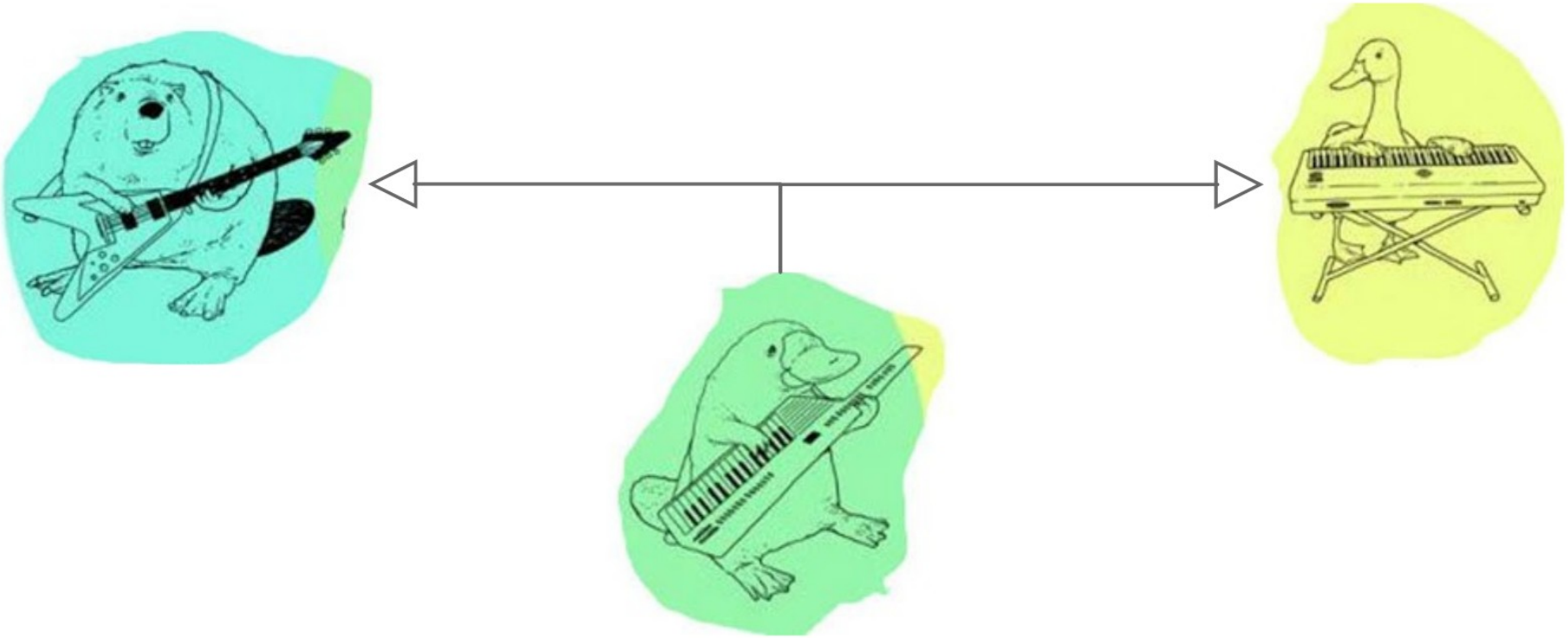
Metodi informatici della Fisica



C++ 10: ereditarietà

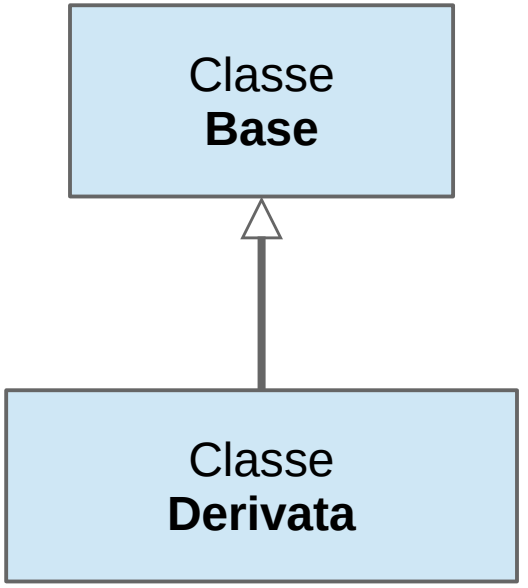
Susanna.Costanza@unipv.it
Andrea.Negri@unipv.it

Paradigmi OO

- **Classe**
  - Definizione di un nuovo tipo, unione di tipi già noti
  - Unificazione tra rappresentazione dei dati e metodi per la loro manipolazione
- **Incapsulamento**
  - Lo stato dell'oggetto (la rappresentazione) non è “esposto” all'utente, che ha accesso ad un'interfaccia pubblica decisa dalla sviluppatore
- **Ereditarietà**
  - Possibilità di derivare nuove classi da quelle già esistenti. Le classi derivate ereditano caratteristiche e proprietà delle classi base
- **Polimorfismo**
  - Possibilità di definire un singola interfaccia (per es il nome di un metodo) che viene mappata su molteplici definizioni in base al tipo dei parametri in gioco
  - Tre tipologie: overloading, template e polimorfismo run-time

