 - i clienti della classe Finestra NON devono avere accesso a dati e operazioni (anche se public) della classe Rettangolo

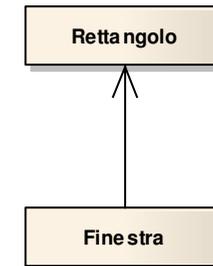
Ereditarietà di realizzazione

- In C++:
`public class Finestra : private Rettangolo`
- La definizione è **statica** (*compile-time*)
- L'implementazione della sottoclasse è facile da modificare, può definire i propri metodi e continuare a usare quelli della superclasse
- La superclasse definisce parte della rappresentazione fisica della sottoclasse, legando a sé la sottoclasse, **rompendo l'incapsulamento** (Finestra vede i membri protected di Rettangolo) **e rendendo più difficile il riuso della sottoclasse**



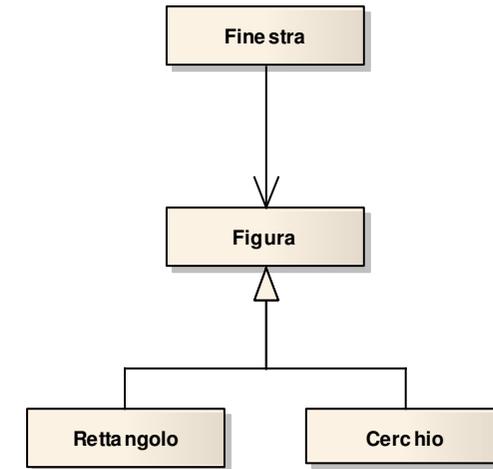
Composizione e delega

- Esiste un'alternativa più interessante: inserire un Rettangolo nella struttura dati di una Finestra: **una Finestra contiene un Rettangolo** (una Finestra è **composta**, tra le altre cose, da un Rettangolo)
- Una Finestra ha accesso indiretto alle operazioni pubbliche di un Rettangolo (una Finestra **delega** al Rettangolo l'esecuzione di alcuni compiti)
- Le interfacce delle classi restano **indipendenti**



Composizione e delega

- L'associazione tra Finestra e Rettangolo può avvenire **dinamicamente** (*run-time*)
- Maggiore flessibilità ed estendibilità!
- Quindi
 - se e solo se vale la relazione IsA, usare **l'ereditarietà**
 - altrimenti, usare la **composizione**