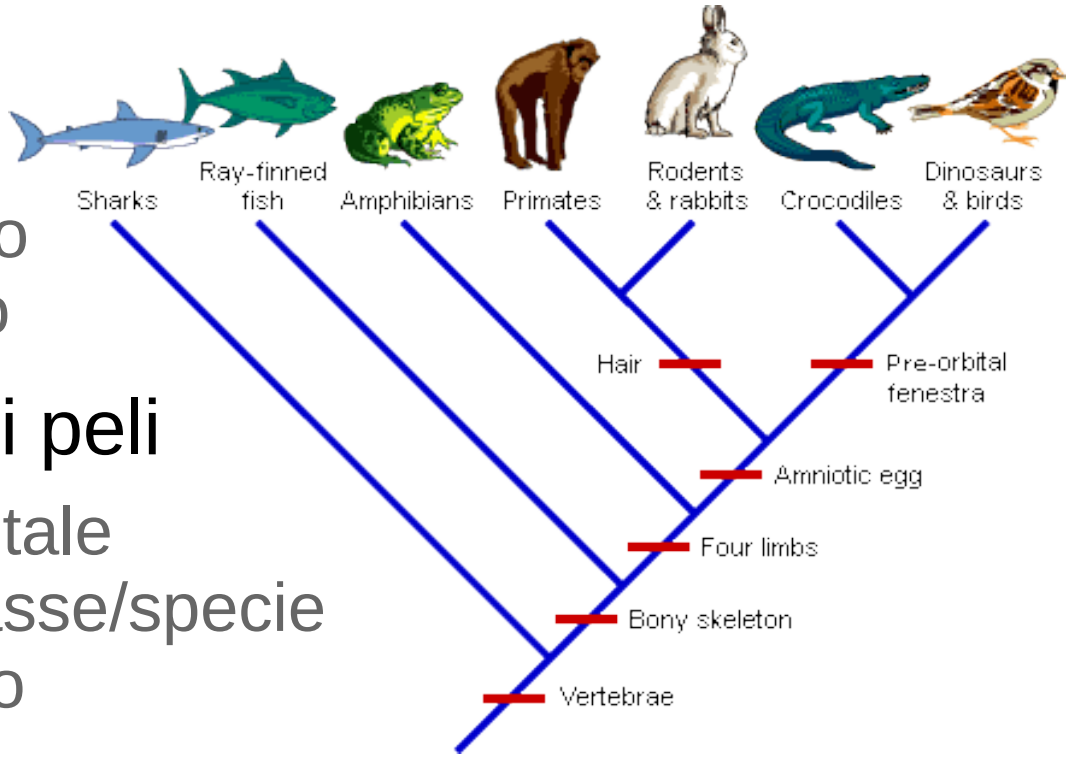
Ereditarietà

- L'**ereditarietà** è uno dei concetti fondamentali della programmazione Object Oriented
  - è una relazione di generalizzazione e/o specificazione tra classi
- Una classe base definisce una qualità generale
  - una classe da essa **derivata** rappresenta una variante specifica di tale qualità
- Invece che re-implementare le caratteristiche comuni
  - la classe derivata **eredita** le caratteristiche della classe base



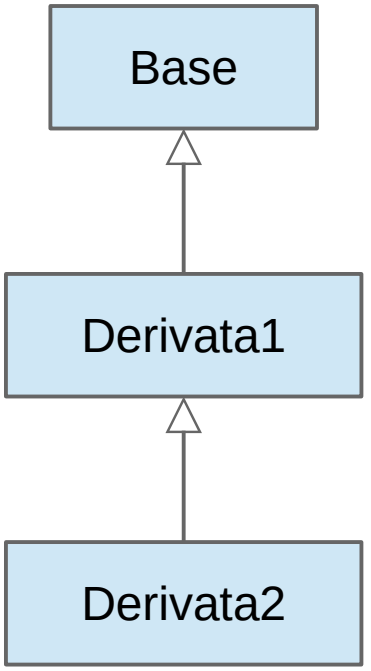
Ereditarietà

- Azzardando un parallelo tassonomico-cladistico
  - Ogni animale è un oggetto di una classe (specie)
  - Le proprietà caratteristiche di una certa specie sono state ereditate da antenati comuni
  - Con i vari rami che indicano una specifica caratteristica
- Gli animali mostrati sono tutti vertebrati
  - cioè hanno tutti ereditato dalla “classe” vertebrato
- conigli e primati hanno i peli
  - perché hanno ereditato tale caratteristica da una classe/specie base comune solo a loro



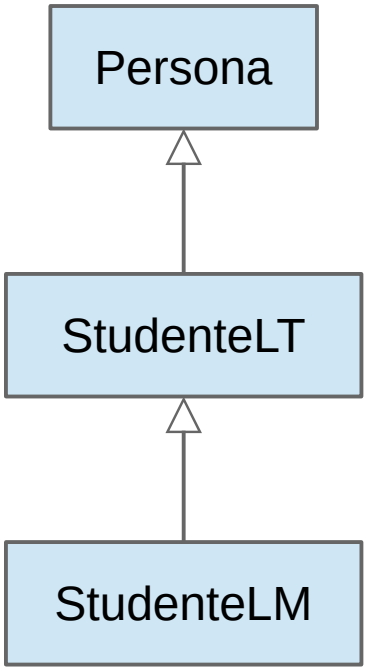
Ereditarietà

- L'ereditarietà facilita il riutilizzo del codice
  - Riducendo le duplicazioni
  - Diminuendo la possibilità di errori di codifica
  - Semplificando la gestione del codice
- Una classe derivata **eredita**
  - **tutti** i **dati** della classe da cui deriva
  - **tutti** i **metodi** della classe da cui deriva
- Cioè la classe derivata **non** deve ridichiarare tali caratteristiche
  - Un errore tipico (e grave) è ridefinire tali caratteristiche
  - NB: anche se non è detto che possa accedere a tali dati o che possa usare tali metodi



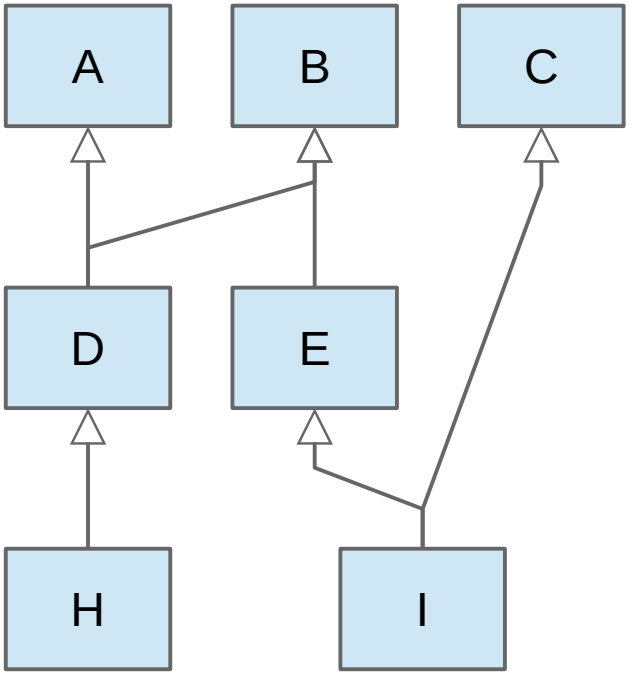
Esempio

- Classe base **Persona**
  - Dati membro: nome, cognome, CF, indirizzo
  - Metodi: costruttore, metodo per accedere al CF, metodo per calcolare l'età, ecc...
- Per definire un classe **StudenteLT** **non** si riparte da **zero**, ma si **eredita** da **Persona** **tutti** i **dati** e **tutti** i **metodi**
  - Si aggiungono solo le caratteristiche specifiche, es: il corso di laurea, l'anno di frequenza e il numero di matricola
- Similmente **StudenteLM** sarà uno **StudenteLT** con qualche caratteristica in più
  - Per esempio il titolo delle tesi triennale



Ereditarietà

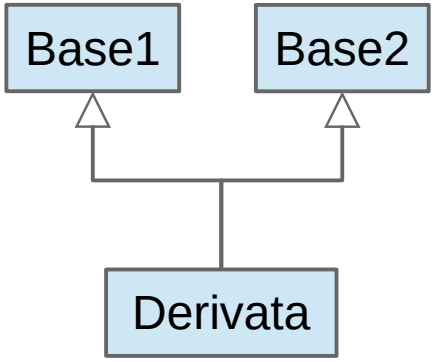
- L'ereditarietà permette quindi di realizzare una gerarchia di classi
  - A seconda del linguaggio può essere singola o multipla
- Ereditarietà **singola**
  - una classe può ereditare da una sola classe madre
  - in tal caso la gerarchia è un **albero**
  - come quello di un file system
- Ereditarietà **multipla**
  - una classe può anche derivare da più classi base
  - la gerarchia è ora un **reticolo**
  - è il caso delle classi C++



Ereditarietà e C++

- La sintassi per definire una classe derivata è

```
class Derivata : public Base1, public Base2, ecc
{
  // Interfaccia della
  // classe Derivata
};
```

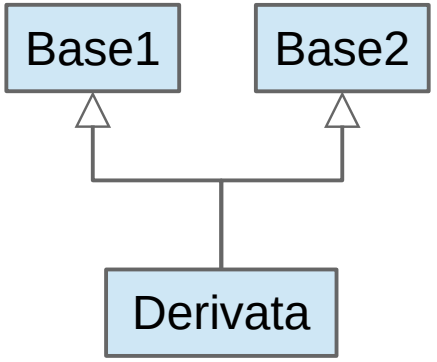


- Cioè
  - La solita sintassi di dichiarazione di una classe
    `class NomeDellaNuovaClasse`
  - Seguita da un “**:**”
  - Quindi una lista, separata da virgole, di classi base da cui la si vuole far derivare (di solito una)
  - E infine l’interfaccia specifica della classe {...}