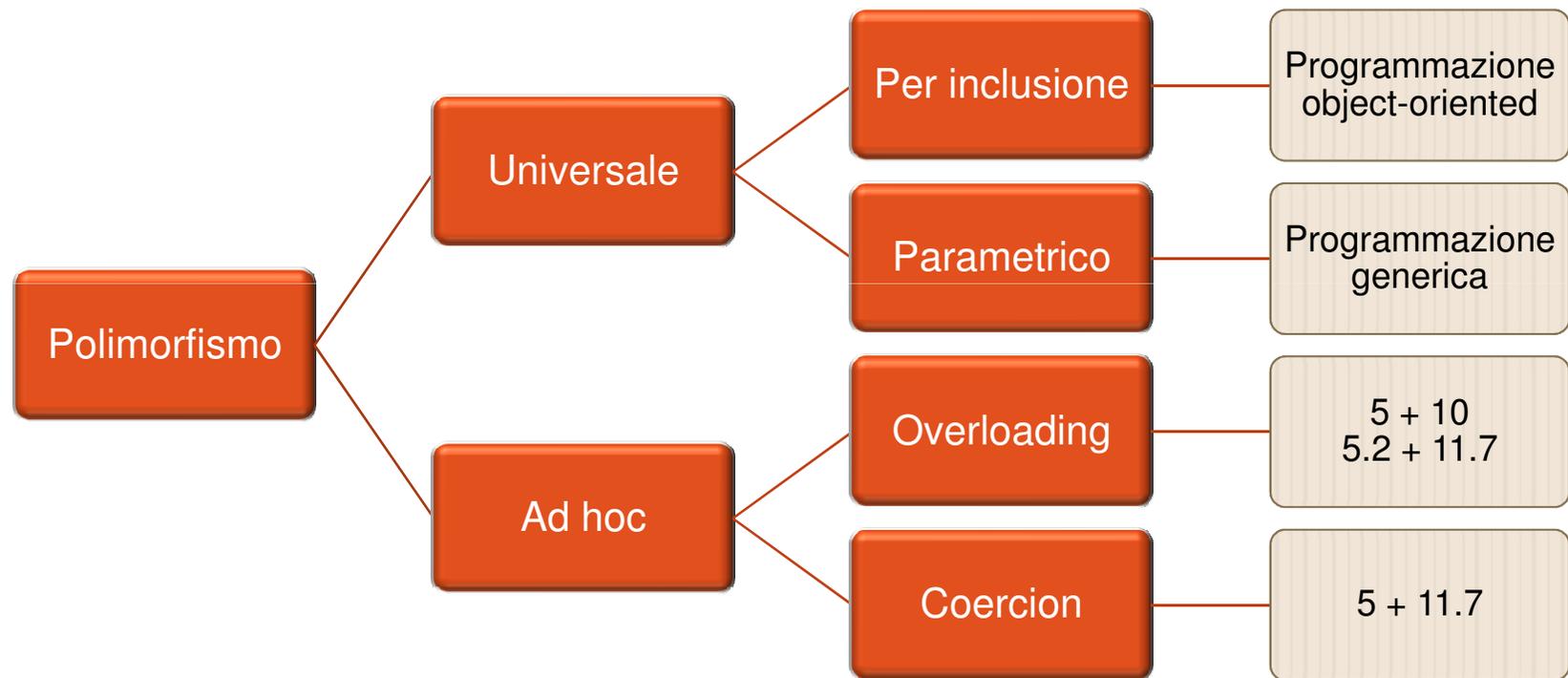


Polimorfismo

- Capacità
 - della stessa cosa di apparire in forme diverse in contesti diversi
 - $5 + 11.7$
 - $5.0 + 11.7$
 - di cose diverse di apparire sotto la stessa forma in un determinato contesto
 - x.**Fun1** run-time possono essere invocato metodi diversi (late-binding)

Polimorfismo

Classificazione Cardelli-Wegner



Polimorfismo (per inclusione)

- **Overriding** (ridefinizione) dei metodi
 - Definizione di un metodo astratto (sicuro)
 - Ridefinizione di un metodo concreto (meno sicuro)
- **Binding dinamico** (o late-binding)
- **Virtual Method Table** (VMT)
è la struttura utilizzata per implementare il binding dinamico

Virtual Method Table

```
public class A
{
    public virtual void Fun1(int x)
    { ... }

    public virtual void Fun2(int y)
    { ... }
}
```