Ereditarietà e C++

- La sintassi per definire una classe derivata è

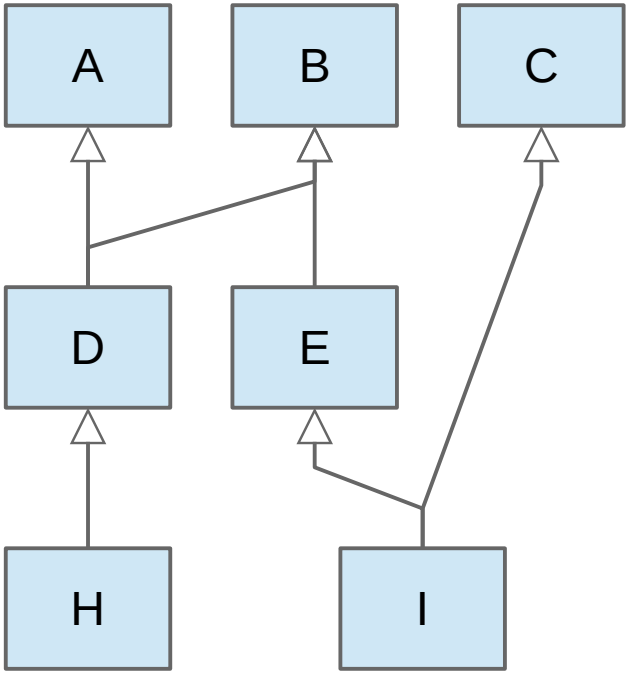
```
class Derivata : public Base1, public Base2, ecc
{
  // Interfaccia della
  // classe Derivata
};
```



- Per ora ignorate il ruolo dello specificatore **public**
  - Serve a definire quali **permessi** di accesso (ai membri della classe Base) possiede un oggetto della classe Derivata
  - Formalmente è possibile dichiarare un'eredità private o protected, ma si usa quasi sempre public

Ereditarietà e accesso

- La classe derivata eredita **tutto**
  - i dati membro delle sue classi base e delle eventuali classi base di queste ultime
  - i metodi, come sopra
- Cioè la **rappresentazione** di un oggetto di una classe derivata
  - è composto da **tutti** i dati membro presenti nelle classi del suo reticolo genealogico
  - Più, ovviamente, eventuali nuove caratteristiche
  - Analogia: una specie ha tutte le caratteristiche delle specie progenitrici



Esempio

```
class Persona
{
 public:
  Persona(string nn, string CF);
  int eta();
  string getNome();
 protected:
  string m_nome;
  string m_cf;
};
```

```
class Studente : public Persona
{
 public:
  Studente(string nn, string CF,
           string corso, int matr);
  int getMatr();
 protected:
  string m_corso;
  int    m_matr;
};
```

NB: Persona eredita m_nome, m_cf, eta(), getNome()

Ereditarietà e accesso

- Ma la classe derivata può accedere **solo** ai membri delle sue classi ancestrali che **non** siano stati dichiarati **privati**
  - Cioè eredita tutto, ma può accedere solo ai dati membro e metodi ereditati che sono public o protected
- **Protected** è un qualificatore di accesso specifico per l'ereditarietà: dati membro o metodi dichiarati protected
  - sono liberamente accedibili e modificabili dalle classi derivate, come se per loro fossero public
  - ma sono private per il resto del programma

Accesso esterno ai membri

- Fuori dalla classe, **protected** equivale a **private**
  - I dati e metodi private/protected non sono accessibili

```
#include <iostream>

// Classe minimale con 2 dati membro
// (1 private e 1 protected) e 3 metodi:
// 1 public, 1 protected e 1 private
class Esempio
{
 public:
  Esempio() : m_a(1), m_b(2) {;}
  void fPub()
  { std::cout << "fPub()" << std::endl;}
 protected:
  void fPro()
  { std::cout << "fPro()" << std::endl;}
  int m_a; // Dato membro protected
 private:
  void fPri()
  { std::cout << "fPri()" << std::endl;}
  int m_b; // Dato membro private
};
```

... prosegue a destra

... segue da sinistra

```
int main()
{
  // Oggetto della classe Esempio
  Esempio e;

  // Accesso via interfaccia pubblica
  e.fPub();  // OK

  // Le seguenti linee danno invece
  // errori di compilazione per
  // accesso esterno a dati membro
  // privare o protected
  e.m_a;    // KO
  e.m_b;    // KO
  e.fPro(); // KO
  e.fPri(); // KO

  return 0;
};
```

esempioEredit1.cxx

Accesso da classe derivata

- Per le classi derivate **protected** equivale a **public**
  - La classe derivata ha accesso a tutti i membri non private

```
#include <iostream>
// Classe minimale con 2 dati membro,
// un costruttore e altri 3 metodi
class Base
{
 public:
  Base() : m_a(9), m_b(7) {;}
  void fPub()
  { std::cout << "fPub()" << std::endl;}
 protected:
  void fPro()
  { std::cout << "fPro()" << std::endl;}
  int m_a; // Dato membro protected
 private:
  void fPri()
  { std::cout << "fPri()" << std::endl;}
  int m_b; // Dato membro private
};

... prosegue a destra
```

```
... segue da sinistra
class Derivata : public Base
{
 public:
  Derivata() : m_c(42) {;}
  void f()
  {
    fPro(); // OK: protected di Base
    m_a++;  // OK: protected di Base
    m_b++;  // KO!: privato di Base
    fPri(); // KO!: privato di Base
  }
 private:
  int m_c;
};
int main()
{
  Derivata d;//Oggetto classe Derivata
  d.f();
  return 0;
};
```

esempioEredit2.cxx

Ereditarietà ed overloading

- Le classi derivate possono fare l'**overloading** dei metodi non private che hanno ereditato
  - Cioè possono specializzare il metodo secondo necessità

```
#include <iostream>
// Semplice classe senza dati membro e
// con un solo metodo
class Nonna
{
 public:
  Nonna() {;}
  void f()
  { std::cout << "Nonna::f()" << std::endl;}
};
// Eredita da Nonna e fa l'overloading di f()
class Madre : public Nonna
{
 public:
  Madre() {;}
  void f()
  { std::cout << "Madre::f()" << std::endl;}
};
... prosegue a destra
```

```
... segue da sinistra
// Eredita da Madre ma non fa
// l'overloading di f()
class Figlia : public Madre
{
 public:
  Figlia() {;}
};

int main()
{
  Nonna  n;
  Madre  m;
  Figlia f;
  n.f(); // -> "Sono Nonna::f()"
  m.f(); // -> "Sono Madre::f()"
  f.f(); // -> "Sono Madre::f()"

  return 0;
};
```

esempioNonnaMadreFiglia.cxx

Ereditarietà ed overloading

- Le classi derivate possono fare l'**overloading** dei metodi non private che hanno ereditato
  - Se non c'è overload → usato il metodo della classe base

```
#include <iostream>
// Semplice classe senza dati membro e
// con un solo metodo
class Nonna
{
 public:
  Nonna() {;}
  void f()
  { std::cout << "Nonna::f()" << std::endl;}
};
// Eredita da Nonna e fa l'overloading di f()
class Madre : public Nonna
{
 public:
  Madre() {;}
  void f()
  { std::cout << "Madre::f()" << std::endl;}
};
... prosegue a destra
```

```
... segue da sinistra
// Eredita da Madre ma non fa
// l'overloading di f()
class Figlia : public Madre
{
 public:
  Figlia() {;}
};

int main()
{
  Nonna  n;
  Madre  m;
  Figlia f;
  n.f(); // -> "Sono Nonna::f()"
  m.f(); // -> "Sono Madre::f()"
  f.f(); // -> "Sono Madre::f()"

  return 0;
};
```

esempioNonnaMadreFiglia.cxx

Specificatore di ereditarietà

- Dichiarando una classe derivata è possibile specificare le modalità di accesso un oggetto della classe derivata ai membri della classe base
  - La derivazione **public** non cambia gli accessi (è il default)
  - La derivazione **protected** nasconde i membri public della base
  - La derivazione **private** nasconde l'intera interfaccia di base sia all'esterno della classe derivata che nelle eventuali classi derivate da classe base

`class Base`	`class Derivata : public Base`	`class Derivata : protected Base`	`class Derivata: private Base`
`public`: Accessibili dalle funzioni membro e dall'esterno	I membri public di Base sono public della Derivata	I membri public di Base sono protected della Derivata	I membri public di Base sono private della Derivata
`protected`: Accessibili solo dalle funzioni membro	I membri protected di Base sono protected della Derivata	I membri protected di Base sono protected della Derivata	I membri protected di Base sono private della Derivata
`private`: Accessibili solo dalle funzioni membro	I membri private di Base inaccessibili nella Derivata	I membri private di Base sono inaccessibili nella Derivata	I membri private di Base sono inaccessibili nella Derivata

Design

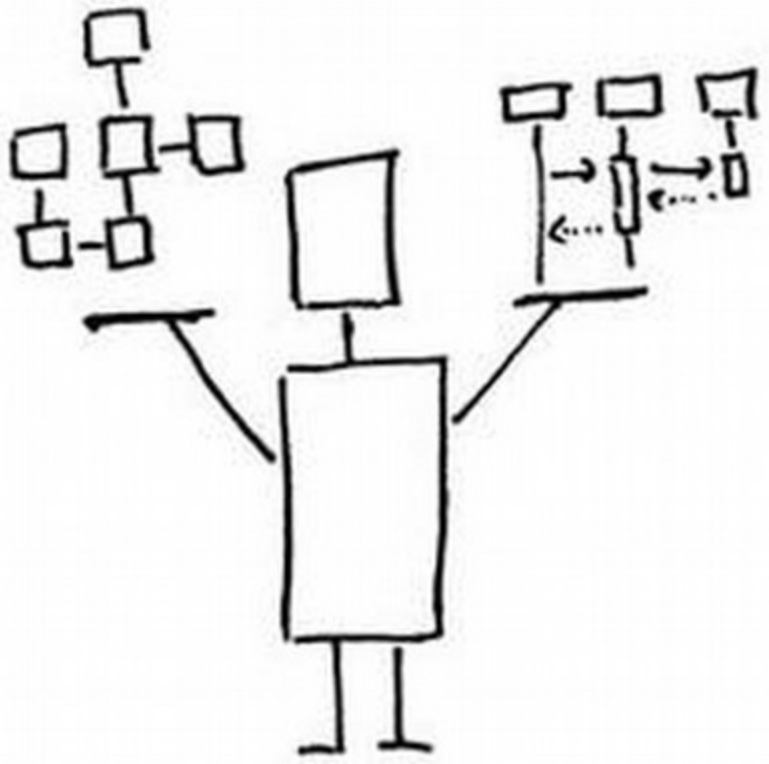
- Nello sviluppo di qualsiasi programma è sempre consigliabile partire da un **design** che
  - consideri tutti gli aspetti del problema
  - tenga conto a priori delle possibili **evoluzioni** del programma: in futuro nuove requisiti potrebbero richiedere sostanziali modifiche del codice
- Senza fase di design si rischia di produrre "**spaghetti code**"
  - la struttura del codice è talmente intrecciata da impedire ulteriori sviluppi
  - nuovi requisiti richiederebbero una completa riscrittura