<<code>>
void A::Fun1(int)

```
public class B : A
{
    public override void Fun1(int x)
    { ... }

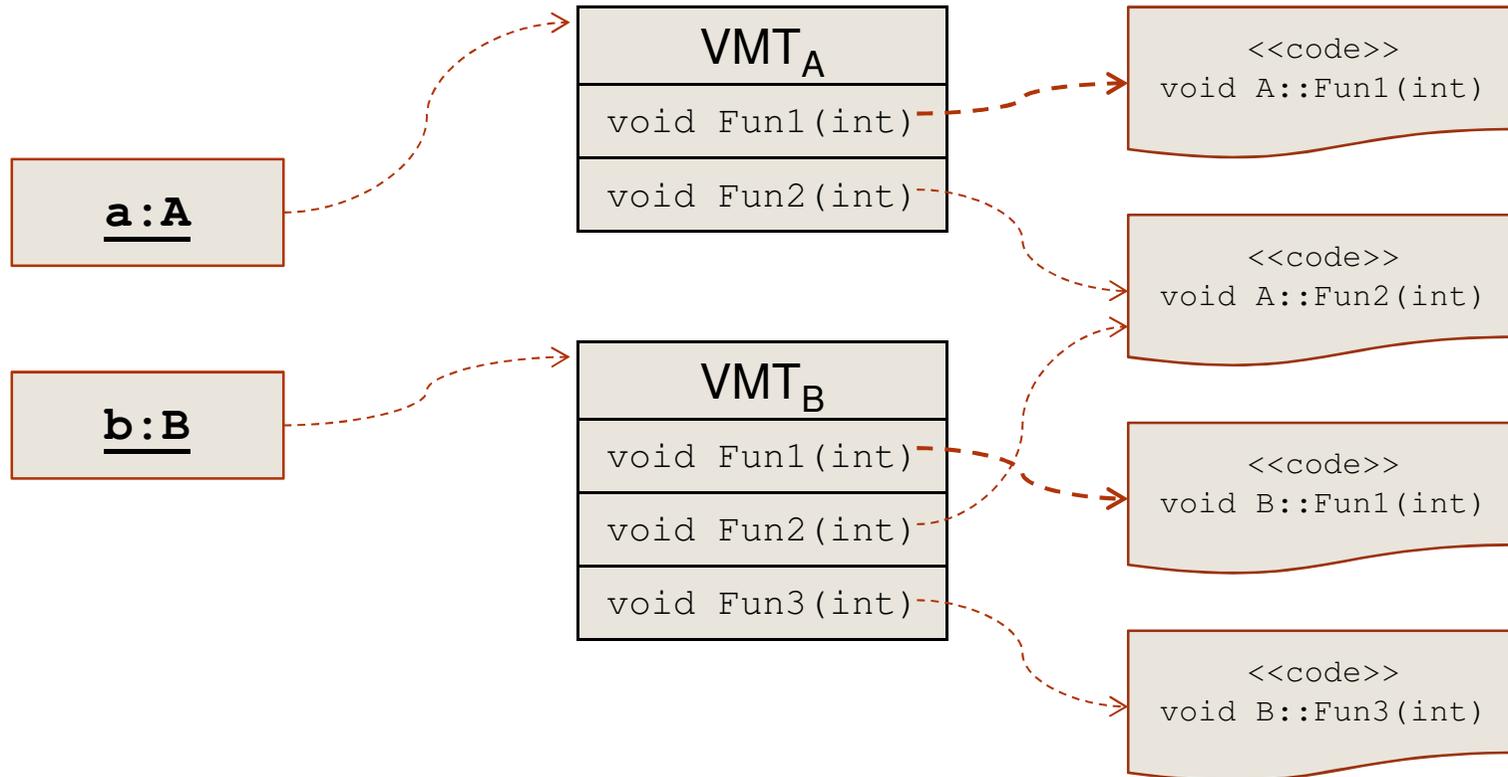
    public virtual void Fun3(int z)
    { ... }
}
```

<<code>>
void A::Fun2(int)

<<code>>
void B::Fun1(int)

<<code>>
void B::Fun3(int)

Virtual Method Table



Virtual Method Table

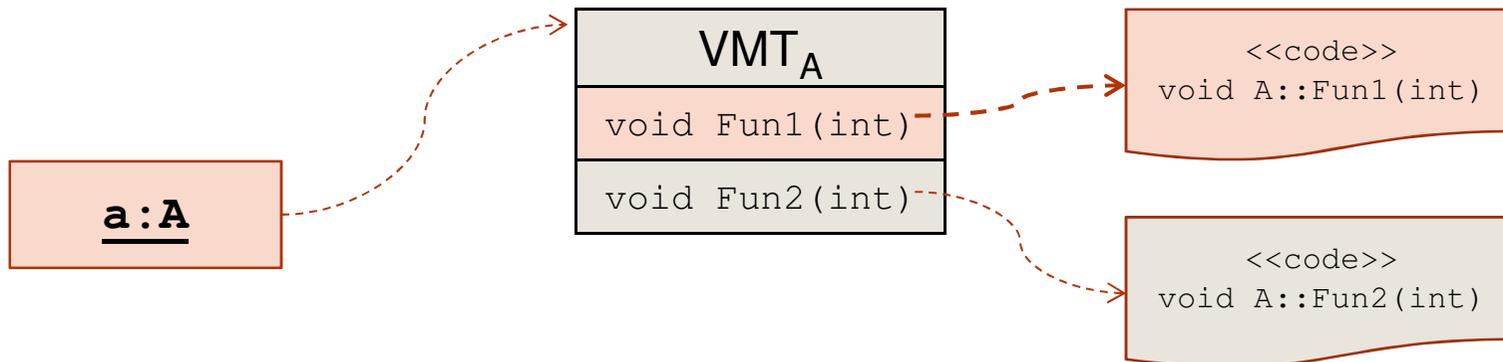
```
A x;
```

```
...
```

```
x = a;
```

```
...
```

```
x.Fun1(10);
```