Design

- Nei linguaggi strutturati l'utilizzo delle funzioni comporta la necessità di una fase di design
  - Sforzo ripagato da maggior leggibilità e gestibilità
  - Ma rimane a discrezione dell'utente
- Nella programmazione OO la fase di design diventa in pratica **obbligatoria**
  - il programmatore è obbligato ad analizzare il problema in termini di classi e relazioni tra di esse
  - deve in pratica creare diagrammi (UML)

Design

- Il **design** diventa la fase principale della programmazione e richiede il **90%** dello sforzo
  - occorre identificare le classi necessarie
  - trovare la migliore gerarchia di ereditarietà
  - definire cosa rendere visibile alle classi derivate e quali margini di azione lasciare a valle dello sviluppo
  - valutare uso di template
  - ….
- Fase di scrittura del **codice**
  - **10%** dello sforzo



Nei grandi progetti

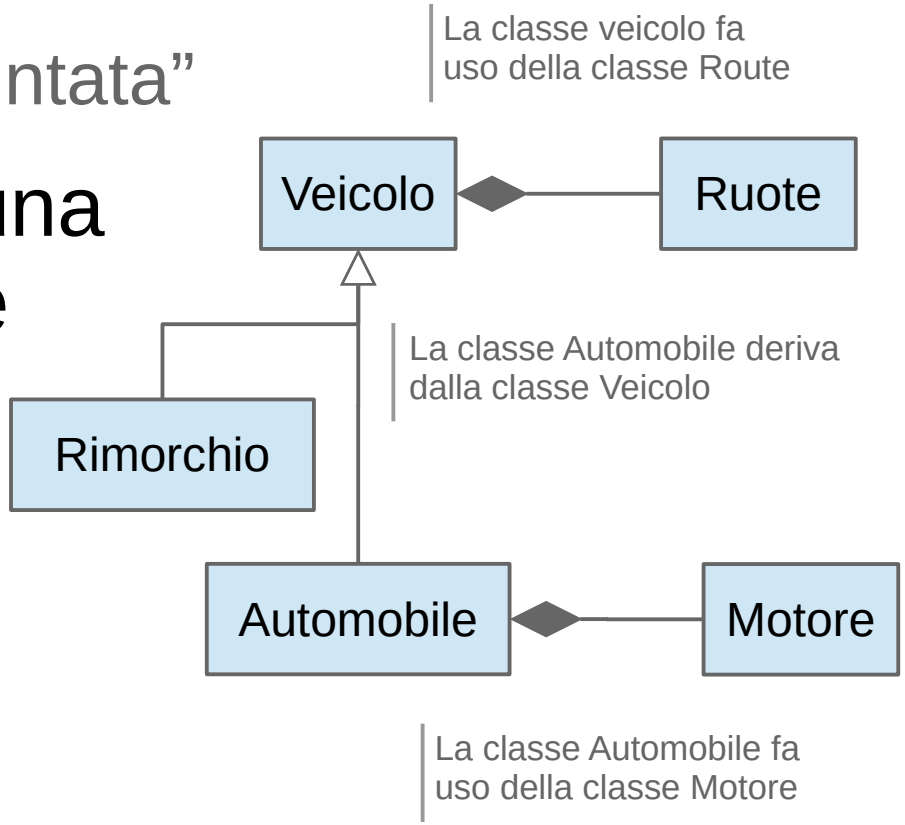
- I programmatori più esperti (sw arch
  - definiscono l'architettura del sistema
  - sviluppano le classi base
- Il resto dei programmatori
  - implementa il codice
  - e sviluppa le classi derivate
  - nell'ambito delle condizioni al contorno definite dai “sw architect”
- I paradigmi OO permettono di definire a priori
  - gli spazi per i possibili sviluppi
  - quale libertà di manovra lasciare nelle classi derivate

UML

- **U**niform **M**odeling **L**anguage
  - un linguaggio di modellazione per la programmazione orientata agli oggetti
- Utile per rappresentare graficamente
  - le classi: dati membro e metodi
  - le relazioni tra di esse
- Esistono vari tipi di diagrammi UML
  - qui vedremo brevemente solo i “class diagrams”

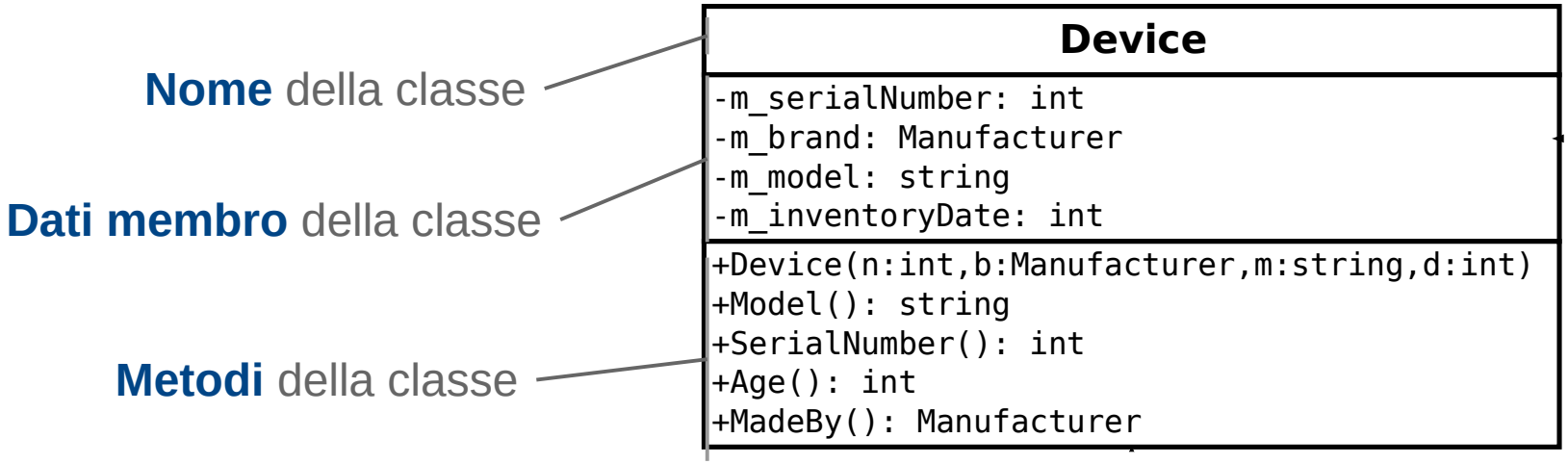
UML: class diagram

- Un **rettangolo**: rappresenta una classe
  - all'interno è possibile specificate i membri
- Freccia a **triangolo**: indica una relazione di ereditarietà tra classi
  - la classe base è quella "puntata"
- Freccia a **rombo**: indica una relazione di composizione
  - cioè una classe contiene oggetti di un'altra classe
  - Il rombo è dal lato della classe ospitante



UML Class Diagrams

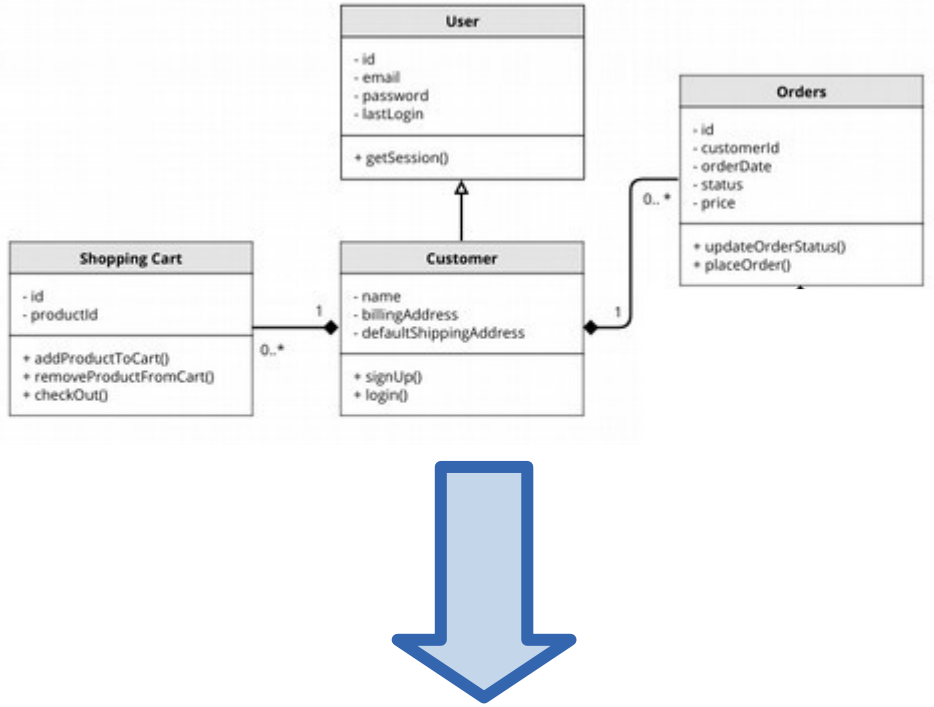
- La sintassi per descrivere dati membro e metodi
  - `Accesso NomeMembro : tipo`



- Il segno iniziale specifica la modalità di accesso al membro
  - "**-**" è private, "**#**" è protected e "**+**" è public
- Il tipo segue il nome del dato/metodo preceduto da un "**:**"
- Esempio: -m_model : string
  - m_model è un dato privato (-) di tipo string

UML Class Diagrams

- Esistono programmi che a partire da diagrammi UML **generano** automaticamente il codice in vari linguaggi di programmazione
  - Es: java, C++, Fortran90, VisualBasic, ecc
- Generano **solo** l'interfaccia
  - cioè le dichiarazioni, ospitate negli header file
  - **non** l'implementazione