Virtual Method Table

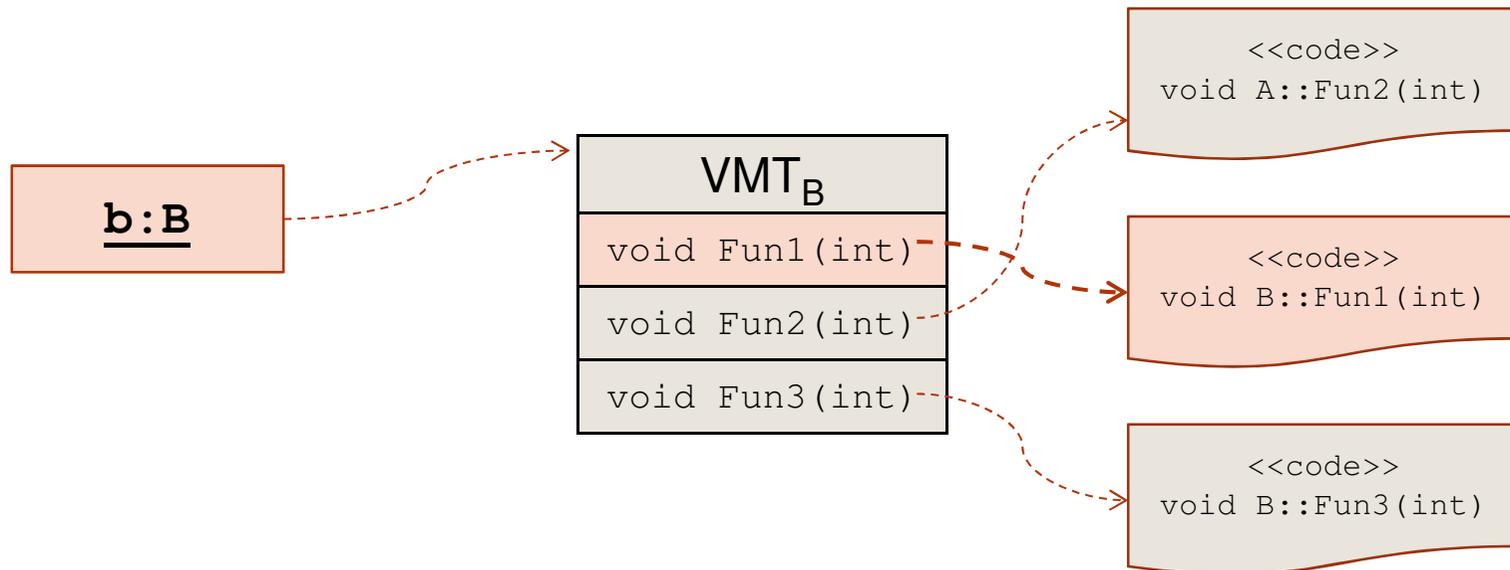
```
A x;
```

```
...
```

```
x = b;
```

```
...
```

```
x.Fun1(10);
```