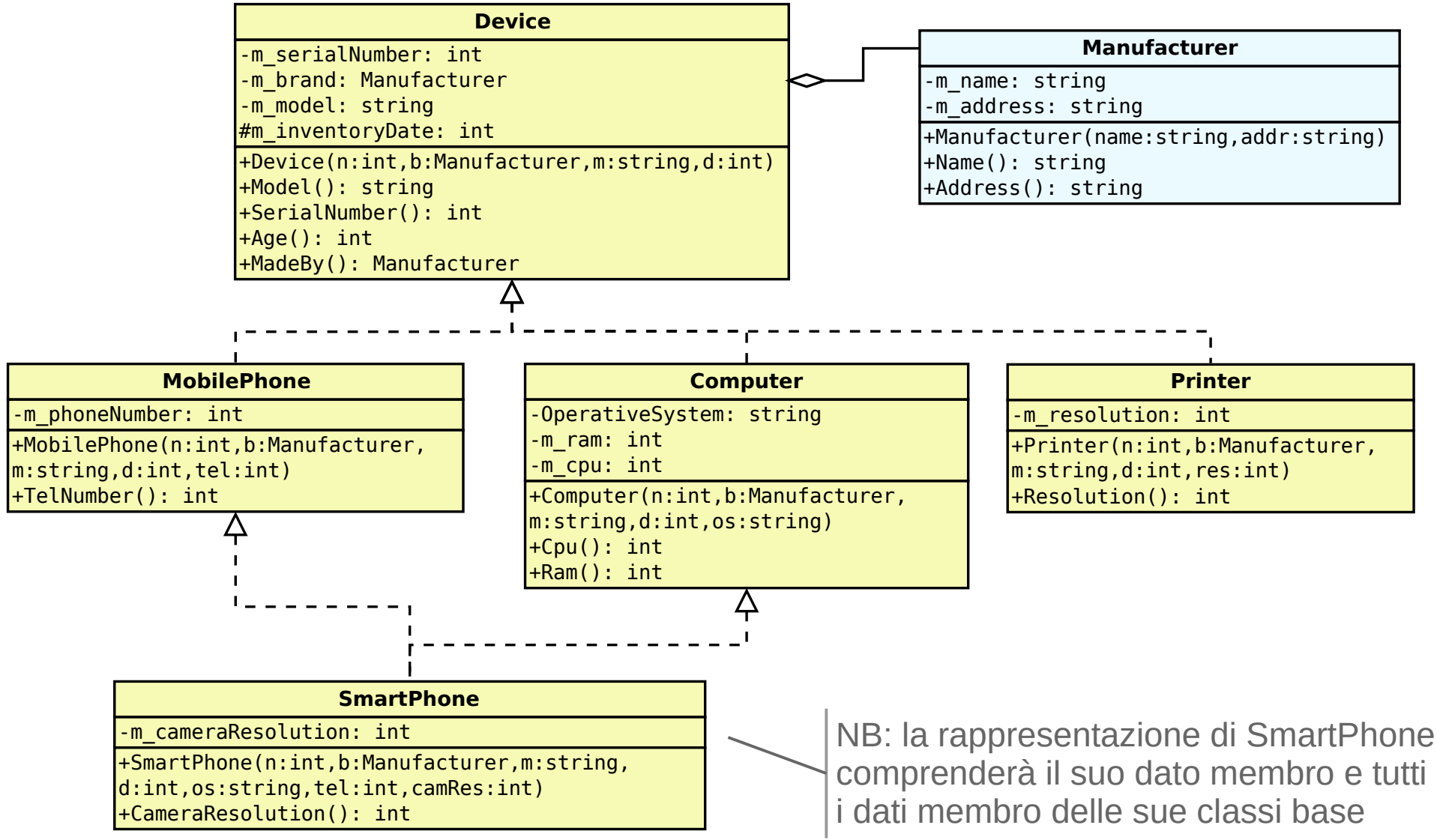
Terminal

File Edit View Search Terminal Help

```
#include "Vettore.h"
#include <vector>

const double G = 6.672e-11;

class CorpoCeleste
{
 public:
  CorpoCeleste(std::string nome, double massa, Vettore pos, Vettore vel);

  Vettore pos();
  Vettore vel();
  double  massa();
  std::string nome();

  virtual void evolviDeltaT(double dT, std::vector<CorpoCeleste*> &corpi);

 protected:
  Vettore     m_pos;
  Vettore     m_vel;
  double      m_massa;
  std::string m_nome;
};
```

7,1 Top

UML: esempio



NB: la rappresentazione di SmartPhone comprenderà il suo dato membro e tutti i dati membro delle sue classi base

Riassunto

- L'**ereditarietà** permette di realizzare una gerarchia di classi
  - Con le classi derivate che ereditano **tutte** le caratteristiche delle classi base
  - **Mai** ridefinire i dati membro nelle classi derivate
- Nelle derivate si può ridefinire (specializzare) metodi delle classi base (**overloading**)
  - Se le condizioni di accesso lo permettono
  - Cioè se il metodo è "visto" come public o protected
- Occorre sempre partire dalla fase di **design**
  - Architetto → Geometra → Muratore