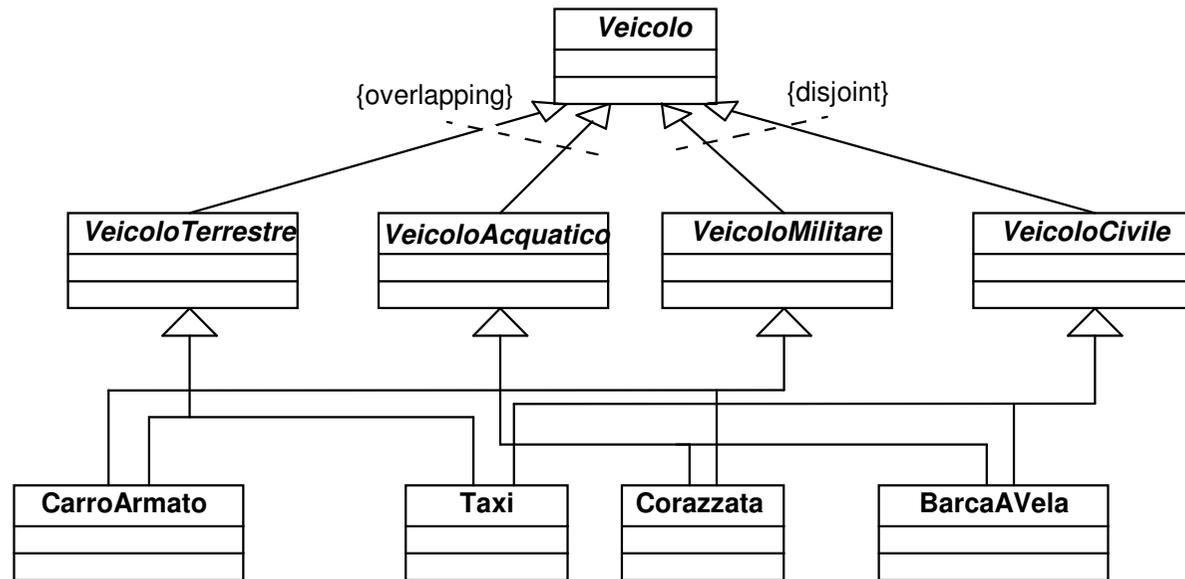


Ereditarietà

- **Ereditarietà semplice**
ogni classe della gerarchia deriva
 - da una e una sola superclasse (Java, .NET)
 - al più da una superclasse (C++)
 - la struttura che si ottiene è sempre un albero
- **Ereditarietà multipla**
almeno una classe della gerarchia deriva da 2+ superclassi (possibile in C++)
Se esistono antenati comuni
 - la struttura che si ottiene è un reticolo
 - si hanno conflitti di nome
 - la gestione può diventare molto complessa

Analisi Ereditarietà multipla

- Tra due o più classi di una gerarchia possono esistere dei vincoli {overlapping} o {disjoin}

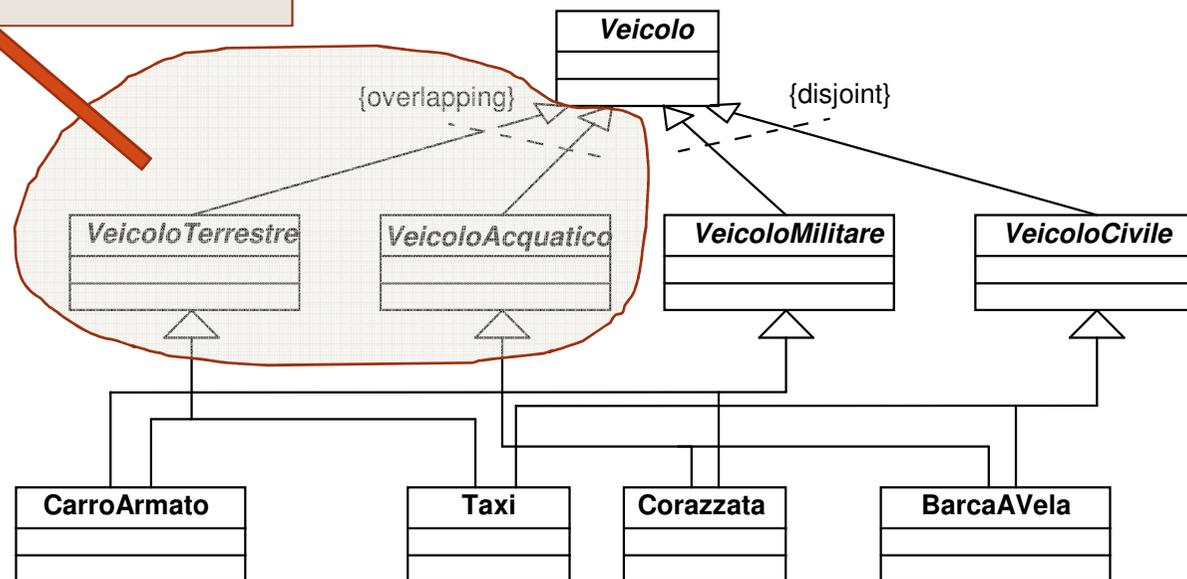


Un reticolo di veicoli

Progettazione

Da ereditarietà multipla a ereditarietà semplice

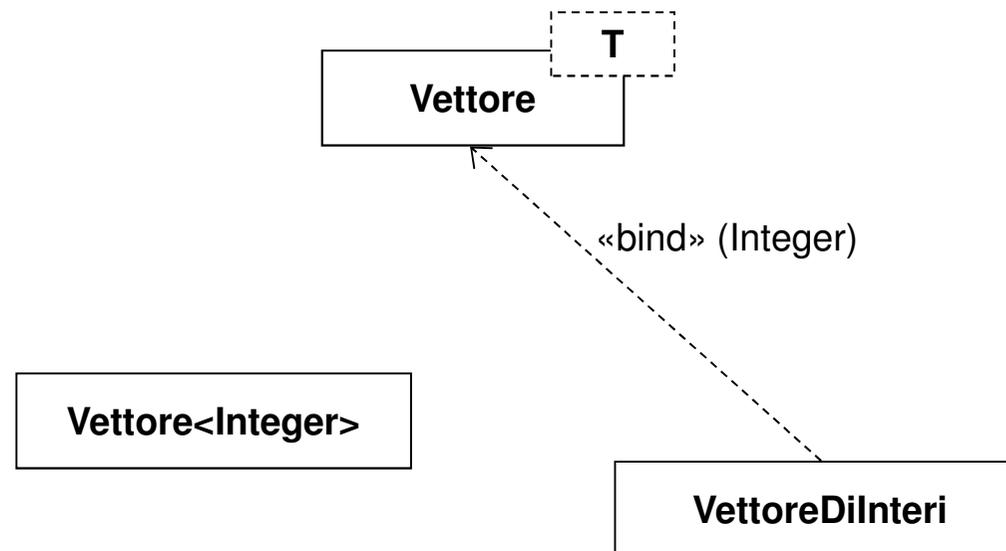
Interfacce
Composizione e delega
...



Un reticolo di veicoli

...alla programmazione generica rispetto ai tipi

- **Classe generica**: classe in cui uno o più **tipi** sono **parametrici**
- Ogni classe generata da una classe generica costituisce una **classe indipendente**
 - ▶ non esiste un legame di ereditarietà



Classe generica Stack (C#)

```
public class Stack<T>
{
    private T[] _array;
    private int _size;
    private const int _defaultCapacity = 4;
    private static T[] _emptyArray = new T[0];

    public Stack()
    {
        _array = _emptyArray;
        _size = 0;
    }

    public int Count
    {
        get { return _size; }
    }
}
```

Classe generica `Stack` (C#)

```
public void Push(T item)
{
    if (_size == _array.Length)
    {
        T[] destinationArray = new
            T[(_array.Length == 0) ? _defaultCapacity :
                (2 * _array.Length)];
        Array.Copy(_array, 0, destinationArray, 0, _size);
        _array = destinationArray;
    }
    _array[_size++] = item;
}
```

Classe generica `Stack` (C#)

```
public T Peek()
{
    if (_size == 0)
        throw new InvalidOperationException("_size == 0");
    return _array[_size - 1];
}

public T Pop()
{
    if (_size == 0)
        throw new InvalidOperationException("_size == 0");
    T local = _array[--_size];
    _array[_size] = default(T);
    return local;
}
```

Classe generica `Stack` (C#)

```
public void Clear()
{
    // Sets a range of elements in the System.Array
    // to zero, to false, or to null,
    // depending on the element type.
    Array.Clear(_array, 0, _size);
    _size = 0;
}
} // Stack<T>
```

Classe generica `Stack` (C#)

```
...
Stack<int> s1;
Stack<double> s2;
Stack<DateTime> s3;
Stack<Stack<int>> s4;
...
for (int j = 1; j <= 20; j++)
    s1.Push(j);
...
while (s1.Count > 0)
{
    int v = s1.Pop();
    // utilizzo di v
}
```

Regole di naming (.NET)

- I **nomi delle classi** devono
 - iniziare con una lettera maiuscola
 - indicare al singolare un oggetto della classe, oppure
 - indicare al plurale gli oggetti contenuti nella classe (se la classe è una classe contenitore)
- **Esempi**
 - Docente
 - Docenti (contiene una collezione di docenti)
 - CorsoDiStudio
 - CorsiDiStudio
 - AttivitaFormativa
 - AttivitàFormativa – accettato in C#